***Objectif* :** Se familiariser avec la notion de SCM (« source code management ») avec l'outil Git

1. Environnement et quelques commandes élémentaires

Pour prendre en main l'outil « git », nous allons privilégier la ligne de commande afin de bien comprendre le fonctionnement de l'outil, en s'affranchissant ainsi de la variété des outils graphiques (simples surcouches aux commandes). Dans la pratique, il est courant d'utiliser une version graphique souvent intégrée au logiciel de développement (e.g. eclipse, visual studio, …). Dans notre cas, on utilisera le « git bash » disponible sur votre machine (répertoire AGI/git) : il s'agit d'un interpréteur unix (supportant donc les commandes unix élémentaires) permettant d'invoquer les commandes git.

Afin de créer et de modifier des fichiers (hors utilisation de git), on pourra utiliser les commandes unix:

- « touch file.txt » pour créer un fichier,
- « echo "ligne" >> file.txt » pour écrire une nouvelle ligne en fin de fichier,
- « mkdir rep » pour créer un répertoire,
- « cd rep » pour changer de répertoire,
- « pwd » pour afficher le chemin courant,
- « ls -al » pour consulter le contenu d'un répertoire,
- « cat » pour afficher le contenu d'un fichier.

La documentation officielle est disponible en local (git help <nom_option>) et en ligne (http://git-scm.com/docs)

Création d'un dépôt	
git init git init --bare	Création d'un dépôt git vide dans le répertoire courant, avec ou sans (option « bare ») répertoire de travail (i.e. fichiers de travail).
git config --global user.name "nom" git config --global user.email "nom@email.com" git config --local user.name "nom" git config --local user.email "nom@email.com"	Configuration d'un utilisateur (nom+email). System : pour tous les dépôts d'une machine. Global : pour tous les dépôts de l'utilisateur courant (e.g. utilisateur unix). Local : uniquement pour le dépôt courant.
git clone path_to_remote git clone --bare path_to_remote	Création d'un dépôt (dans répertoire courant) avec ou sans (option « bare ») répertoire de travail, en clonant un dépôt existant (path_to_remote).

Commande d'édition du dépôt local	
git add fichier.txt	Ajoute le « fichier.txt » sous suivi de version, ou indexe le fichier modifié pour validation.
git commit -m "message" fichier.txt	Valider le « fichier.txt » (récemment ajouté ou modifié)
git tag <version>	Etiquette la dernière validation (typiquement version d'un livrable)
git checkout <id>	Rétablir le fichier du répertoire de travail dans une version antérieure (numéro de « commit » ou tag) ou d'une branche
git branch <nom>	Créer une branche « nom ».
git merge <nom_branche> -m "msg"	Intègre les dernières modifications d'une branche (« nom_branche ») à la branche courante (souvent « master »). Ceci se traduira par un nouveau « commit » avec le message « msg »

Observation	
git log	Affiche l'historique des validations (« commits »)
git status	Affiche l'état des fichiers (e.g. modifiés et non « commited »)
git tag	Affiche les versions étiquetées correspondant souvenent à des livrables (« tags »)
git gui	Démarre une interface graphique
git branch	Affiche les branches (l'option « -a » affiche toutes les branches, même celles associées à un dépôt distant)
git diff	Affiche la différence entre différents états d'un fichier (par défaut : état courant et dernier état validé).

Travail collaboratif	
git remote add <nom> <chemin_vers_depot_distant>	Ajoute une dépôt distant qui sera associé au nom « nom » dans le dépôt courant (local). Lors d'un clonage, le dépôt distant cloné est automatiquement ajouté comme dépôt distant (par défaut, avec le nom « origin »).
git remote	Affiche les dépôts distants ajoutés au dépôt courant.

git push <remote> <local_branch>:<remote_branch>	Pousse les modifications locales (supposée validées i.e. « commited ») sur un dépôt distant (« remote »). On également préciser la branche locale considérée et la branche distant « cible ». « git push » sans paramètre utilisera les informations par défaut : « origin » pour le dépôt distant, « master » pour les branches.
git fetch nom	Recupère les modifications distantes validées de la branche « master » du dépôt distant « nom » dans une branche locale (« nom/master »). Les modifications sont intégrées par « git merge nom/master ».
git pull <remote> <remote_branch>	Equivalent à « fetch » puis « merge » : récupère les modifications distantes du dépôt distant « nom » et effectue un « merge » avec la branche courante du dépôt local.

2. Historisation

L'objectif est de faire évoluer le contenu d'un fichier, en gardant une trace des différents états intermédiaires, ainsi qu'en étiquettant des versions importantes du fichier.

Le scénario à réaliser est le suivant :

1- Créer un répertoire « project »..

2- Initialiser dans le répertoire projet un dépôt « git » avec la commande « git init » (exécutée depuis le répertoire « project »).

3- Configurer ce dépôt avec votre nom et votre email, ce qui permettra d'identifier l'auteur des modifications :
git config --global user.name "nom"
git config --global user.email "nom@email.com"

4- Créer un fichier « file.txt » avec une ligne « ligne01 » et ajouter le fichier au contexte (« staged area» : près pour validation - « commit ») avec l'instruction « git add file.txt »). Consulter l'état du dépôt avec « git status ».

→ Affichage console
#nouveau : file.txt

5- Ajouter la seconde ligne suivante au fichier : « ligne02 » ; puis « commiter » cette nouvelle version. Afficher l'historique des modifications avec la commande « git log »

→ Affichage console:
[master (root-commit) 527129d] first commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 file.txt
commit 527129d538992d5d625a74a45aa175237fdfc5c4
Author: nom <nom@email.com>
Date: Tue Feb 10 23:17:23 2015 +0100

first commit

6- On considère cette version correspond à un livrage du code que l'on va nommer « v0.1 ». Intégrer cette notion en utilisant la commande « git tag v0.1 ».

7-Ajouter une nouvelle ligne « ligne03 ». Afficher les différences (« git diff ») avec la précédente version commitée.

→ *Affichage console :*
diff --git a/file.txt b/file.txt
index 4337b7a..ab09d39 100644
--- a/file.txt
+++ b/file.txt
@@ -1 +1,2 @@
ligne01
ligne02
+ligne03
[master 9979bbd] second commit

8- « Commiter » cette modification.

9- Revenir à la version marquée 0.1 (« git checkout v0.1») et afficher le contenu du fichier (« cat ») :

→ *Affichage console :*
ligne01
ligne02

10-Revenir à la dernière (« checkout » avec « master » ou « HEAD » comme cible), et afficher le contenu du fichier (« cat ») :

→ *Affichage console :*
ligne01
ligne02
ligne03

11-Ajouter et commiter une nouvelle ligne « ligne04 ». Marquer cette version comme « v0.2 ». Afficher ensuite les différentes versions : « git tag ».

→ *Affichage console :*
v0.1
v0.2

12-Consulter graphiquement le dépôt : « git gui »

13-Consulter graphiquement le dépôt avec eclipse : créer un projet (« new project ») à l'endroit de votre répertoire ; consulter l'historique de votre fichier (« bouton souris droit », « Team », « Show in History »).

3. Développement avec « branches »

A partir d'une version donnée du code, on souhaite pouvoir poursuivre le développement principal, tout s'autorisant de partir sur des développements annexes sur une période plus courte. Il peut typiquement s'agir d'une correction d'un bug majeur qui peut nécessiter plusieurs étapes et affecter plusieurs fichiers.

On considère le « file.txt » suivant :
igne01
igne02
ligne03

On constate qu'il y a une erreur (« bug ») sur les deux premières lignes : il manque la lettre « l » devant « igne01 ». Pour corriger cette erreur, nous allons travailler sur une branche secondaire afin de ne pas bloquer les développements sur la branche principale.

1-Créer le dépôt, configurer-le avec votre nom et votre email, et intégrer le fichier « fichier.txt » avec les 3 lignes indiquées.

2- Créer une branche « bug », et afficher les branches disponibles sur le dépôt (« branch ») ainsi que votre état (« status »).
→ *branches*
bug
** master*
→ *status*
 # Sur la branche master
rien à valider, la copie de travail est propre

3- Placer vous sur la branche « bug » (« checkout »), et reafficher les branches disponibles ainsi que l'état.
→ *branches*
** bug*
master
→ *status*
 # Sur la branche bug
rien à valider, la copie de travail est propre

4- Vous êtes toujours sur la branche « bug » : éditer le fichier afin d'ajouter le « l » manquant sur la première ligne seulement. Commiter cette modification avec le message « file.txt : ligne01 fixed ».

→ *sortie console*
[bug b22f5fa] file.txt: ligne01 fixed
1 file changed, 1 insertion(+), 1 deletion(-)

5- On suppose que le développement se poursuit sur la branche principale (master) : placer vous sur cette branche (« checkout »). Consulter l'état du fichier (« cat »): la correction apportée ne doit pas apparaître car celle-ci a été faite sur la branche. Ajouter une nouvelle ligne au fichier (ligne04) et commiter cette ajout.
→ *sortie console*
Basculement sur la branche 'master'
[master da3c76c] file.txt: ligne04 added
1 file changed, 1 insertion(+)
→ *contenu du fichier*
igne1
igne2
ligne3
ligne4

6- Revenez sur la branche « bug », corriger la ligne02 du fichier.txt (« igne02 » → « ligne02 ») et commiter la modification. Afficher l'état du fichier.
→ *sortie console*
Basculement sur la branche 'bug'
[bug 0bb780f] file.txt: ligne02 fixed

1 file changed, 1 insertion(+), 1 deletion(-)
→ *contenu du fichier*
ligne1
ligne2
ligne3

7-Nous allons intégrer les corrections apportées sur la branche « bug » à la branche principale : on commence par se positionner sur la branche « master » (« checkout ») - on affichera le contenu du fichier - , puis on intègre les modifications de la branche « bug » : git merge bug -m "fusion". Afficher l'état du fichier après modification : les deux premières sont corrigées, et la ligne04 doit apparaître (ajoutée en parallèle du correctif sur la branche « bug »).
→ *sortie console*
Basculement sur la branche 'master'
→ *contenu du fichier avant « merge »*
igne1
igne2
ligne3
ligne4
→ *sortie console*
fusion automatique de file.txt
Merge made by the 'recursive' strategy.
file.txt | 4 ++--
1 file changed, 2 insertions(+), 2 deletions(-)
→ *contenu du fichier apres « merge »*
ligne1
ligne2
ligne3
ligne4

8- Afficher graphiquement toutes les évolutions : « git gui » → afficher l'historique du dépôt.

4. Développement collaboratif

Pour se familiariser avec le développement collaboratif, nous allons considérer un projet avec trois dépôts (dans trois répertoires distincts): deux dépôts locaux (« repo1 » et « repo2 » associés aux utilisateurs « nom1 » et « nom2 ») et un dépôt distance « remote ». En fin d'exercice, les dépôts seront associés à des répertoires :
repertoire_courant :
- → **répertoire « repo1 »**
  - → fichiers + « .git/ »
- → **répertoire « repo2 »**
  - → fichiers + « .git/ »
- → **répertoire « remote »**
  - → fichiers + « .git/ »

1-Créer un répertoire « repo1 », aller dans ce répertoire (« cd repo1 »), puis initialisation un dépôt git, et configurez-le avec le nom « nom1 » et le email « nom1@email.com » (avec l'option local). Créer un fichier file.txt avec une seule ligne « ligne01 » : ajouter puis commiter le fichier. Afficher les logs (« git log »)

→ sortie console :
[master (root-commit) a809cc3] first commit
1 file changed, 1 insertion(+)
create mode 100644 file.txt
→ *log :*
commit a809cc3fea2e5f610056124f1a16c3ab833eb6d6
Author: nom1 <nom1@email.com>
Date: Thu Feb 12 11:22:20 2015 +0100

first commit

2-Remonter dans l'arborescence. Créer un réperoire « remote » puis aller dans ce répertoire. Nous allons initialiser le dépôt « remote » par « clonage » du « repo1 » : git clone -- bare path_to_repo1 . (« . » désigne le répertoire courant). Afficher ensuite le « log »
→ *sortie console :*
Clonage dans le dépôt nu '.'
fait.
→ *log :*
commit 01cd94657d0868352bfb6489626e3055225c9a7d
Author: nom1 <nom1@email.com>
Date: Thu Feb 12 11:35:56 2015 +0100

first commit

3-Remonter dans l'arborescence. Créer un répertoire « repo2 » puis aller dans ce répertoire. Nous allons initialiser le dépôt « repo2 » par « clonage » du dépôt remote : « git clone path_to_remote . » (« . » désigne le répertoire courant). Configurez ensuite ce dépôt avec le nom « nom2 » et le email « nom2@email.com » (avec l'option local). Afficher ensuite le « log ».

→ *sortie console :*
Clonage dans '.'
fait.
→ *log :*
commit e1e856ea6f0bf65b1950329e7a78245a92dd26d6
Author: nom1 <nom1@email.com>
Date: Thu Feb 12 12:14:39 2015 +0100

first commit

4-Depuis « repo2 » : afficher le contenu du fichier. Ajouter la ligne ensuite une ligne « ligne02 » en fin de fichier : ajouter puis commiter cette modification. Afficher le log (on voit les deux modifications : celle faite par « nom1 » puis celle que vous venez de faire (« nom2 »).

→ *sortie console :*
[master c2e33a4] file.txt: second ligne
1 file changed, 1 insertion(+)
→ *log :*
commit c2e33a41e29292c6450a7e6e8fe4566a62a29f59
Author: nom2 <nom2@email.com>
Date: Thu Feb 12 12:18:16 2015 +0100

file.txt: second ligne

commit 3a18c8de1039a9099d1613948e49ec09adc18354
Author: nom1 <nom1@email.com>
Date: Thu Feb 12 12:18:16 2015 +0100

first commit

5-Mise à jour du dépôt distant (« remote ») depuis le dépôt courant « repo2 » : restez dans le dépôt « repo2 », et « pousser » (« push » sans option) la modification précédente vers le dépôt « remote ».
→ sortie console :
Counting objects: 5, done.
Writing objects: 100% (3/3), 254 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To path_to/remote
bf251e5..df8c86f master -> master

6-Vérifiez que la mise à jour a bien été intégrée au dépôt « remote » : placez-vous dans le répertoire « remote », et consulter le « log ». On constate que les deux modifications sont indiquée : celle effectuée par le « nom1 » (« repo1 ») et la dernière effectuée par le « nom2 » (« repo2 »). On constate par ailleurs que, dans le répertoire « remote », il n'y a pas de fichier « file.txt » : l'historique des états du fichiers est pourtant stocké (répertoires propres à « git »), mais ce dépôt est « bare », c'est à dire sans répertoire de travail (i.e. pas de copie « éditable » des fichiers car il ne s'agit pas d'un dépôt de travail – i.e. pas de développeur). Ceci permet de pousser vers ce dépôt sans risquer d'incohérences avec des fichiers qui seraient localement modifiés.

→ log :
commit f092e137f8e18837eace1b5e519ade83faf750b6
Author: nom2 <nom2@email.com>
Date: Thu Feb 12 14:15:46 2015 +0100

file.txt: second ligne

commit 25d7ffb3eb2edab62a392f62470f4cfd143f4355
Author: nom1 <nom1@email.com>
Date: Thu Feb 12 14:15:46 2015 +0100

first commit

7-On souhaite récupérer en « local » dans « repo1 » les modifications apportées au « remote » depuis « repo2 ». Placez-vous dans le répertoire « repo1 ». Pour commencer, il faut ajouter « remote » comme dépôt distant à « repo1 » : contrairement au dépôt « repo2 », « repo1 » n'a pas été initialisé par clonage (c'est « remote » qui a cloné « repo1 » et non l'inverse). Pour cela, utilisez la commande « git remote add distant chemin_vers_remote », et vérifier qu'il est bien enregistré comme dépôt distant : « git remote » :
→ *retour console de « git remote »*
distant

8-Toujours depuis « repo1 », vous pouvez récupérer en local des dernières modifications disponibles sur le « remote » : « git fetch distant ». Cela crée une branche locale : affichez la liste des branches (« git branch -a »), puis positionnez-vous sur la branche nouvellement créée (« checkout ») pour consulter l'état du fichier.

→ sortie « fetch »
remote: Counting objects: 5, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From path_to/remote

* [new branch] master -> distant/master

→ branches (« git branch -a » pour afficher toutes les branches, compris celles associées à un « remote »)
* master
  remotes/distant/master
→ positionnement sur branche et consultation contenu fichier :
Note: checking out 'distant/master'.

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using -b with the checkout command again. Example:

git checkout -b new_branch_name

HEAD est maintenant sur e42fdbc... file.txt: second ligne
ligne01
ligne02

9-Revenez sur la branche principale : « checkout master ». Consulter l'état du fichier. Intégrez ensuite les modifications importées sur la branche (« merge distant/master»). Consulter à nouveau l'état du fichier.
→ *fichier avant merge*
ligne01
→ *merge*
Updating 01cd7ae..216e34b
Fast-forward
file.txt | 1 +
1 file changed, 1 insertion(+)
→ *fichier après merge*
ligne01
ligne02

10-Ajouter une nouvelle ligne puis pousser sur le dépôt distant (« push distant master:master »).

→ *sortie console du push*
Counting objects: 5, done.
Writing objects: 100% (3/3), 253 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To path_to/remote
b6644c1..5f52c52 master -> master
La branche master est paramétrée pour suivre la branche distante master depuis distant.

11-La mise à jour de « repo1 » peut paraître fastidieuse car « repo1 » n'est pas le clone du « remote » (c'est l'inverse). Si « repo1 » était le clone du « remote », un certain nombre de choses auraient été configurées automatiquement, simplifiant considérablement les commandes. Par exemple, il a fallu explicitement ajouter un « remote » à « repo1 » : en cas de clone du dépôt distant (e.g. cas de « repo2 »), le dépôt est automatiquement configuré (nom : « origin »), et la commande « fetch » utilise ce dépôt distant par défaut (inutile de préciser « fetch distant »). Pour illustrer cela, retournons dans le « repo2 » qui est un clone du « remote », avec certains éléments automatiquement configurés :

- **Placez-vous dans « repo2 »**
- **Récupérer les modifications apportées par « repo1 » sur « remote » : « git pull » (équivalent de « fetch » + « merge »)**
- **Vérifier que le fichier est mis à jour.**
- **Modifier ce fichier (ajout « ligne04 » en fin de fichier) et commiter la modification**
- **Pousser sur le « remote » : « git push ».**

→ sortie console « pull »
remote: Counting objects: 5, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From path_to/remote
  6884765..057ea2c master -> origin/master
Updating 6884765..057ea2c
Fast-forward
file.txt | 1 +
1 file changed, 1 insertion(+)

→ affichage contenu fichier
ligne01
ligne02
ligne03

→ sortie console de la modificaton et commit du fichier
[master d3a3274] file.txt: ligne04
1 file changed, 1 insertion(+)

→ sortie console du « push »

```
Counting objects: 5, done.
Writing objects: 100% (3/3), 258 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To path_to/remote
   057ea2c..d3a3274  master -> master
```

12-De retour dans le « repo1 » (cd repo1), vous pouvez récupérer ces modifications : « git pull distant master »

→ sortie console

```
From path_to/remote
 * branch            master     -> FETCH_HEAD
Updating 7128b69..09cd5ae
Fast-forward
 file.txt | 1 +
 1 file changed, 1 insertion(+)
```

BRAVO, VOUS AVEZ TERMINÉ !!!! Mais vous pouvez recommencer avec eclipse par exemple...

5. Avec Eclipse

1-Créer un répertoire vide « monProjet » et initialiser le dépôt (git init) en ligne de commande.

2-Démarrer eclipse et créer un nouveau projet eclipse sur le répertoire « monProjet » : afficher ensuite l'historique « git » dans eclipse (bouton droit : menu « Team » puis « Show in history »).

3-Reprendre ensuite, depuis eclipse, les scénarii « historisation » et « branches ».