TP de génie logiciel

Yannick Loiseau yannick.loiseau@univ-bpclermont.fr

TP 1

Gestion de version avec Git

Merci à Pierre-Antoine Papon *pour son aide dans l'élaboration de ce TP.*

1.1 Présentation

Un système de gestion de versions est un logiciel qui permet de stocker un ensemble de fichiers tout en gardant une chronologie de toutes les modifications qui ont été effectuées dessus. Cela offre la possibilité de récupérer d'anciennes versions de documents et d'examiner l'historique des modifications. Un tel logiciel permet également de travailler de manière distribuée (il se comporte en ce sens comme un serveur de fichiers) afin que des personnes distantes puissent collaborer sur un même projet. Pour que la collaboration de plusieurs personnes soit efficace, une fonctionnalité permet de gérer les conflits lors de modifications concurrentes de la même partie d'un fichier.

Il existe deux philosophies pour la gestion de versions : l'une utilisant un système centralisé et l'autre un système décentralisé. Dans un gestionnaire de versions centralisées (exemple : CVS et Subversion[SVN]), on ne retrouve qu'un seul dépôt des versions. Un serveur va donc centraliser tous les fichiers constituant un projet et contenir l'historique des versions. Dans ce système, l'utilisateur va devoir récupérer sur son poste de travail une version (généralement la dernière) des fichiers et va toujours travailler sur une version locale du projet. Il soumettra ensuite ses modifications au serveur pour que le dépôt central soit modifié. Un tel système va simplifier la gestion des versions mais peut s'avérer contraignant notamment lors d'un travail sans connexion au réseau. Par opposition, dans un gestionnaire de versions décentralisées (exemple : Git[Git], Mercurial[Hg] et Bazaar[BZR]), chaque utilisateur va posséder sa propre copie du dépôt contenant les fichiers et l'historique complet. Il sera donc plus long de cloner un dépôt que de récupérer une version courante, puisque tout l'historique sera récupéré. Cependant l'utilisateur pourra ensuite travailler tranquillement en local et en cas de panne du serveur central, aucune donnée ne sera perdue puisque chaque contributeur possède l'ensemble du dépôt. La majorité des opérations se feront en local, et l'utilisateur se connectera au serveur central que pour soumettre son travail aux autres collaborateurs.

Git est un système de gestion de versions décentralisé créé par Linus Torvalds pour la gestion du noyeau Linux. Ce TP est une introduction à l'utilisation de ce logiciel dans le contexte du développement, mais il peut évidemment s'appliquer à la gestion de tous types de documents, principalement textuels.

1.2 Démarche

Les fichiers sont stockés dans un dépôt central. Ce dépôt peut être situé sur un serveur distant, ou sur la machine locale. Dans ce TP, nous allons créer un dépôt local pour des raisons pratiques. Cependant, dans le cas général, le dépôt est le plus souvent situé sur un serveur, ce qui permet facilement de partager le contenu entre différents utilisateurs et entre différents postes de travail. Dans ce cas, l'accès au serveur peut se faire soit à l'aide d'un protocole spécifique à git, soit encapsulé dans du ssh ou du HTTP pour être accessible plus facilement.

Afin de travailler, il faut récupérer une copie locale de ceux-ci ; c'est la *copie de travail*, sur laquelle les modifications seront apportées. Une fois les modifications effectuées, il faut transmettre celles-ci au dépôt (*commit*), qui mettra à jour ses versions si tout se passe bien, ou refusera s'il y a un conflit qu'il ne peut résoudre seul. Afin de faciliter la gestion des versions, il est très fortement recommandé de faire une description des modifications apportées. On peut en effet afficher un historique des modifications, et récupérer n'importe quelle version du projet.

Il est ainsi conseillé de mettre à jour la version du dépôt (i.e. transmettre les modifications) fréquemment, par petites étapes. Par exemple, dans le cas d'un programme, on implémente une fonction, on la teste, on transmet ; on implémente une autre fonction, on la teste, on transmet, etc. Ainsi, on minimise les risques de conflits et on facilite le suivi. Il faut dans la mesure du possible ne transmettre que des versions *valides*, c'est-à-dire testées pour du code par exemple, afin de garder la version du dépôt dans un état stable et fonctionnel.

Avant de commencer à travailler sur une copie de travail, il faut bien évidemment penser à la mettre à jour (sauf si on vient de la créer), afin d'intégrer les dernières modifications éventuellement effectuées par les autres membres du projet.

1. Récupération de la copie de travail
2. Modifications
3. Mise à jour de la version de travail
4. Résolution éventuelle de conflits
5. Vérification des modifications
6. Transmission des modifications
7. Mise à jour de la version de travail
8. Modifications
9. ...

1.3 Utilisation de Git

Avant de commencer, créons nous un environnement de travail. Dans un terminal, tapez les commandes suivantes :

```
mkdir -p TPGenieLog/Git/
cd TPGenieLog/Git/
```

Puis configurons git :

```
git config --global user.name "Prenom Nom"
git config --global user.email "nom.prenom@dns.tl"
```

Utilisez la commande suivante pour vérifier votre configuration actuelle :

```
git config --list
```

Commandes de bases

Le programme de base (client) ne fonctionne qu'en ligne de commande. Il existe cependant de nombreuses interfaces graphiques ou d'extensions pour les explorateurs de fichiers ou les interfaces de développement ([gitGUI ; tortoisegit ; egit]). Référez-vous au manuel pour plus de détails sur les commandes, ou faites `git help` en ligne de commande.

Création d'un dépôt

Commençons par initialiser un projet en créant un dépôt vide avec la commande `init`. Nous allons également configurer ce dépôt afin qu'il accepte des `push` locaux (inutile dans la majorité des cas) :

```
git init
git config receive.denyCurrentBranch ignore
```

Dans le cas d'un dépôt situé sur une autre machine que celle de travail (cas d'un serveur distant), seule la première commande est nécessaire, mais elle doit être exécutée sur le serveur en question.

Cette commande crée un dépôt git dans le répertoire `.git`. Ce répertoire contient toutes les informations nécessaires à git pour gérer le dépôt ; les données ne doivent donc *jamais* être éditées à la main (sauf si l'on sait vraiment ce que l'on fait !). Il contient un ensemble de répertoires et de fichiers :

`branches/` contenait auparavant des raccourcis (obsolète),

`config` contient les options de configuration spécifique au projet,

`description` est un fichier utilisé pour le gui gitweb afin de parcourir le répertoire via un navigateur web,

`HEAD` pointe sur la branche courante. Dans la plupart des cas, il s'agit de refs/heads/-master,

`hooks/` contient des scripts qui sont exécutés à certains moments lorsque vous travaillez avec git, comme après un `commit` ou avant un `rebase`,

`info/` contient notamment un fichier contenant des règles d'exclusion pour ignorer le versionnage de certains fichiers dans le *dépôt local*. Attention ce n'est pas la même chose qu'un `.gitignore` qui sera partagé avec les tous contributeurs,

`objects/` contient tout le contenu de votre base de données,

`refs/` contient toutes les références de votre dépôt, que ce soit les stashes, tags, les branches distantes et locales.

Vous pouvez examiner le contenu des répertoires et des fichiers pour mieux comprendre comment est stockée l'information.

Commandes de bases

À tout moment, vous pouvez consulter l'état du répertoire de travail :

```
git status
```

Commencer par créer plusieurs fichiers puis consulter à nouveau l'état du répertoire :

```
echo "mon super code" > code.txt
echo "des tests"  > test1.txt
echo "encore des tests" > test2.txt
git status
```

Vous pouvez commencer à suivre un fichier en utilisant la commande `add`. Vérifier ensuite l'état pour voir ce qui a changé :

```
git add code.txt
git status
```

Vous pouvez également suivre l'ensemble des fichiers non suivis en utilisant la commande `add .` puis vérifier encore une fois l'état du répertoire.

```
git add .
git status
```

Afin de valider les modifications sur le dépôt, utilisez la commande `commit`. Vérifier ensuite l'état du répertoire.

```
git commit -m "init repo"
git status
```

La commande `log` permet d'examiner l'historique des modifications :

```
git log
```

Modifier ensuite le fichier `test1`. Puis vérifier l'état du répertoire. Vous pouvez ensuite utiliser la commande `diff` pour consulter les différences. Elle montre donc les lignes exactes qui ont été ajoutées, modifiées ou effacées.

```
echo "plus de tests" >> test1.txt
git status
git diff
```

Valider ensuite vos modifications. Le `-a` évite d'avoir à retaper la commande `add`. Examiner à nouveau l'historique des modifications.

```
git commit -a -m "modify tst1"
git log
```

Malheureusement vous avez fait une erreur dans votre message de commit. Vous pouvez cependant modifier votre message avec le paramètre `amend` de la commande `commit`. Vous pouvez ensuite vérifier que le message a bien changé avec `log`.

```
git commit --amend -m "modify test1"
git log
```

Finalement votre dernière modification sur le fichier `test1` ne vous convient plus. Vous pouvez annuler le commit avec `revert` mais que se passe-t-il réellement ? Vérifier avec `log`.

```
git revert HEAD
# quitter vi avec  :q
git log
```

Pour revenir plusieurs révisions en arrière, il faut le spécifier derrière le HEAD avec un tilde . Essayer la commande suivante :

```
git revert HEAD~2
# quitter vi avec  :q
git log
ls
```

Que s'est-il passé ? Avant de continuer, faites un `revert` de la dernière révision.

Supposons maintenant que vous voulez renommer le fichier `test2` en `test`. Utiliser la commande `mv` :

```
git mv test2.txt test.txt
git status
ls
git commit -m "rename test2 in test"
git log
```

Vous pouvez utiliser la même commande pour déplacer un fichier dans un autre répertoire.

Maintenant supprimer le fichier `test` et enlevons le du suivi avec `rm` :

```
git rm test.txt
ls
git status
git commit -m "delete test"
git log
```

Avant de finir les commandes de bases, juste une petite précision pour signaler que git offre la possibilité d'exclure certains fichiers du suivi. Cela peut être utile par exemple en C pour éviter de mettre les fichiers `.o` qui sont générés lors de la compilation. Pour cela, il faut créer un fichier `.gitignore` et y spécifier les fichiers qu'il doit ignorer. Dans le cas des fichiers `.o` il faudra ajouter l'instruction `*.o` dans notre fichier `.gitignore`.

Gérer plusieurs branches

Les branches permettent de contenir des versions déviant de la ligne de développement principale, comme des expérimentations de solutions qui seront plus tard éventuellement fusionnées avec la branche principale, ou des versions antérieures encore maintenues.

À l'initialisation du dépôt, une première branche est créée. C'est la branche *master* qui va contenir les sources principales et qui évolue lors du développement du projet. On peut vérifier les branches existantes avec la commande `branch` :

```
git branch
```

L'utilisateur peut rajouter une branche quand il en a besoin en appelant encore la commande `branch` mais cette fois en lui passant en paramètre le nom de la nouvelle branche à créer. Vérifier ensuite que la nouvelle branche a bien été créée.

```
git branch experimental
git branch
```

L'astérisque `*` devant le nom de la branche vous signale la branche sur laquelle vous travaillez. Avant de continuer votre développement, passez sur la nouvelle branche que vous venez de créer avec `checkout` puis vérifiez bien que vous avez changé de branche.

```
git checkout experimental
git branch
```

Nous allons ensuite recréer le fichier `test2` dans cette nouvelle branche et propager la modification :

```
echo "encore des tests" > test2.txt
git add .
git status
git commit -m "add test2"
git log
```

Retourner sur la branche *master* pour voir ce qu'il s'est passé :

```
git checkout master
git branch
```

On utilise à nouveau `log` pour examiner l'historique des modifications :

```
git log
ls
```

Il ne s'est rien passé, même le `ls` ne voit pas apparaître le fichier `test2`... Mais c'est normal puisque nous avons créé le fichier `test2` dans la branche *experimental* et non dans la branche principale *master*. Les branches sont donc bien indépendantes les unes des autres. Nous avons maintenant fini de travailler sur la branche *experimental*, et nous voulons fusionner cette branche dans la branche *master*, nous utilisons donc la commande `merge` :

```
git merge experimental
git log
ls
```

La fusion s'est bien passée puisqu'il n'y a pas eu de conflit. Le fichier `test2` est maintenant dans la branche *master*. Nous pouvons maintenant supprimer la branche *experimental* qui ne sert plus. Pour se faire, nous allons utiliser la commande :

```
git branch -d experimental
git branch
```

La branche *experimental* est bien supprimée. Mais que ce serait-il passé s'il y avait eu un conflit. Essayons d'en créer un. Créons une nouvelle branche *conflict* puis modifions le fichier `test1` sur cette branche :

```
git branch conflict
git branch
git checkout conflict
git branch
echo "plus de tests" >> test1.txt
git commit -a -m "modify test1 from conflict"
git log
```

Repassons maintenant sur la branche *master* et modifions à présent le fichier `test1` :

```
git checkout master
git branch
echo "moins de tests" >> test1.txt
git commit -a -m "modify test1 from master"
git log
```

Maintenant essayons de fusionner :

```
git merge conflict
```

Nous obtenons le message suivant :

```
Auto-merging test1.txt
CONFLICT (content) : Merge conflict in test1.txt
Automatic merge failed ; fix conflicts and then commit the result.
```

La commande diff nous permet de voir d'où vient le problème. Il faut ensuite éditer le fichier pour résoudre le conflit puisqu'il ne sait pas « dans quel ordre ranger » les deux lignes ajoutées :

```
git diff
vi test1.txt
# modifier le document manuellement (i pour passer en mode insertion)
# faites Esc pour quitter le mode insertion
# et  :wq pour enregistrer le document et quitter vi
git status
git commit -a -m "edit manually to merge master and conflict"
git log
```

On peut ensuite supprimer la branche *conflict* :

```
git branch -d conflict
git branch
```

Collaborer avec d'autres personnes

Nous allons maintenant voir les bases pour collaborer à plusieurs sur un même projet. Commençons par voir le fonctionnement de la commande clone qui permet de cloner un dépôt déjà existant. Replacez-vous dans le répertoire TPGenieLog que vous avez créé au début de ce TP. N'ayant pas de dépôt distant, nous allons cloner le répertoire Git sur lequel vous venez de travailler. Nous devons changer le nom du clone pour ne pas avoir deux répertoires ayant le même nom.

```
cd ..
git clone Git Git2
```

```
cd Git2
ls
git log
```

À la suite de cette manipulation, nous pouvons constater que le dépôt a bien été cloné correctement. Retournons sur le dépôt Git puis modifions le fichier test1 :

```
cd ../Git
echo "encore plus de tests" >> test1.txt
git status
git commit -a -m "modify test1"
git log
```

Retournons sur le dépôt Git2 et regardons le fichier test1 :

```
cd ../Git2/
cat test1.txt
```

Le fichier n'a pas changé (normal). Nous allons maintenant récupérer les modifications apportées sur le dépôt Git avec la commande pull :

```
git pull origin
cat test1.txt
git log
```

Quand on clone un dépôt, git crée une référence à ce dépôt sous le nom *origin* pour pouvoir s'y référencer plus facilement par la suite.

Nous allons maintenant apporter une modification à notre dépôt Git2 puis la propager dans le dépôt Git. Pour se faire, nous allons utiliser la commande push :

```
echo "test pour le push" > test3.txt
git status
git add .
git commit -m "add test3"
git log
git pull origin
git push origin
```

Normalement, il faut toujours faire un pull avant de faire un push afin de s'assurer qu'il n'y a pas eu de modification sur le dépôt distant entre temps. Vérifions dans le dépôt Git que la modification a bien été propagée :

```
cd ../Git
git log
ls
```

C'est bon !

1.4 Conclusion

Nous avons vu dans ce TP quelques possibilités offertes par Git mais nous sommes encore loin de maîtriser complètement ce gestionnaire de versions. Toutefois, ces premières bases vous permettront de commencer à l'utiliser dans vos projets personnels comme professionnels. Pour approfondir votre lecture, vous pouvez lire le livre [Cha09] qui est sous licence Creative Commons. Et si les lignes de commandes vous font peur, pensez à utiliser une interface graphique ou le plugin git directement dans votre IDE préféré. Les sites GitHub[GitHub] ou Bitbucket[Bitbucket] permettent de créer des dépôts git en ligne gratuitement alors ne vous en privez pas.

Pour aller plus loin, vous pouvez explorer les différents « workflow » [1] git, qui décrivent la manière d'organiser le travail collaboratif en utilisant différentes branches git.

Vous pouvez également explorer les fonctionnalités avancées de git, comme la manipulation de l'historique avec la commande `rebase`.

1. par exemple http://nvie.com/posts/a-successful-git-branching-model/

TP 2

Les outils de « build »

Merci à Sylvain Desgrais *pour son aide dans l'élaboration de ce TP.*

2.1 Introduction

Nous allons voir dans ce TP un aperçu des outils pour construire des projets de type « code ». Ces outils sont nombreux [1] (presque autant qu'il existe de langages de programmation), mais ils sont indispensables dès que le projet « grossit », que ce soit en nombre de développeurs ou en nombre de lignes de code.

Chaque « builder » a ses avantages et ses inconvénients, c'est pourquoi il est important de bien le choisir au début du projet.

2.2 Make

Make est un programme qui permet d'automatiser un ensemble de tâches qui sont décrites dans un fichier appelé Makefile.

C'est l'un des plus anciens gestionnaires de compilation mais aussi l'un des plus utilisés aujourd'hui.

Sa capacité à exécuter des commandes shell le rend approprié à compiler n'importe quel type de projet, d'en organiser le déploiement, de créer la documentation relative au projet, etc...

Une tâche possède un nom (la cible) et une liste de dépendances. Une dépendance peut être une autre tâche ou un fichier sur le disque.

Le projet

Créer un répertoire projet dans votre répertoire personnel. Créer ensuite deux fichiers function.h et function.c dans ce répertoire projet qui contiennent respectivement les sources suivantes :

1. Voir par exemple https://en.wikipedia.org/wiki/List_of_build_automation_software

Listing 2.1 – function.h

```
#include <stdio.h>
#include <stdlib.h>
int function(char * name);
```

Listing 2.2 – function.c

```
#include <stdio.h>
#include <stdlib.h>
#include "function.h"
int function(char * name){
      printf("hello %s !\n",name);
      return 0;
}
```

Créer maintenant le fichier main.c qui contient le source suivant :

Listing 2.3 – main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "function.h"
int main(int argc, char ** argv){
      function(argv[1]);
      return 0;
}
```

Nous allons maintenant compiler ce projet « à la main » :

Listing 2.4 – compilation à la main dans un shell

```
$ gcc -c function.c -o function.o
$ gcc -c main.c -o main.o
$ gcc main.o function.o -o example
$ ./example $USER
hello sylvain !
```

Avant de continuer, penser à effacer les fichiers .o et l'exécutable.

Premier Makefile

Créer un fichier Makefile comme indiqué ci-dessous :

Listing 2.5 – Makefile

```

all : main.o function.o
```

```
	gcc main.o function.o -o example

main.o : main.c
	gcc -c main.c -o main.o

function.o : function.c
	gcc -c function.c -o function.o

clean :
	rm -rf *.o example
```

Attention, l'indentation des actions doit impérativement être faite avec le caractère tabulation, et non des espaces.

Nous allons maintenant tester notre Makefile :

- taper la commande `make` dans un shell.
- retaper la commande `make` dans un shell : que se passe-t-il ?
- modifier le fichier function.c et retaper la commande `make` dans un shell : que se passe-t-il ?
- taper la commande `make clean` dans un shell.

Makefile évolué

Make est un langage à part entière. Il est possible d'utiliser des variables et certaines variables sont prédéfinies au sein du langage :

`$@`	Le nom de la cible
`$<`	Le nom de la première dépendance
`$^`	La liste des dépendances
`$?`	La liste des dépendances plus récentes que la cible
`$*`	Le nom du fichier sans suffixe

Voici le source d'un Makefile évolué (ce fichier s'appellera Makefile2) :

Listing 2.6 – Makefile2

```
CC=gcc
RM=rm
PROG=example
OBJS= main.o function.o
```

```
all : $(PROG)

main.o : function.h

$(PROG) : $(OBJS)
	@echo linkage :
	$(CC) -o $@ $^

%.o : %.c
	@echo compilation de $< :
	$(CC) -o $@ -c $<

clean :
	$(RM) -rf *.o $(PROG)
```

— taper la commande `make -f Makefile2` dans un shell.

— retaper la commande `make -f Makefile2` dans un shell : que se passe-t-il ?

— modifier le fichier function.h et retaper la commande `make -f Makefile2` dans un shell : que se passe-t-il et pourquoi ?

— taper la commande `make -f Makefile2 clean` dans un shell.

2.3 SCons

SCons[scons] est une sorte de *make* mais écrit en python. Il est donc multi-plateforme et permet de manipuler les cibles et les dépendances en python. Il est bien sur possible de gérer toute sorte de projets (C,C++,LaTeX,Java,.NET,...) et tout est fait pour qu'il y ait le moins de choses à gérer.

Le fichier SConstruct

Créer un fichier SConstruct comme indiqué ci-dessous :

Listing 2.7 – SConstruct

```
Program('example',['main.c','function.c','function.h'])
```

— taper la commande `scons` dans un shell.

— retaper la commande `scons` dans un shell : que se passe-t-il ?

— modifier le fichier function.h et retaper la commande `scons` dans un shell : que se passe-t-il ?

— taper la commande `scons -c` dans un shell. Quel est le rôle de cette dernière commande ?

Le fichier SConstruct peut contenir n'importe quelle instruction Python et de nombreuses fonctions pour la gestion et la compilation du projet sont disponibles.

2.4 Rake

A l'instar de *SCons*, *Rake*[rake] possède toutes les qualités d'un bon gestionnaire de compilation, mais celui-ci est écrit en Ruby. Il est donc, lui aussi multi-plateforme, et va nous permettre d'utiliser toute la puissance de Ruby pour construire notre projet.

Le fichier Rakefile

Créer un fichier Rakefile comme indiqué ci-dessous :

Listing 2.8 – Rakefile

```
require 'rake/clean'

CLEAN.include('*.o','example')

task  :default => ["example"]

SRC = FileList['*.c']
OBJ = SRC.ext('o')

rule '.o' => '.c' do |t|
  sh "cc -c -o #{t.name} #{t.source}"
end

file "example" => OBJ do
  sh "cc -o example #{OBJ}"
end

# File dependencies go here ...
file 'main.o' => ['main.c', 'function.h']
file 'function.o' => ['function.c']
```

— taper la commande `rake` dans un shell.

— retaper la commande `rake` dans un shell : que se passe-t-il ?

— modifier le fichier function.h et retaper la commande `rake` dans un shell : que se passe-t-il ?

— taper la commande `rake clean` dans un shell. Quel est le rôle de cette dernière commande ?

2.5 Ant

Plus ancien que les deux outils précédents, *Ant*[ant] n'en est pas moins aussi efficace. Celui-ci est écrit en Java et est d'ailleurs largement utilisé pour construire les projets écrits en Java. Il est par contre moins efficace pour des projets de tout autre type, même si la compilation d'un projet écrit en C, par exemple, reste possible. L'ensemble des tâches à accomplir pour construire le projet sont décrites dans un fichier XML.

Le projet

Nous allons dans un premier temps construire un petit projet en Java comme suit :

— Créer un répertoire projet-java dans votre répertoire personnel.

— Créer le sous répertoire projet-java/src.

— Les 2 fichiers source : Main.java et Function.java seront à déposer dans le répertoire projet-java/src.

— Déposer le fichier build.xml dans le répertoire projet-java.

Créer le fichier projet-java/src/Function.java comme indiqué ci-dessous :

Listing 2.9 – Function.java

```
public class Function{
	public void function(String name){
		System.out.println("hello " + name + "!");
	}
}
```

Créer le fichier projet-java/src/Main.java comme indiqué ci-dessous :

Listing 2.10 – Main.java

```
public class Main{

	public static void main(String[] args){
		Function function = new Function();
		function.function(args[0]);
	}
}
```

Le fichier build.xml

Créer un fichier build.xml comme indiqué ci-dessous :

Listing 2.11 – build.xml

```
<project name="Project" default="compile" basedir=".">
```

```
    <description> simple example build file</description>
    <property name="src" location="src"/>
    <property name="build" location="build"/>
    <property name="dist"  location="dist"/>

    <target name="init">
        <tstamp/>
        <mkdir dir="${build}"/>
    </target>

    <target name="compile" depends="init" description="compile the source " >
        <javac srcdir="${src}" destdir="${build}"/>
    </target>

    <target name="dist" depends="compile" description="generate the ↩
  distribution" >
        <mkdir dir="${dist}/lib"/>
        <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" ↩
  basedir="${build}">
            <manifest>
                <attribute name="Main-Class" value="Main"/>
            </manifest>
        </jar>
    </target>

    <target name="clean" description="clean up" >
        <delete dir="${build}"/>
        <delete dir="${dist}"/>
    </target>
</project>
```

— taper la commande `ant` dans un shell.

— retaper la commande `ant` dans un shell : que se passe-t-il ?

— modifier le fichier `Function.java` et retapper la commande `ant` dans un shell : que se passe-t-il ?

— taper la commande `ant dist` dans un shell : que se passe-t-il ?

— taper la commande `ant clean` dans un shell. Quel est le rôle de cette dernière commande ?

— taper à nouveau la commande `ant dist` dans un shell : que se passe-t-il ?

Ant est l'outil indispensable à toute personne qui participe à un projet Java ou JavaEE. Il existe de nombreux types de tâches : exécution des tests unitaires en post-compilation,

création de la javadoc, création des war, des ear, déploiement des applications JavaEE sur un serveur d'application,...

Cependant il manque une fonctionnalité importante : la gestion des bibliothèques et de leurs dépendances. *Ivy* est une extension à Ant qui permet de définir la liste des dépendances et les dépôts sur lesquels aller les chercher.

2.6 Conclusion

Ce TP n'est qu'un aperçu des outils dit « de build ». Il en existe de nombreux autres qui ont aussi leurs qualités, même si ils sont moins utilisés. Parmi ceux là il est indispensable que vous connaissiez *Maven*[maven] qui est le plus utilisé dans le monde Java. Son fonctionnement de base est similaire à celui de *Ant* mais il intègre en plus une réelle gestion de dépendances pour les paquets Java, il se comporte très bien avec de nombreux langages. Il intègre aussi une interface web de gestion, envoie des comptes rendu de compilation et de test, etc. Et ceci n'est qu'un tout petit aperçu de ce qu'il est capable de faire. Maven a aussi ses défauts dont le principal est d'être exclusif : difficile d'intégrer un projet qui n'a pas été à la base conçu avec Maven et difficile d'utiliser autre chose une fois qu'on se sert de celui-ci. Une alternative est : *Gradle*[gradle]. C'est l'outil qui rassemble tous les autres. Pour aller plus loin, vous pouvez essayer de refaire la partie Java en utilisant Maven et Gradle.

Enfin, citons dans l'univers Javascript *Grunt*[grunt] et *Gulp*[gulp].

Une fonctionnalité orthogonale mais souvent intégrée au outils de build est la gestion de dépendance. C'est en effet un élément crucial dans la reproductibilité d'une compilation automatique, puisqu'il s'agit de récupérer, installer et configurer automatiquement l'ensemble des bibliothèques tierces nécessaires au projet. De plus en plus d'outils de compilation automatique intègrent donc la gestion de dépendance. D'autres la délèguent à un outils tiers. On peut citer par exemple Bazel ou Bower, PIP en python, Gem en ruby, ou npm dans le monde javascript.

TP 3

Tests Unitaires

Dans ce TP inspiré de [Pil04], on se propose d'expérimenter l'utilisation des tests unitaires en Java. Pour cela, on va réaliser un classe permettant de gérer des chiffres romains.

3.1 Rappel sur les chiffres romains

Dans la numérotation romaine, il y a sept symboles combinés de différentes manières pour composer les nombres :

I = 1
V = 5
X = 10
L = 50
C = 100
D = 500
M = 1 000

La construction de nombres plus complexes suit les règles suivantes :

- Les symboles sont additifs : I vaut 1, II vaut 2 et III vaut 3. VI vaut 6 $(5 + 1)$, VII vaut 7, and VIII vaut 8.
- Les symboles des multiples (I, X, C, et M) peuvent être répétés jusqu'à trois fois. À partir de quatre, il faut soustraire du symbole suivant. Ainsi, 4 est représenté par IV $(5 - 1)$et non pas IIII, 40 s'écrit XL $(50 - 10)$, XLI donne 41, et XLIV donne 44.
- De même, si 8 est VIII, 9 est IX $(10 - 9)$ et pas VIIII, puisque I ne peut être répété que trois fois. Ainsi, 90 est XC et 900 est CM.
- Les symboles des « 5 » ne peuvent pas être répétés. 10 est donc toujours représenté par X (jamais VV), et 100 toujours par C (jamais LL).
- Les chiffres romains s'écrivent du plus grand vers le plus petit et se lisent de gauche à droite. L'ordre compte : DC est 600 $(500 + 100)$, mais CD est 400 $(500 - 100)$. CI est 101, mais IC n'est pas valide (99 s'écrit XCIX, $(100 - 10) + (10 - 1)$)

Les propriétés suivantes devront être respectées :

— il n'existe qu'une seule manière de représenter un nombre en chiffres romains ;

— réciproquement, un nombre romain valide représente un seul nombre décimal ;

— seuls les nombre entre 1 et 3 999 peuvent être représentés (avec ce système).

— Le 0 n'existe pas.

— Les nombres négatifs ne peuvent pas être représentés.

— Seuls les nombres entiers peuvent être représentés.

3.2 Module de conversion

Les algorithmes de conversion sont en fait assez simples et sont donnés ci-dessous.

Algorithme 1 : Conversion vers un chiffre romain
Données : table des symbole contenant les couples $(symbole, nombre)$ connus
Entrées : nombre entier en décimal n
Résultat : chiffre romain
Début
  $resultat \leftarrow$ '' ;
  Pour chaque *couple* $(symbole, nombre)$ *de la table des symboles* **faire**
    Tant que $n \geq nombre$ **faire**
      $resultat \leftarrow resultat + symbole$;
      $n \leftarrow n - nombre$;
  retourner $resultat$

Algorithme 2 : Conversion à partir d'un chiffre romain
Données : table des symbole contenant les couples $(symbole, nombre)$ connus
Entrées : chiffre romain s
Résultat : nombre entier en décimal
Début
  $resultat \leftarrow 0$;
  $index \leftarrow 0$;
  Pour chaque *couple* $(symbole, nombre)$ *de la table des symboles* **faire**
    Tant que $\texttt{SousChaine}(s, index, index + \texttt{Longueur}(symbole)) = symbole$ **faire**
      $resultat \leftarrow resultat + nombre$;
      $index \leftarrow index + \texttt{Longueur}(symbole)$;
  retourner $resultat$

Une base pour le fichier RomanNumber.java vous est données dans l'archive du TP.

Pour simplifier la conversion, les combinaisons « soustractives » ont été ajoutées à la table des symboles. De plus, l'expression régulière de validation des chiffres romains vous est donnée.

3.3 Tests Unitaires et TDD

Principe des tests unitaires

Les tests unitaires ont pour but de tester les fonctions (ou les méthodes) à un niveau élémentaire, avec une approche « boite noire », c'est-à-dire que l'on ne connaît que la signature de la fonction à tester. On teste donc des unités fonctionnelles indépendantes d'un programme plus global.

Il existe des cadres (*framework*) de test unitaire pour la plupart des langages de programmation, souvent inspirés de SUnit, le framework pour Smalltalk. Dans l'architecture de ce type de framework, on distingue les *cas de test* (Test Cases) regroupant les tests proprement dit. De ce point de vue, ils sont à corréler aux cas d'utilisations. Ils sont généralement implémentés par des classes, les tests étant les méthodes de celles-ci. Les cas de tests sont eux-mêmes regroupés dans des suites de tests (tests suites), permettant de tester toute la fonctionnalité ou la classe.

Les méthodes de test ne testent qu'un aspect particulier du cas, correspondant à une spécification particulière, et lève une exception si le résultat n'est pas celui attendu. Cette vérification se fait par l'intermédiaire de méthodes spécifiques (e.g. `assertEquals`, `assertRaises`, etc.). Là aussi, il existe de nombreuses bibliothèques d'assertions, fournissant des fonctions pour faciliter l'écriture des test.

Mise en œuvre

Nous allons suivre les principes du développement guidé par les tests (Test Driven Development), c'est-à-dire écrire les tests unitaires testant ces fonctions *avant* de les écrire. C'est un des principes mis en œuvre dans les méthodes agiles.

Les tests à effectuer seront (au moins) les suivants :

Tests de réussite :

- Donner les bonnes valeurs décimales pour des valeurs romaines connues.
- Donner les bonnes valeurs romaines pour des valeurs décimales connues.

Test d'échec :

- Échouer (lever une exception) pour des valeurs négatives
- Échouer pour des valeurs non entières
- Échouer pour des valeurs en dehors de l'intervalle $[1, 3999]$
- Échouer pour des valeurs avec trop de répétitions de symboles
- Échouer pour des valeurs avec des répétitions de paires
- Échouer pour des valeurs avec des antécédents incorrects (e.g. VX ou XCX)

Test de validité :

- Pour tout entier $n \in [1, 3999]$, $fromRoman(toRoman(n)) = n$
- les chiffres romain sont en capitales
    1. toRoman retourne des capitales

2. fromRoman échoue si on n'a pas que des capitales

Le fichier RomanNumberTest.java donne deux premiers tests pour vous guider.

3.4 Travail

L'idée est donc d'écrire les tests en premier, et d'implémenter les fonctionnalités petit à petit jusqu'à ce que tous les tests passent :

1. implémentez la totalité des tests unitaires (au moins 12) d'après les spécifications précédentes ;
2. implémentez les méthodes toRoman et fromRoman jusqu'à ce que tous les tests passent ;
3. implémentez et testez les méthodes définies dans java.lang.Number [1].

Le modèle fournit pour ce TP utilise l'outil *Gradle*[gradle] présenté dans le TP 2. Le fichier de configuration build.gradle défini les tâches et les dépendances du projet. Ici, les tâches étant celles standard d'un projet Java, utiliser le plugin java est suffisant puisqu'il défini automatiquement les principales tâches. Les dépendances spécifiées sont JUnit[junit] qui est le cadre d'exécution des test en Java, et Hamcrest[hamcrest] qui est une bibliothèque d'assertions.

Ces outils ne sont présentés dans ce TP que de manière très superficielle. Ils offrent en effet des fonctionnalités bien plus riches que ce qu'il est possible d'aborder dans un simple TP. Pensez à regarder leur documentation !

Pour exécuter les tests, il suffit de lancer la tâche test de Gradle (./gradlew test sous linux).

Bien sûr, gradle n'est pas nécessaire, mais c'est plus simple. Il suffit en fait d'avoir les bibliothèques dans le classpath, et de lancer les tests (après compilation) avec java org.junit.runner.JUnitCore roman.RomanNumberTest. Par exemple, si les bibliothèques sont installées sur le système :

```
export CLASSPATH=build\
 :/usr/share/java/hamcrest-all.jar\
 :/usr/share/java/junit4.jar
javac -d build src/*/java/roman/*.java
java org.junit.runner.JUnitCore roman.RomanNumberTest
```

Pour aller plus loin, vous pouvez intégrer un outil de couverture de test, ou *code coverage* (par exemple Jacoco[jacoco]). Ce type d'outils analyse l'exécution des tests et mesure le pourcentage de code effectivement testé.

1. https://docs.oracle.com/javase/8/docs/api/java/lang/Number.html

Bibliographie

[ant] *Apache Ant*. URL : http://ant.apache.org/ (cf. p. 16).

[Bitbucket] *Bitbucket*. URL : https://bitbucket.org/ (cf. p. 10).

[BZR] *Bazaar*. URL : http://bazaar.canonical.com/en/ (cf. p. 1).

[Cha09] S. Chacon. *Pro Git*. 1re éd. Pro. APress, 2009, p. 288. ISBN : 978-1430218333. URL : http://git-scm.com/book (cf. p. 10).

[egit] *EGit*. URL : http://www.eclipse.org/egit/ (cf. p. 3).

[Git] *Git*. URL : http://git-scm.com/ (cf. p. 1).

[gitGUI] *GUI Clients*. URL : http://git-scm.com/downloads/guis (cf. p. 3).

[GitHub] *GitHub*. URL : https://github.com/ (cf. p. 10).

[gradle] *Gradle*. URL : http://www.gradle.org/ (cf. p. 18, 22).

[grunt] *Grunt : The JavaScript Task Runner*. URL : http://gruntjs.com/ (cf. p. 18).

[gulp] *Gulp.js*. URL : https://gulpjs.com/ (cf. p. 18).

[hamcrest] *Hamcrest*. URL : http://hamcrest.org/ (cf. p. 22).

[Hg] *Mercurial SCM*. URL : https://www.mercurial-scm.org/ (cf. p. 1).

[jacoco] *JaCoCo*. URL : http://www.jacoco.org/jacoco/ (cf. p. 22).

[junit] *JUnit*. URL : http://junit.org/junit4/ (cf. p. 22).

[maven] *Apache Maven*. URL : http://maven.apache.org/ (cf. p. 18).

[Mec04] R. Mecklenburg. *Managing Projects with GNU Make*. 3e éd. O'Reilly Media, nov. 2004, p. 304. ISBN : 978-0-596-00610-5. URL : http://oreilly.com/catalog/make3/book/index.csp.

[Pil04] M. Pilgrim. *Dive Into Python*. Apress, 2004, p. 413. ISBN : 1590593561 (cf. p. 19).

[rake] *Ruby Rake*. URL : http://rubyforge.org/projects/rake/ (cf. p. 15).

[scons] *The SCons Fundation*. URL : http://www.scons.org/ (cf. p. 14).

[SVN] *Subversion*. URL : http://subversion.apache.org/ (cf. p. 1).

[tortoisegit] *TortoiseGit*. URL : https://tortoisegit.org/ (cf. p. 3).