IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

TP – Utilisation de Git avec Eclipse

INFO – MAPD

Objectifs

L'objectif de cette séance est de découvrir rapidement l'environnement de travail collaboratif que vous utiliserez pour le fil rouge. On suppose que Java et Eclipse sont déjà installés sur votre machine. À l'issue de cette séance vous serez en capacité technique de développer votre projet en équipe, plus concrètement vous serez capables de :

- Créer un espace projet avec un dépôt Git (sur la plateforme GitLab de l'école)
- Utiliser les commandes de base de Git pour votre projet

Git ?

L'écriture de logiciels nécessite généralement la coopération de plusieurs développeurs. Pour faciliter cette coopération, on peut (il vaut mieux) utiliser un outil de gestion de version (« *Version Control System* » (VCS). Il existe deux types de VCS :

- centralisés : Subversion (SVN), CVS...
- distribués : Git, Mercurial, Darcs...

Ils permettent tous le stockage de fichiers, le suivi des modifications de ces fichiers et leur partage.

Dans le cas de Git, il s'agit d'un VCS distribué. Comme le montre la figure 1, chaque développeur synchronise sa copie de travail avec un dépôt local que l'on synchronise à son tour avec un ou plusieurs autres dépôts distants. Dans le cadre de cette initiation, nous utiliserons la configuration la plus simple : un seul dépôt de référence pour tous les membres de l'équipe.

Pour bien comprendre le fonctionnement de Git, vous devez connaître le vocabulaire qui y est associé...(voir figure 2)

- Répertoire de travail : contient les fichiers sur lesquels on travaille en local (ce que vous avez l'habitude de « regarder »...)
- Espace Git : tous les fichiers et répertoires nécessaires à Git pour gérer les fichiers et leurs versions. Tous ces éléments se trouvent dans le répertoire `.git` de votre répertoire de travail. Dans cet espace vous avez trois types d'informations :

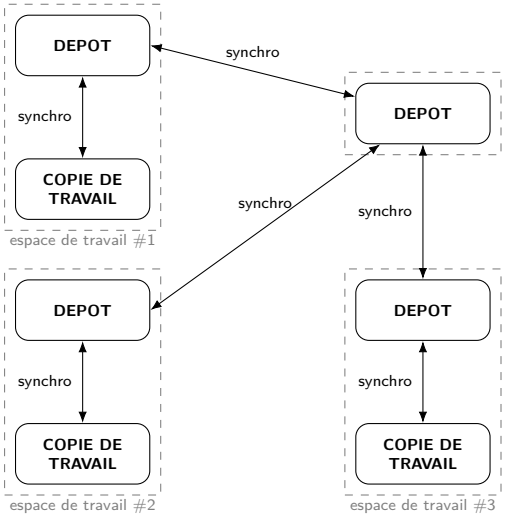

FIGURE 1 – Principe d'utilisation de Git

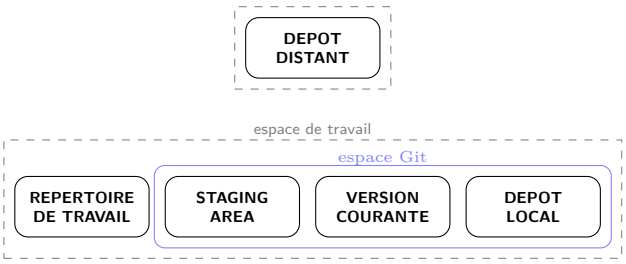

FIGURE 2 – Un peu de vocabulaire associé à Git

— Dépôt local/distant (*local/remote repository*) : il s'agit des dépôts qui contiennent les changements locaux/distants qui ont été faits ;
— Version courante : il s'agit de la « version de référence », c.à.d. la version qui a été sélectionnée en local et sur laquelle on construit quelque chose. Dans le vocabulaire Git on parle de *HEAD* ;
— *Staging area* : c'est ici que Git conserve la liste des modifications faites en local à la version courante et qui n'ont pas encore été intégrées au dépôt distant. Ce niveau local de gestion de versions permet par exemple au développeur de revenir à un état atérieur s'il le souhaite, ce qui est une sécurité appréciable [1]. Dans le vocabulaire Git on parle aussi de *index*.

Principes de fonctionnement de Git

Chaque développeur désirant travailler sur un projet, doit d'abord demander à Git de :

— créer le dépôt local qui contiendra les changements locaux (si ce dépôt n'existe pas au préalable) ;
— cloner le dépôt distant qui contient le sources dans son dépôt local ;
— créer le dépôt distant, si celui-ci n'existe pas au préalable.

1. Vosu préférez travailler sans filet ?

Lorsque vous démarrez votre participation à un projet Git, vous serez dans un des deux cas de figure suivants :

1. le dépôt distant existe et vous n'avez pas de fichiers dans votre espace de travail local que vous voulez integrer au dépôt (cas courant quand vous voulez commencer à participer à un projet de développement existant) : vous devez alors cloner le dépôt distant sur votre machine ;
2. le dépôt distant existe et vous voulez y intégrer des fichiers que vous avez dans votre espace de travail : vous devez alors créer le dépôt local qui contiendra vos fichiers et l'associer au dépôt distant ;
3. le dépôt distant n'existe pas et vous n'avez pas de fichiers dans votre espace de travail local que vous voulez versionner : vous devez créer d'abord le dépôt distant pour vous trouvez dans le premier cas ;
4. le dépôt distant n'existe pas mais vous avez des fichiers dans votre espace de travail local que vous voulez versionner : vous devez créer le dépôt distant et le dépôt local puis faire le lien entre les deux.

Dans cette activité d'initiation, vous mettrez en pratique les cas 2. (dans la partie sur le développement avec un seul développeur) et 1. (dans la partie sur le développement à plusieurs).

Une fois ces opérations réalisées (on a un dépôt distant et un dépôt local qui sont liés), le développeur peut apporter des modifications aux fichiers dans son espace de travail avec ses outils de développement habituels et demander à Git de prendre en compte ces modifications comme constituant une nouvelle version de son dépôt local. Enfin, le développeur peut propager les modifications de son dépôt local dans le dépôt distant pour les partager avec les autres développeurs.

Selon ce modèle, il est possible que plusieurs développeurs fassent en parallèle des modifications chacun de leur côté sur leur propre copie locale. Le moment venu, il faudra donc gérer correctement les propagations des modifications vers le dépôt distant commun : cette gestion globale des versions est le rôle principal de Git. Chaque développeur travaille donc directement sur une copie du projet, et Git apporte les services de synchronisation globale.

Git permet ainsi de travailler à plusieurs simultanément sur un même fichier [2]. En effet, Git sait détecter les collisions des modifications et prévient les développeurs des incohérences potentielles. Git n'est cependant pas assez intelligent pour corriger automatiquement ces incohérences : les développeurs devront décider du contenu à conserver pour la nouvelle version du fichier.

Ce que Git peut faire pour une équipe, il peut aussi le faire pour un seul développeur et on pourra profiter de cet outil pour gérer ses propres versions de fichiers. Pour cette séance d'initiation, nous n'aborderons que les commandes de base de Git. Tout d'abord, nous verrons comment une personne seule peut tirer bénéfice de l'utilisation de Git (aspects de versionnage). Le développement collaboratif sera ensuite abordé, en utilisant un dépôt distant géré par la plateforme GitLab de l'école. Finalement, nous présentons des réponses à des questions fréquentes concernant l'usage de Git dans le cadre du projet fil rouge.

Découverte de GitLab

Parmi plein d'autres services, IMT-Atlantique met à votre disposition une plateforme d'hébergement : `gitlab.imt-atlantique.fr`.

2. En termes techniques : il n'y a pas d'*exclusion mutuelle* d'accès.

GitLab est une plateforme Web de gestion de projets très complète qui peut être utilisée pour gérer un projet de développement de logiciel. Du point de vue gestion de versions de code, elle permet notamment l'utilisation de Git. Plutôt que d'utiliser une plateforme d'hébergement avec laquelle votre confidentialité n'est assurément pas garantie, utilisez donc une plateforme de l'école !

Dans le cadre d'un projet de développement, plusieurs membres sont amenés à collaborer (p.ex. chef de projet, encadrant(s), développeurs). Ce TP s'attache plus particulièrement à vous présenter le rôle de membre développeur, et parfois celui d'administrateur, d'un projet hébergé sur une plateforme GitLab. Sur un même projet, plusieurs membres peuvent être administrateurs et un administrateur a naturellement plus de fonctionnalités accessibles qu'un développeur.

La plateforme GitLab est accessible à l'URL `gitlab.imt-atlantique.fr`. Cet accès nécessite une authentification (login/mdp école).

Créer un projet sur GitLab

Cette partie est plus particulièrement destinée au membre administrateur d'un projet et a pour but de vous permettre de créer un nouveau projet sur la plateforme GitLab. D'autres fonctionnalités non présentées dans ce TP sont également disponibles. Le guide utilisateur de GitLab (cf. `https://docs.gitlab.com/`) les présente de manière exhaustive.

Soumettre un projet

Cette étape vous permet de créer un projet et ainsi d'en devenir membre administrateur. Pour le TP, une fois connecté sur la plateforme, pour créer un projet :

- Cliquez sur le bouton *New project* (en haut à droite) puis *Create blank project*
- Remplissez le champ nom (par ex. `mapd-d-2022-test`), **décochez la case** « *Initialize repository with a README* »pour que le projet soit vide et laissez les autres paramètres avec la valeur par défaut

Une présentation détaillée des différentes options du projet est donnée dans l'aide GitLab `gitlab.imt-atlantique.fr/help/user/project/repository/index.md` dans la rubrique *Create a project*.

Ajouter des membres à son projet

Un membre administrateur de projet peut ajouter/inviter d'autres membres (p.ex. encadrants, développeurs) dans son projet. Ces membres pourront modifier (selon les droits qui leur sont accordés) les fichiers du projet et profiter des divers outils collaboratifs.

En tant que membre administrateur de votre projet :

- Cliquez sur *Menu* (en haut à gauche) puis *Projects > Your projects* et sélectionnez le projet que vous venez de créer
- Cliquez sur *Project information* (du menu à gauche) puis *Members*

Pour ajouter des membres à votre projet :

- Cliquez sur le bouton *Invite members* en haut à droite
- Tapez l'adresse mail du membre à ajouter (votre encadrant)
- Associez-lui le rôle *Owner*[3]. Tous les détails sur les droits de chaque rôle se trouve `docs.`

3. Un rôle permet d'attribuer un ensemble de droits (p.ex. lire, écrire, administrer) à chacun des membres

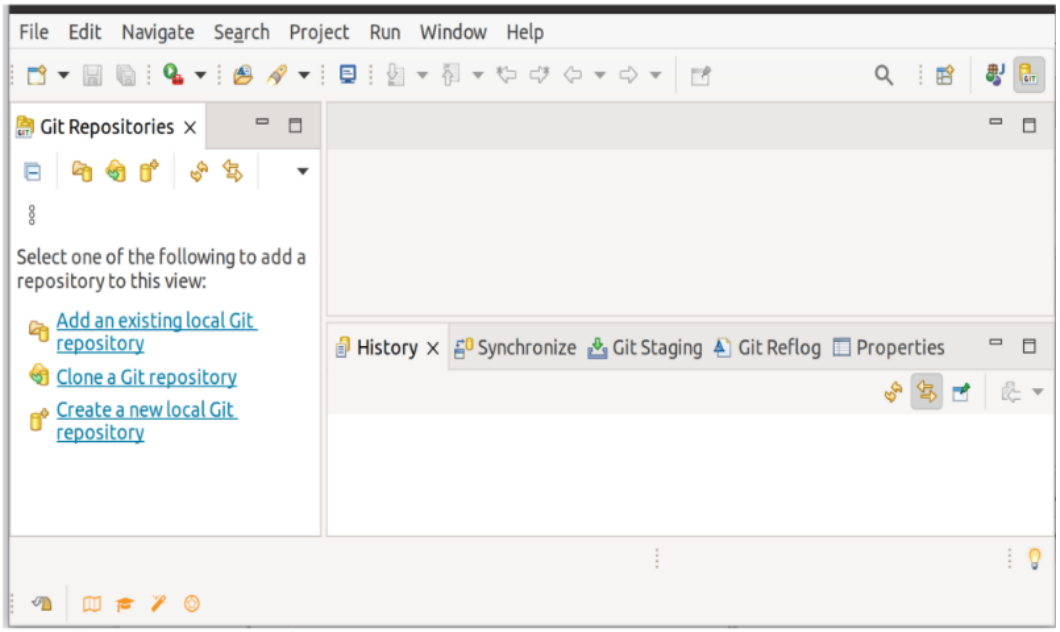

FIGURE 3 – Perspective Git d'Eclipse

`gitlab.com/ee/user/permissions.html#project-members-permissions`

— Cliquez sur *Invite*

Remarque : les membres administrateurs (rôle *Owner*) ont la possibilité par la suite de modifier ces droits.

Découverte de Git avec Eclipse

Git a été créé par Linus Torvald pour gérer l'évolution du système Linux : pas étonnant que Git soit plus particulièrement adapté à une utilisation sous ce système d'exploitation ! Heureusement pour ceux d'entre vous qui utilisent un autre système d'exploitation, les versions récentes d'Eclipse permettent de travailler directement avec Git, sans passer par les commandes en ligne.

Utilisation d'un « bac à sable »

Pour votre initiation, vous allez utiliser un projet « bidon » juste pour expérimenter les manipulations de Git. De la même manière, pour ne pas polluer votre espace de travail Eclipse habituel, vous allez utiliser un espace de travail spécifique.

Lancez donc Eclipse en choisissant un nouvel espace de travail que vous nommerez `UN`[4].

Dans Eclipse, le menu *Windows* > *Perspective* > *Open perspective* > *Other...* permet d'ouvrir la perspective Git (voir figure 3). On voit ainsi qu'il est directement possible de créer un nouveau dépôt Git local, d'en utiliser un déjà existant et de cloner un dépôt distant existant...

d'un projet par rapport aux différents outils offerts par la plateforme (p.ex. demande, dépôt du code et de la documentation). Plusieurs rôles sont prédéfinis : *Owner* et *Maintainer* appelés membres administrateurs dans ce sujet, *Developer*, *Reporter* et *Guest*.

4. Vous comprendrez le choix de ce nom un peu plus tard.

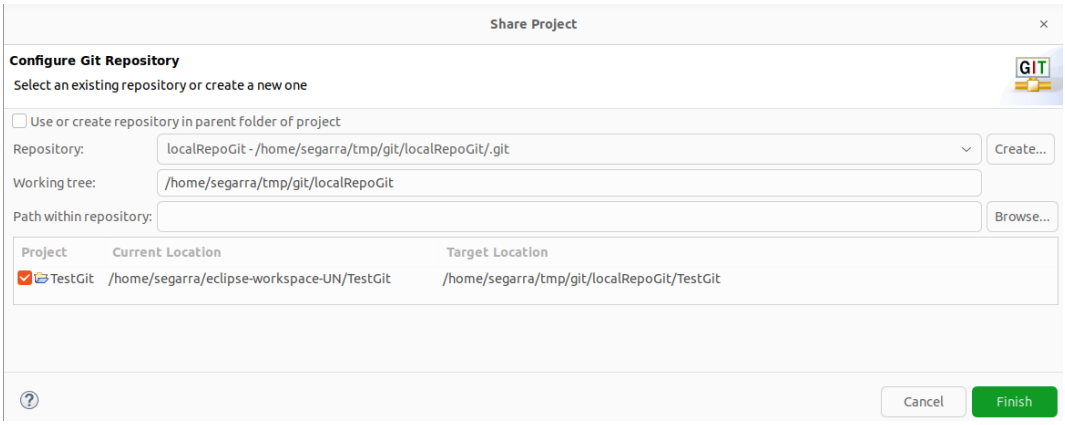

FIGURE 4 – Création d'un dépôt Git (local)

Vous pouvez changer rapidement de perspective via les petites icônes tout en haut à droite : revenez directement à la perspective Java.

Développement avec un seul contributeur

Ici, vous allez pratiquer le cas 2. identifié dans « Principes de fonctionnement de Git ». Vous avez un dépôt distant créé sur GitLab (qui ne contient rien pour l'instant) et vous souhaitez commencer à travailler sur un projet Java qui va revolutionner le monde des sports ...

Vous allez d'abord créer un nouveau projet Java, (appelez-le `TestGit` par exemple) comme vous avez maintenant l'habitude de le faire avec Eclipse. Créez-y une classe `Main` minimale dont le `main()` qui vous affiche un message de courtoisie (« Hi Champion ! »)

Création du dépôt local. Il faut d'abord gérer ce projet avec Git en local sur votre propre machine (créer le dépôt Git local). Pour cela, sélectionnez votre projet dans le *Package Explorer* cliquez sur le bouton droit de la souris et demandez *Team* > *Share project* (voir figure 4) pour créer un nouveau dépôt Git local sur votre machine. Cliquez sur le bouton *Create...* et choisissez comme emplacement un nouveau dossier vide. **Attention,** vous devez donner la valeur `main` au paramètre *Default branch name* dans la fenêtre qui s'ouvre.

Opérations sur l'espace de travail local. Une fois que vous avez créé le dépôt local, dans le *Package Explorer*, des minis points d'interrogation sont venus décorer les icônes des différents constituants du projet : ceci vous indique qu'il y a des fichiers dans votre répertoire travail qui ne sont pas connus dans le dépôt local Git auquel vous venez de lier votre projet. Il va vous falloir indiquer lesquels doivent être synchronisés avec ce dépôt : vous pouvez en ajouter (*add*), en retirer (*rm*), les déplacer (*mv*)... De fait, ces commandes ajoutent des opérations dans la *« staged area »* de Git (programment des actions sur le dépôt local). Ces opérations ne seront effectivement réalisées que lorsqu'elles sont validées : c'est ce qu'on appellera « faire un *commit* ».

Sélectionnez votre fichier `Main.java` et, via le menu contextuel (clic sur le bouton droit de la souris), ajoutez-le à la liste des fichiers suivis (*Team* > *Add to index*). Voyez l'impact sur son icône (et éventuellement sur les dossiers le contenant) qui indique maintenant que l'ajout de ce fichier est planifié.

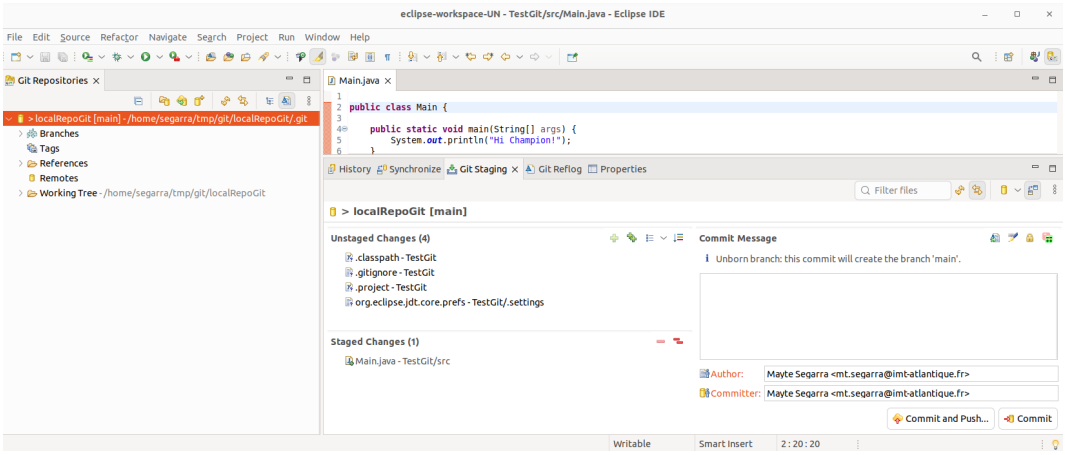

FIGURE 5 – Perspective Git : l'ajout du fichier `Main.java` est planifié

Vous pouvez aussi le constater si vous revenez dans la perspective Git (cf. « *Staged changes* » dans la figure 5), et les autres fichiers (cachés) de votre projet sont signalés comme non référencés (« *unstaged* ») dans le dépôt local. En utilisant le menu contextuel, vous pouvez facilement passer un fichier de « *staged* » à « *unstaged* », et inversement.

Chaque *commit* produit une nouvelle version locale du projet, qui doit être associée à un commentaire explicitant à quoi correspond cette nouvelle version : c'est la base de la documentation de l'historique des versions. Pour pouvoir faire le *commit* ici, il faut donc rajouter un commentaire (*Commit Message*), par exemple « *Version de départ* ». Faites votre *commit*. Dans l'onglet *Git repositories* (à gauche), constatez son impact en dépliant au besoin les différents dossiers.

Revenez à la perspective Java, et modifiez le message affiché dans votre `main()`. Constatez la modification de présentation des différents constituants de votre projet dans le *Package Explorer*.

Il n'est pas nécessaire de basculer tout le temps entre les deux perspectives *Java* et *Git* : la vue[5] *Git Staging* peut faire partie de votre perspective Java : si ce n'est pas le cas, vous pouvez la rajouter manuellement via le menu *Window* > *Show view*. Voyez comment `Main.java` y est signalé comme ayant été modifié par rapport au dépôt local (« *unstaged* »)

Via le *Package Explorer*, vous pouvez directement comparer votre fichier avec sa version précédente (*Compare with* > *Index*, voir figure 6), ou revenir à la version précédente (*Replace with* > *Index*). De fait, tout ce qui a été versionné (cf. *commit*) et les éditions en cours sur la machine locale sont disponibles : un vrai filet de sécurité quand on édite beaucoup de fichiers !

Sur ces mêmes principes, vous pourrez construire progressivement votre programme, ajouter des paquetages et/ou des fichiers au fur et à mesure, les modifier, et enregistrer aux moments opportuns des versions des fichiers de votre choix dans votre dépôt local.

Développer à plusieurs

Voici venu le temps de partager votre travail avec vos équipiers. Pour cela, le plus simple est d'utiliser le dépôt que vous avez créé sur GitLab au début de l'activité. Vous allez donc pratiquer le cas 1. identifié dans la partie « Principes de fonctionnement de Git ».

5. Dans Eclipse, une vue correspond à ce qui est affiché dans une fenêtre.

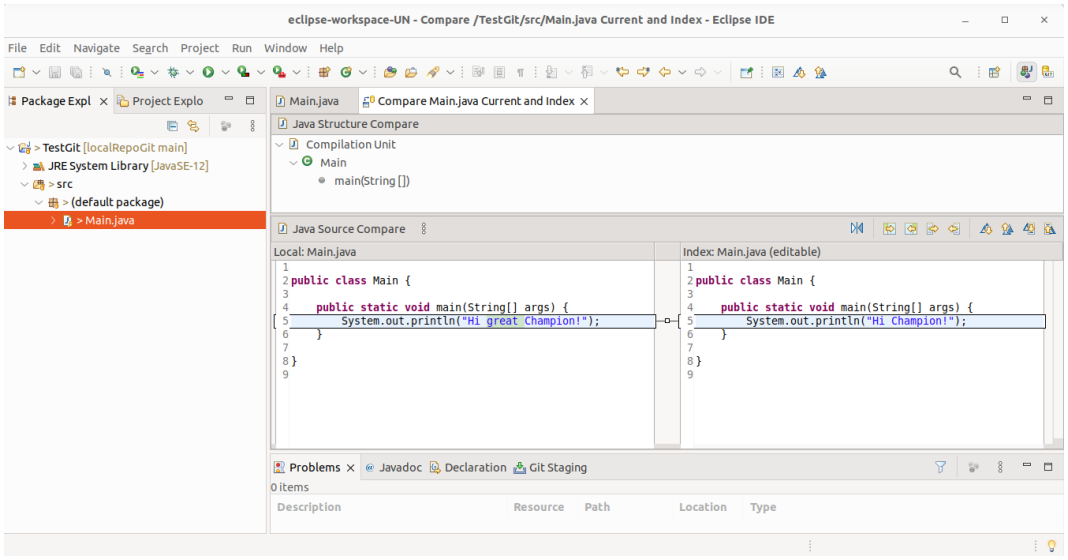

FIGURE 6 – Comparaison de la version actuelle et de la dernière vesion commitée

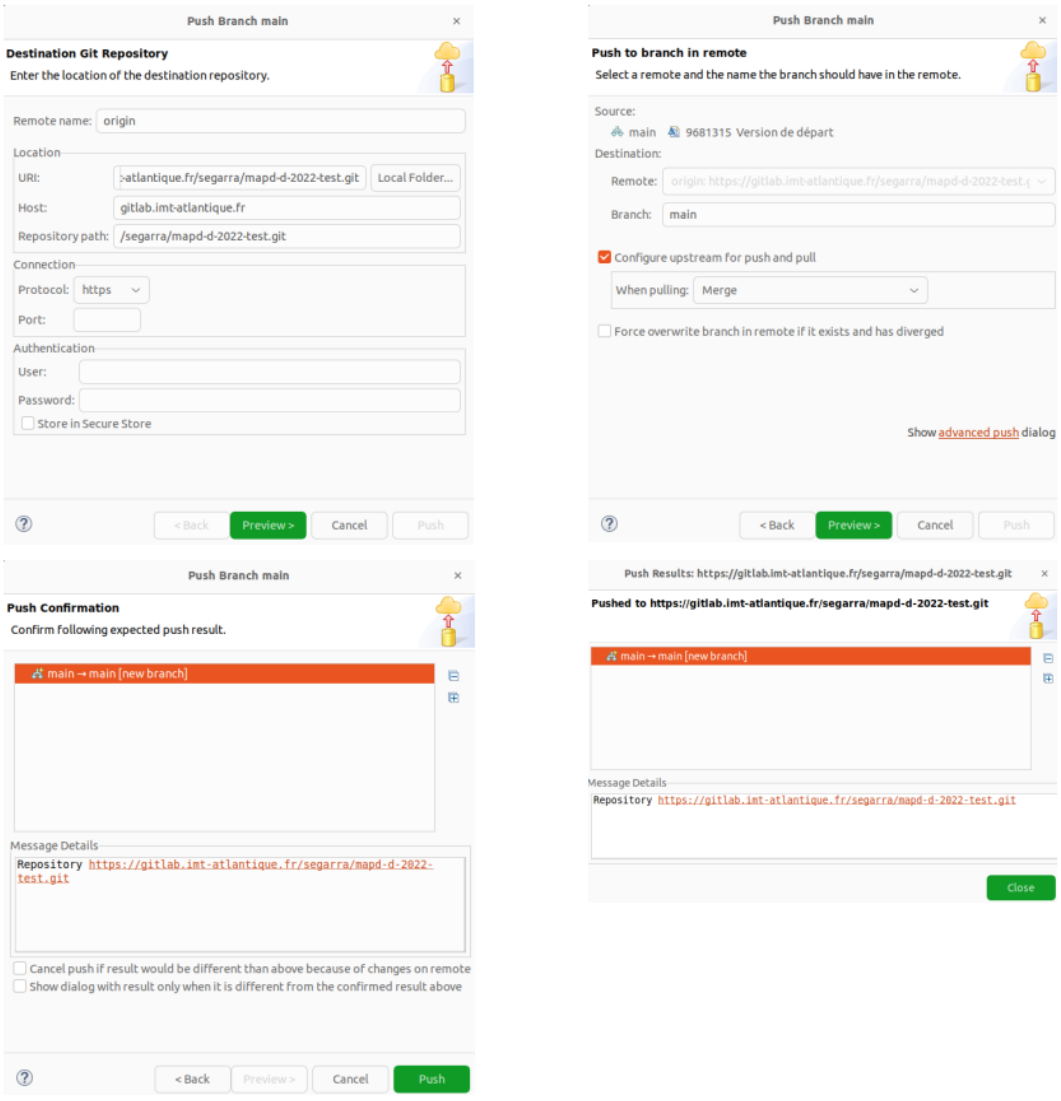

FIGURE 7 – Enchaînement d'écrans pour pousser un projet

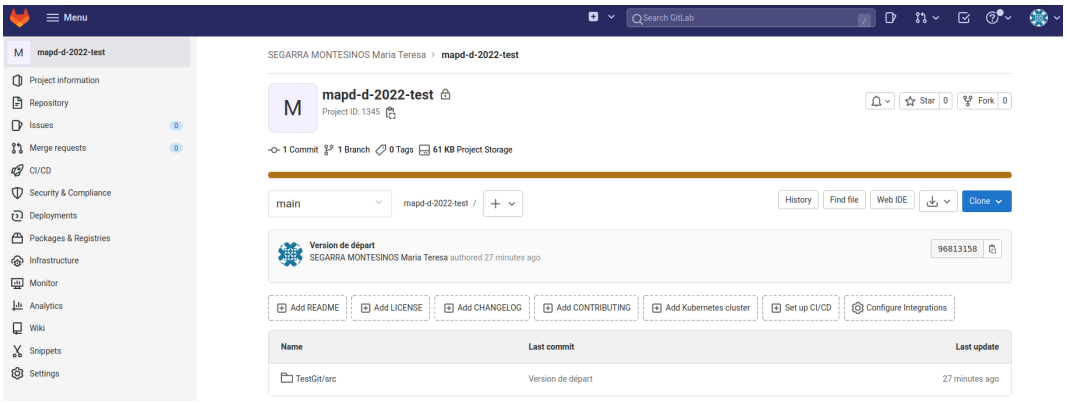

FIGURE 8 – Le projet à été poussé sur GitLab

Publier son dépôt local

Rendre disponible une copie distante de son dépôt local s'appelle « faire un *push* », ou « pousser » son projet. Pour ce faire, dans le *Project Explorer*, faites un *Team* > *Push branch main...* de votre projet.

Saisissez l'URI de votre dépôt sur GitLab. Vous trouvez cette information sur GitLab, bouton *Clone*. Choisissez l'URI donnée dans *Clone with HTTPS*.

La partie *Authentification* permet d'entrer une fois pour toutes ses identifiant et mot de passe pour ce dépôt (NB : ce sont donc ceux du compte école), mais ce n'est pas forcément une bonne idée de stocker son mot de passe école dans une configuration Eclipse... Ces informations vous seront donc demandées à la volée un peu plus tard.

Gardez toutes les options par défaut jusqu'à la finalisation de votre opération : les copies d'écran de la figure 7 illustrent l'enchaînement global de ces opérations.

Vous pouvez maintenant consulter le dépôt distant directement sur GitLab (voir figure 8) : les versions qui avaient été enregistrées localement y sont maintenant répertoriées.

Vous pouvez afficher le code de votre `Main.java` directement via le navigateur.

Quel est le message prévu dans cette version de `Main.java` ? La modification que vous aviez apportée au message à afficher n'a pas été prise en compte ! C'est normal, vous n'aviez pas fait le *commit* avant de faire votre *push*...

Il faut donc d'abord faire un *commit* pour enregistrer cette nouvelle version sur votre dépôt local :

- Ajoutez `Main.java` à l'index des fichiers (*Add* > *Add to index*)
- Faites votre *commit* (NB : n'oubliez pas d'associer un message de version adéquat)

Si vous demandez maintenant à Eclipse une comparaison de votre `Main.java` avec la version courante (*Compare with* > *Index*), vous constatez qu'il n'y a maintenant plus de différence : votre version de travail (celle d'Eclipse) est en phase avec la dernière version de votre dépôt local.

Cependant, comme vous pouvez aussi le constater sur GitLab, le dépôt distant n'est pas à jour (rien n'a changé pour l'instant).

Faites maintenant votre *push*, et constatez la propagation sur le dépôt GitLab (voir figure 9). À noter qu'il est possible de demander un *push* dès le *commit* : il y a un bouton *Commit&Push* pour cela.

Avant de pouvoir travailler avec d'autres personnes, il faut donc bien comprendre ces 3 niveaux de réplication des fichiers :

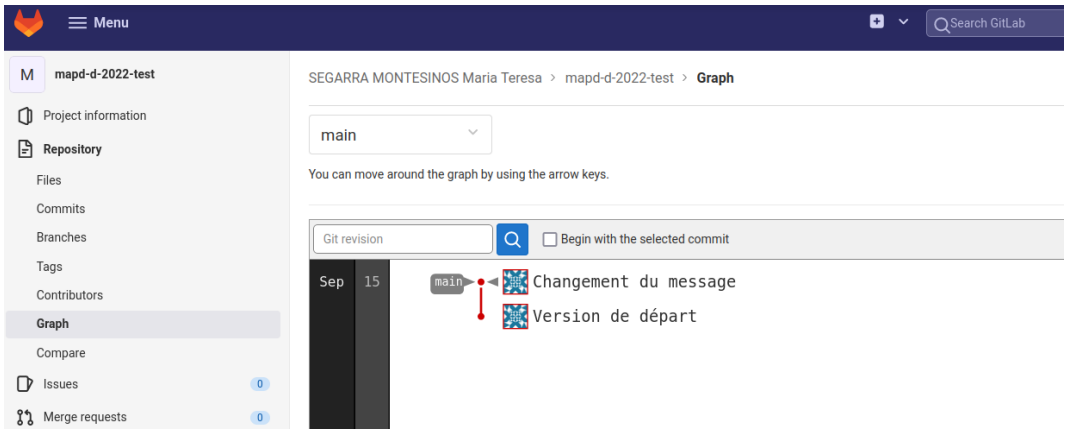

FIGURE 9 – GitLab : il y a bien eu 2 *push*

— Le répertoire de travail : dans notre configuration, c'est directement l'espace de travail d'Eclipse. Les mises à jour se font donc via la commande *Save* d'Eclipse. La version précédente est alors écrasée ;
— L'espace local Git : il comprend la *staged area* et le dépôt local (et la version locale courante). La première contient toutes les modifications sur les fichiers du répertoire de travail qu'on souhaite valider. Avec Eclipse, il est possible de déplacer des fichiers de « *staged* » à « *unstaged* » pour indiquer les modifications qu'on souhaite valider [6]. Le deuxième est alimenté par les commandes *commit*, commandes qui valident les mises à jour de la *staged area*. Lorsqu'un *commit* valide une version d'un fichier, cette version devient la version courante du fichier dans le dépôt local ;
— Le dépôt distant que vous alimentez via le *push* de Git.

Travailler à plusieurs

Nous allons voir maintenant comment Git permet à plusieurs personnes de contribuer au même projet. Pour pouvoir jouer cette initiation tout seul, vous allez vous-même jouer le rôle d'équipier (en plus de votre propre rôle). Il vous faudra pour cela utiliser deux espaces de travail Eclipse différents : vous en avez déjà `UN`, vous allez donc créer un nouvel espace nommé `DEUX`. Et il faudra être très attentif à celui que vous utilisez lors des manipulations qui vont suivre, car sinon rien ne se passera comme prévu. **Les 2 rôles sont désignés par UN et DEUX dans la suite de l'activité**. Eclipse permet de passer d'un espace de travail à l'autre (via le menu *File > Switch workspace*), mais nous vous recommandons plutôt de lancer simultanément deux instances de Eclipse, une avec chaque espace de travail.

Lancez donc une nouvelle fois Eclipse (sans arrêter l'instance que vous avez lancé au début), en choisissant `DEUX` comme nouvel espace de travail.

Le travail d'initialisation du projet dans le dépôt distant ne doit être fait qu'une seule fois. Pour travailler sur un projet déjà mis à disposition sur un dépôt (distant), la logique de travail sera la suivante :

— Faire un clone local du dépôt distant, c'est-à-dire créer un dépôt local contenant une copie du projet distant, avec toutes les informations nécessaires à Git pour pouvoir synchroniser les actions ultérieures

6. Pour ceux qui connaissent Git, c'est l'équivalent des commandes `git add/rm/mv` ...

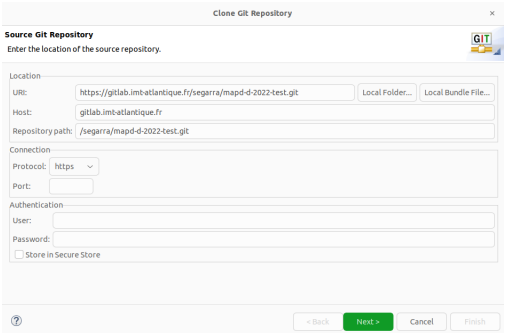

FIGURE 10 – Cloner le dépôt distant

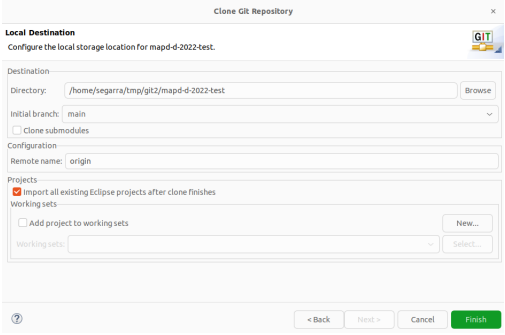

FIGURE 11 – Faire le lien entre le dépôt local et un dépôt distant

— Travailler sur cette copie locale avec Eclipse, en faisant notamment des *commit* et des *push* comme vu précédemment, et toutes les autres actions proposées par Git

Il faut cependant commencer par mettre dans le dépôt distant tous les fichiers de configuration qu'Eclipse utilise pour gérer les projets : dans `UN`, passez en « *staged* » les 4 fichiers marqués « *unstaged* » et faites un *Commit&Push* si vous ne l'avez pas encore fait.

Tous les fichiers du projet sont maintenant disponibles dans le dépôt distant.

Cloner un projet existant

Dans la vue *Git repositories* (cf. *Perspective Git*) de l'Eclipse `DEUX`, choisissez *Clone a git repository* et entrez-y l'URI du dépôt GitLab du projet sur lequel vous allez travailler (voir figure 10). La figure 11 vous permet d'expliciter le dépôt local où stocker votre projet. **Attention**, la localisation du dépôt doit évidemment être différente de la localisation pour le dépôt local de `UN`.

Précisez également que vous souhaitez importer automatiquement les projets qu'Eclipse trouvera sur le dépôt. Revenez dans la perspective Java : il y a maintenant un nouveau projet correspondant au dépôt distant : vous pouvez travailler !

Synchronisation de plusieurs développeurs

Cette partie, vise à mettre en évidence les aspects synchronisation de Git entre plusieurs développeurs.

Développeur DEUX

Le développeur `DEUX` est chargé de traduire en français les messages affichés par le programme. Il va donc changer « Hi great Champion » en « Salut grand champion ! ». Faites cette modification dans Eclipse `DEUX` et vérifiez que cela fonctionne correctement (> *Run*). Comme le travail est terminé, faites un *Commit&Push*. Vous pouvez vérifier sur GitLab que le dépôt distant a bien intégré cette modification.

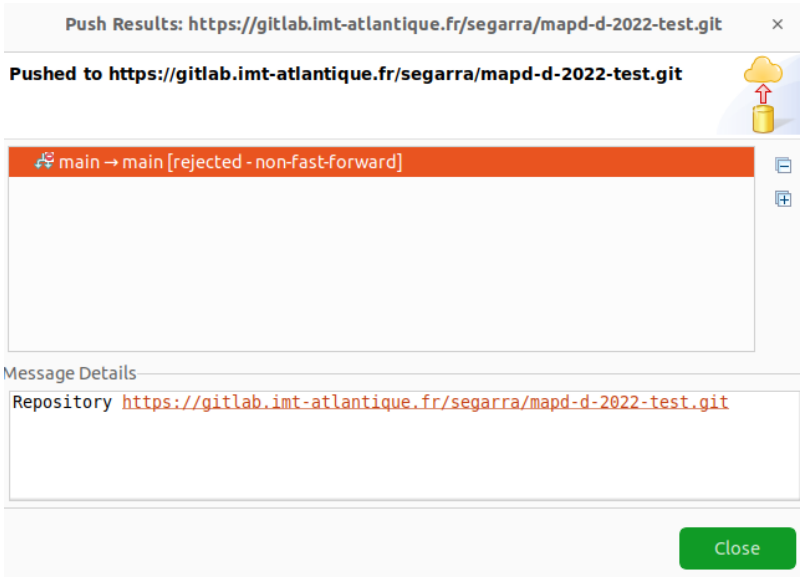

FIGURE 12 – Echec du *push*

Développeur UN

Pendant ce temps, le développeur `UN` ajoute en début du `main()` l'affichage de « Starting the dialog ».

Faites cette modification dans Eclipse `UN` et vérifier que cela fonctionne correctement. Comme le travail est terminé, faites un *Commit&Push*.

Que se passe-t-il ? Il est indiqué que l'opération *push* a échoué (voir figure 12) : on peut le vérifier sur GitLab où le dépôt distant n'a pas été modifié celle fois-ci.

Par contre, le *commit* a bien été réalisé : le dépôt local de `UN` a une nouvelle version, comme le montre l'onglet *History* de la perspective Git (voir figure 13). En effet, Git « sait » que `UN` a modifié une version du fichier qui a entre temps été modifiée. Il faut donc que `UN` prenne en compte les modifications apportées par `DEUX` avant de pouvoir faire son *push*. Nous allons utiliser pour cela la commande *pull*. Sélectionnez `Main.java` et appliquez-lui (clic droit) la commande *pull* : *Team* > *Repository* > *Pull*. La fenêtre d'édition de `Main.java` comporte maintenant toutes les modifications que `UN` doit prendre en compte pour décider de son action (voir figure 14).

Remarquez la façon qu'a Git de signaler les parties qui ont changé sur le dépôt distant et celles qui ont changé sur le dépôt local [7].

Il faut maintenant que `UN` fasse ses choix, et produise une nouvelle version qui tienne compte de toutes ces informations. Dans notre exemple, `UN` va donc conserver la traduction faite par `DEUX` et l'affichage de « Starting the dialog » comme illustré dans la figure 15. Il peut ensuite faire un *Commit&Push*, qui cette fois-ci va pouvoir aboutir.

7. Une difficulté dans notre cas est que l'auteur (cf. champ « author ») des différentes versions est le même pour `UN` et `DEUX` (même identifiant pour la connexion sur le dépôt distant). Lors d'un vrai travail en équipe, ce champ permettra de distinguer clairement les versions poussées par chaque équipier.

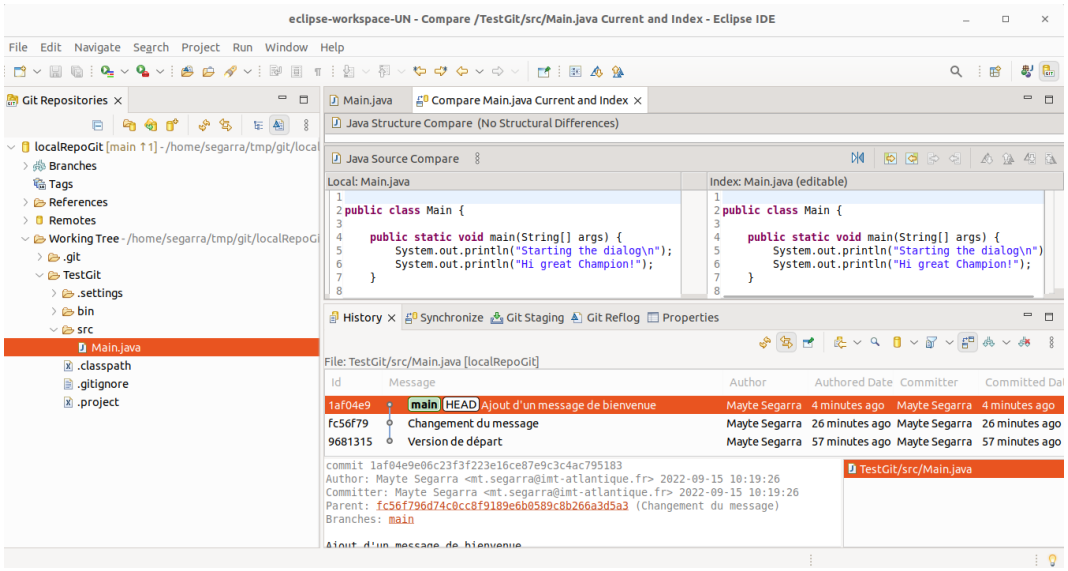

FIGURE 13 – Historique des versions sur `UN`

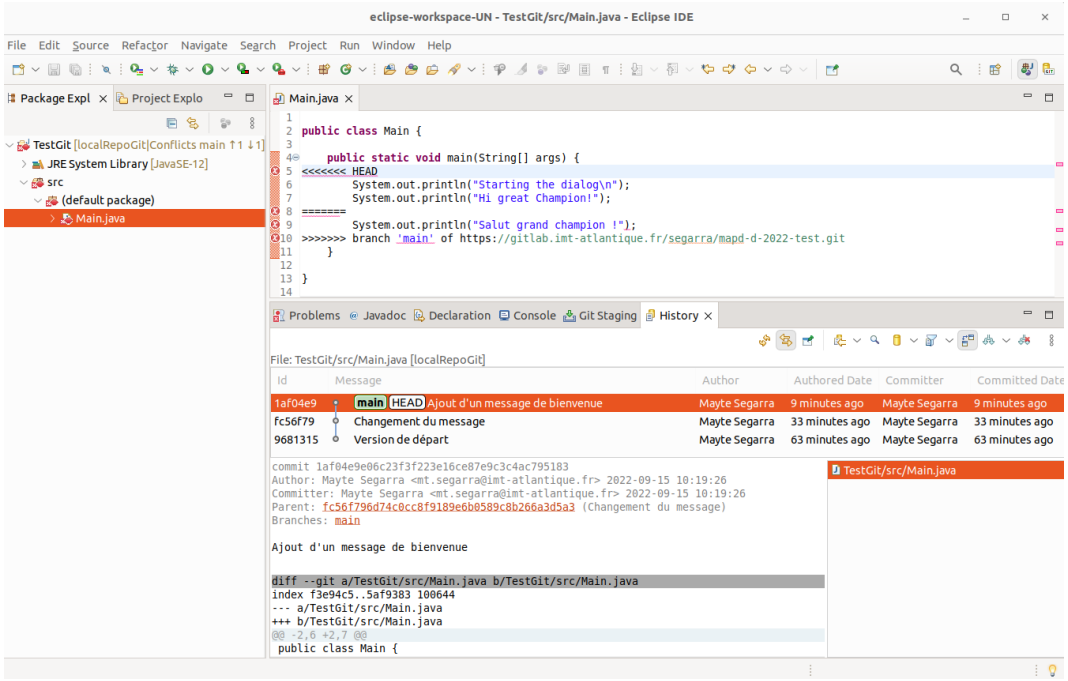

FIGURE 14 – Résultat du *pull*

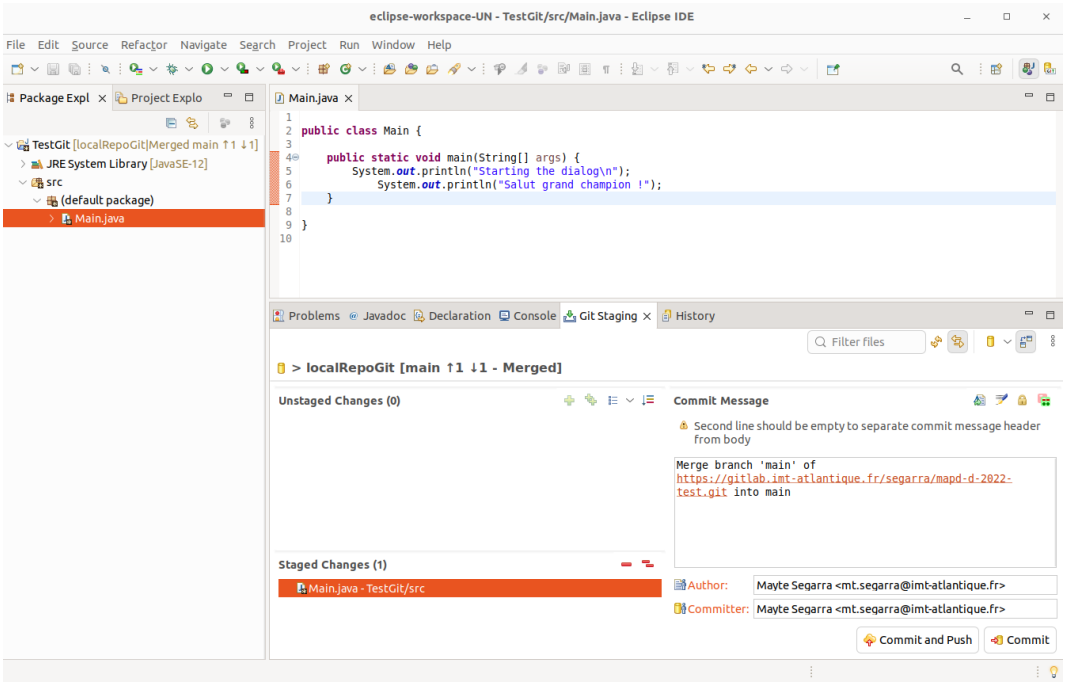

FIGURE 15 – `UN` a fait le tri des modifications

Développeur `DEUX`

Quelque temps plus tard, `DEUX` veut voir s'il y a de novueaux messages à traduire dans le proramme. Il va donc resynchroniser son dépôt local avec le dépôt distant. Il va évidemment utiliser lui aussi la commande *pull* pour cela.

Faites le *pull* sur le projet de `DEUX`. Vous pouvez constater que sa version de `Main.java` est mainenant à jour (même version que la figure 15). Il faut donc que `DEUX` traduise le premier message, qui est en anglais, et fasse ensuite un *Commit&Push*. Faites-le.

Vous connaissez maintenant les principes de base de Git et vous les avez pratiqué en utilisant Eclipse.

Pour aller plus loin

Quels sont les formats de fichiers gérés efficacement par Git

Le type préférentiel de fichier géré par Git est le texte brut. Il saura donc gérer efficacement les fichiers de ce type, qu'il s'agisse de code informatique (p.ex. programme Java) ou de document d'information (p.ex. fichier LaTex). Par contre, Git n'apportera pas ses pleines fonctionnalités avec des fichiers dits « binaires », typiquement produits par des traitements de texte Wysiwyg (MS-Word ou LibreOffice). Donc, un projet sans production de code et avec une documentation différente de texte brut n'est pas le meilleur candidat pour utiliser avantageusement Git.

Les autres commandes Git

Vous avez principalement découvert l'utilisation des commandes Git suivantes :

— *commit*, pour propager les modifications de l'espace de travail vers le dépôt local ;
— *push*, pour propager les modifications du dépôt local vers le dépôt distant ;
— *pull*, pour rapatrier sur le dépôt local les modifications apportées par d'autres au dépôt distant.

De façon un peu cachée, vous avez également utilisé *clone* (pour créer une copie locale du projet), *add* (pour déclarer les fichiers candidats au prochain *commit*), *log* et *diff* (pour afficher les différentes versions et leurs différences), *status* (qui se traduit par les petites annotations graphiques dans Eclipse). Avec cela, vous pouvez déjà travailler pour de vrai. Il existe cependant de nombreuses autres commandes, que vous pourrez découvrir plus tard, selon vos envies et vos besoins : vous pourrez vous référer au site `https://ndpsoftware.com/git-cheatsheet.html` pour avoir une vision synthétique de tout ce qui existe (il suffit de choisir une « colonne » pour afficher toutes les commandes qui s'appliquent à cette colonne).