Rapport

Présentation du projet

Ce projet a eu pour but de créer un logiciel de gestion de versions, c'est-à-dire un logiciel permettant le suivi et la gestion des différents fichiers composant un projet.

En effet, lors d'un projet informatique, on effectue énormément de modifications et on aimerait sauvegarder ces modifications et garder un historique de ces différentes versions. De plus, il se pourrait que l'on se trompe lors de la modification d'un fichier du projet, et on aimerait pouvoir revenir en arrière, remplacer la version actuelle par une précédente. Dans le même temps, si l'on travaille à plusieurs, ce qui est souvent le cas, le fait que chacun puisse travailler sur différentes parties et versions du projet en même temps permet de gagner en productivité. On voudrait donc garder une trace de ces différentes versions du projet, pouvoir regarder qui a effectué des modifications sur telle version du projet, et on aimerait pouvoir fusionner certaines versions entres elles une fois le moment venu.

Notre projet a donc pour vocation de répondre à ces objectifs.

Structure du projet

Pour pouvoir répondre à ces objectifs, notre projet contient différentes structures :

Liste chaînée

Cette structure nous permet de lier différentes données entres elles et de pouvoir les manipuler.

WorkFile et WorkTree

Un WorkFile est une structure permettant de représenter un fichier, en effet cette structure est composée du nom du fichier représenté, de son hash et de son mode, c'est-à-dire des droits en écriture, lecture et exécution liés à ce fichier. Un WorkTree est un tableau de WorkFile, cette structure permet donc de regrouper entres eux les WorkFile.

Commit

Un commit est une structure permettant de soumettre une nouvelle version du projet ainsi que les différentes informations liées à celle-ci. Pour ce faire, un commit est en fait une table de hachage contenant forcément un WorkTree, donc un ou des fichiers, et d'autres informations facultatives telles que l'auteur, la date de soumission, et un message.

Notre projet contient de nombreux fichiers préfixés par les noms ci-dessous :

LC

Ces fichiers contiennent toutes les informations permettant de manipuler les listes chaînées.

hachage

Ces fichiers contiennent toutes les informations permettant de hacher et d'obtenir les hash des différents fichiers composant un projet.

directory

Ces fichiers contiennent toutes les informations permettant de manipuler les répertoires et les composantes du répertoire.

worktree

Ces fichiers contiennent toutes les informations permettant de manipuler les WorkFile et les WorkTree.

commit

Ces fichiers contiennent toutes les informations permettant de créer un commit et de les manipuler.

branch

Ces fichiers contiennent toutes les informations permettant de créer des branches, qui sont des successions de commit sous forme de listes chaînées, la branche principale étant la branche master, et de les manipuler. Pour ce faire nous utilisons dans ce projet des références, qui sont les fichiers contenant uniquement le hash d'un commit. Nous avons des références pour chaque branches créés, ces références contiennent le hash du dernier commit de la branche associée, et il y a la référence HEAD qui indique le commit courant.

merge

Ces fichiers contiennent toutes les informations permettant de fusionner intelligemment deux branches entres elles.

main

Ces fichiers permettent de tester les fonctions des fichiers précédents. Ne pas oublier de lancer le script bash associé, si demandé dans le fichier main en question, afin d'obtenir les prérequis utiles au teste des différentes fonctions.

En fait, notre projet se base sur le hachage, en effet chaque fichier à sauvegarder pour une version donnée est haché et placé dans un répertoire lié à son hash. Ces hash sont renseignés dans les WorkFile, qui sont eux-mêmes renseignés dans des WorkTree. Puis, on hache le Worktree et on renseigne ce hash dans des commits, qui eux-mêmes sont hashés et renseignés dans les branches. Ces branches correspondent aux versions du projet.

Fonctionnement du projet

L'exécutable myGit contient les différentes fonctionnalités de notre projet dont voici le mode d'emploi :

- **./myGit init**
  Initialise le répertoire de références.
  Cette commande crée le dossier .refs/ contenant les références du projet à l'aide de la fonction initRefs. Le fichier .current_branch contenant la branche actuelle est également créé à l'aide de initBranch.

- **./myGit list-refs**
  Affiche toutes les références existantes.
  Cette commande crée une liste chaînée des fichiers présents dans le dossier .refs/ grâce à la fonction listdir, la convertie en chaîne de caractères grâce à ltos et l'affiche.

- **./myGit create-ref <name> <hash>**
  Crée la référence <name> qui pointe vers le commit correspondant au hash donné.
  Cette commande appelle la fonction createUpdateRef afin de créer le fichier nommé <name> et y écrire <hash>.

- **./myGit delete-ref <name>**
  Supprime la référence <name>.
  Cette commande appelle la fonction deleteRef qui supprime le fichier représentant la référence nommée <name>.

- **./myGit add <elem> [<elem2> <elem3> …]**
  Ajoute un ou plusieurs fichiers/répertoires à la zone de préparation (pour faire partie du prochain commit).
  Cette commande appelle autant de fois la fonction myGitAdd qu'il y a d'éléments. Cette fonction ajoute les éléments au fichier .add .

- **./myGit list-add**
  Affiche le contenu de la zone de préparation.
  Cette commande lit le fichier .add ligne par ligne et affiche les éléments qui y sont présents.

- **./myGit clear-add**
  Vide la zone de préparation.
  Cette commande écrase le fichier .add en l'ouvrant en mode écriture.

- **./myGit commit <branch_name> [-m <message>]**
  Effectue un commit sur une branche, avec ou sans message descriptif.
  Cette commande appelle la fonction myGitCommit, qui va obtenir un WorkTree à partir du contenu du fichier .add, ensuite ce WorkTree est hashé et un commit est créé avec ce hash et le message si l'option m est donnée.

- **./myGit get-current-branch**
  Affiche le nom de la branche courante.
  Cette commande appelle la fonction getCurrentBranch qui va lire le contenu du fichier .current_branch et l'afficher.

- **./myGit branch <branch_name>**
  Crée une branche qui s'appelle <branch_name> si elle n'existe pas déjà.
  Si la branche existe déjà, la commande doit afficher un message d'erreur.
  Cette commande, si la branche <branch_name> n'existe pas, appelle la fonction createBranch qui crée le fichier <branch_name> et y place le contenu de HEAD.

- **./myGit branch-print <branch_name>**
  Affiche le hash de tous les commits de la branche, accompagné de leur message descriptif éventuel. Si la branche n'existe pas, un message d'erreur est affiché.
  Cette commande appelle la fonction printBranch qui va parcourir la liste chaînée qu'est une branche grâce au champs predecessor du commit associé et afficher à chaque fois le contenu de la branche, c'est-à-dire le hash de ce commit.

- **./myGit checkout-branch <branch_name>**
  Réalise un déplacement sur la branche <branch_name>.
  Si cette branche n'existe pas, un message d'erreur est affiché.
  Cette commande appelle la fonction myGitCheckoutBranch qui met à jour, si la branche existe, les fichiers .current_branch et HEAD afin qu'elle pointe vers le dernier commit de <branch_name> et restore le commit associé en appelant la fonction restoreCommit.

- **./myGit checkout-commit <pattern>**
  Réalise un déplacement sur le commit qui commence par <pattern>.
  Des messages d'erreur sont affichés quand <pattern> ne correspond pas à un seul commit.
  Cette commande appelle la fonction myGitCheckoutCommit qui permet de se déplacer sur la branche dont le dernier commit correspond au pattern, pour ce faire si le pattern est associé à un seul commit alors on peut se déplacer, sinon s'il est associé à plusieurs commits alors on demande à l'utilisateur de préciser le pattern, sinon on affiche un message d'erreur car le pattern est erroné.

- **./myGit merge <branch> <message>**
  Réalise une fusion de la branche <branch> avec la branche courante.
  Cette commande, s'il n'y a pas de collisions, appelle la fonction merge qui va récupérer le WorkTree associé à la branche courante et à <branch>, va fusionner ces WorkTree, et crée un nouveau commit sur la branche courante avec ce nouveau WorkTree. Si au contraire, il y a des collisions, alors on va demander à l'utilisateur de choisir entre soit garder les fichiers en collisions de la branche courante, soit ceux de <branch>, soit de choisir un par un. Ces choix vont permettre par la suite d'appeler la fonction merge car les collisions auront été éliminées.