Compte Rendu du projet de structures de données

Réalisation d'un logiciel de gestion de versions

David Pulido Cornejo, Ana Sofía Mora Villagómez

15 avril 2023

1 L'objectif du projet :

Git est un logiciel gratuit qui permet à des milliers de programmeurs de travailler simultanément sur des différentes versions d'un projet qui n'évolue pas forcement de forme linéaire. Dans le cadre de ce projet nous allons nous intéresser à reconstruire les structures de données et les fonctions de git permettant à ses utilisateurs d'enregistrer ses différentes versions d'un projet pour pouvoir manipuler ces différentes instances du projet et fusionner les nombreuses versions du projet. Construire un gestionnaire inspiré en git est un projet exigeant à cause des différentes structures de données employées qui peuvent perdre facilement un développeur inattentif. Mais ce projet est tres gratifiant grâce au défi qu'il impose. En plus, une fois que le programmeur ait compris le fonctionnement du gestionnaire, le projet permet à son créateur de contempler la beauté des abstractions en l'informatique.

2 Fonctionnement général :

Pour pouvoir gérer un projet et ses différentes instances nous devons tout d'abord comprendre comment nous allons représenter l'ensemble de fichiers qui constituent le projet. Pour cela nous allons construire deux structures qui vont être comme les cellules et les noyaux de notre gestionnaire. Ces deux structures vont être WorkTree et WorkFile. Chaque fichier de notre projet va être enregistré dans un WorkFile qui contiendra le nom, le hash et le mode du fichier qu'il représente. Pour obtenir le hash du contenu d'un fichier nous allons employer la fonction Sha256, nous dévoilerons prochainement l'utilité de ce hash. Les WorkFiles associés aux documents du projet vont être stockés dans un tableau de WorkFiles encapsulé par une structure nommée WorkTree. Cet ensemble de structures nous permettra de répertorier les documents d'un projet. Maintenant que les structures représentant un projet sont bien définies, il est nécessaire de concevoir les protocoles de sauvegarde des versions du projet et comment l'utilisateur accèdera à ces instances de son projet. Pour sauvegarder une instance d'un ensemble de documents nous allons créer des blobs (binary large object), ils s'agissent des copies d'un document à un instant donné. Les blobs auront pour nom le hash de leur contenu dans lequel un slash est introduit

après le deuxième caractère, en plus à ce hash est concatené une extension. Cette duplication de documents est effectuée par la fonction blobContent dans notre projet. Alors pour sauvegarder une instance d'un projet nous devons dupliquer avec blobContent tous les documents du WorkTree représentant le projet. Cette opération est effectuée par la fonction saveWorkTree. Elle va creer les copies de tous les documents d'un WorkTree et modifier les hash dans les WorkFiles de tous les documents, le hash de chaque document deviendra le hash correspondant à une de ces copies. C'est ainsi que le WorkFile fait le lien entre un document et une copie d'une de ses instances. Un WorkTree va nous permettre d'accéder à une version d'un ensemble des documents. Nous allons sauvegarder les WorkTrees dans des blobs qui sont des fichiers qui décrivent tout leur contenu. Dès que nous avons plusieurs WorkTrees il est possible d'accéder aux différentes versions d'un projet. Nous allons créer la fonction restoreWorkTree qui est comme l'inverse de saveWorkTree. Si un WorkTree correspond à l'enregistrement d'une instantanée du projet, restoreWorkTree va copier le contenu de cette version passé du projet sur les fichiers manipulés par le client. C'est ainsi que nous avons établit une structure pour garder les fichiers d'un projet et les copies des différentes versions du projet au cours du temps.

Voici quelques chemins qui synthétisent les idées que nous venons d'expliquer :

Nous allons stocker chaque WorkTree sous la forme de fichiers qui sont des blobs générés par saveWorkTree, ces fichiers nous permettent d'accéder à des versions passées des projets. Ces fichiers auront des noms incompréhensibles pour le client puisqu'ils sont dérivés de leur hash. En plus ce n'est pas pratique pour l'utilisateur de manipuler ces fichiers pour manipuler ces instances du projet car il doit se souvenir de la chronologie de chaque WorkTree. Pour résoudre ce problème nous allons introduire les commits. Un commit est une structure contenant une table d'hachage où nous allons stocker le hash (qui peut être vu comme le nom) d'un WorkTree, un auteur, un message et un hash correspondant au commit qui le précède. C'est ainsi que les commits vont former une liste qui imposera un ordre chronologique sur les WorkTrees construits. Plus on avance dans la liste, plus ancien devient le commit. Les commits seront aussi enregistrés par des fichiers crées par la fonction blobContent. Le problème de ces listes de commits c'est qu'en se déplaçant dans la liste pour employer restoreWorktTree et accéder aux versions précédentes des projets, on perd le nom du hash des commits plus récents et nous ne pouvons pas revenir en arrière pour les retrouver. Pour résoudre cet inconvénient, nous allons créer un répertoire caché «./refs» où nous allons créer les fichiers HEAD et MASTER. MASTER contiendra toujours le hash du commit plus récent, pendant que HEAD va nous permettre de nous déplacer entre nos commits, HEAD contiendra le hash du commit sur lequel on travaille.

C'est ainsi que les références nous permettront voyager dans notre histoire d'enregistrements. Si le client souhaite travailler sur une version précédente de son projet pour faire des modifications qui divergent par rapport aux changements qu'il à fait précédemment, il peut restaurer un commit du passé, modifier les documents à partir de cette instance et faire une nouvelle séquence de commits. Nous allons avoir alors deux listes de commits qui convergeront à partir d'un certain commit. En effet la façon dont laquelle nous avons crée les commits nous permet de construire des graphes orientés de commits. Nous devons

alors créer des structures pour manipuler les branches de nos graphes. Nous allons créer alors la structure brach, une branch va être un fichier dans ./refs dont le nom sera attribué par l'utilisateur et branch contiendra aussi le hash de la feuille (dernier commit) d'une branche de du graphe. Nous pouvons imaginer qu'une branch est l'équivalent d'un master pour chaque branche, c'est une étiquette qui nous permet de suivre l'évolution de chaque branche et de nous repérer sur le graphe . Pour manipuler ces branches nous devons savoir dans quelle branche le client travaille à un instant donné, pour cela nous allons créer le fichier caché "current_branch" qui contiendra le nom de la branche sur laquelle le client est situé. Nous allons proposer à notre client de ce déplacer de deux façons possibles dans notre graphe d'instances. Soit il va choisir un commit et le restaurer grâce à la fonction myGitCheckoutCommit place le Head sur le commit choisi et le restaure, ou soit il emploi myGitCheckoutBranch qui permet au client de restaurer le dernier commit de la branche choisie, en ce cas le HEAD est placé sur ce dernier commit et le fichier ".current_branch" est actualisé.

Maintenant que nous avons établi l'ensemble de structures et de fonctions permettant au client de se déplacer sur les différentes versions de son projet, nous devons fournir à l'utilisateur les moyens pour enregistrer son travail dans ce gestionnaire. Pour cela nous allons créer le fichier caché ".add" qui va être notre zone de préparation pour générer des commits. Ce fichier va être un WorkTree auquel le client peut ajouter tous les documents qu'il souhaite grâce à la fonction myGitAdd. Une fois que le client ait une nouvelle version du projet, il pourra la sauvegarder grâce à la fonction myGitCommit. Cette dernière fonction va générer un commit représentant le contenu de ".add" (qui sera vidé après l'exécution de la fonction). Ce commit contiendra un message rédigé par le client et il sera placé au lieu qui lui correspond dans l'arbre. En plus la référence branch de la branche du nouveau commit est actualisée avec HEAD.

Nous avons construit un gestionnaire qui permet à son utilisateur de manipuler des différentes versions de son projet et de les sauvegarder. Mais normalement dans un projet il est désirable de pouvoir avoir une version finale et fonctionnelle qui contient toutes les parties essentielles de chacune des instances sur lesquelles l'utilisateur a travaillé. Dans son état actuel notre gestionnaire n'est pas très différent à un album de photos. C'est pour cela que nous introduirons la fonction de fusion merge qui permet de faire un nouveau WorkTree avec les fichiers du dernier commit d'une branche choisie par l'utilisateur et la ".current_branch" s'il n'y a pas de conflits. On considère que deux documents des WorkTrees de deux commits forment un conflit s'ils ont le même nom et un hash différent. En ce cas l'utilisateur devra choisir un commit où les documents incompatibles seront effacés par createDeletionCommit. Une fois que les conflits soient gérés par l'utilisateur, il pourra employer merge pour unifier deux branches. Pour rendre notre gestionnaire plus facile à utiliser directement sur la terminale nous avons crée un main qui fonctionne comme un menu qui donne accès à l'utilisateur aux fonctions du gestionnaire qui lui permettent de sauvegarder son projet, aux fonctions permettant la visualisation de l'évolution du projet, aux les fonctions de déplacement sur les commits et les branches et finalement main donnera l'accès aux fonctions de fusion de branches.

3 Gestion des fichiers du code :

3.1 Les différents répertoires du projet :

3.2 La répartition des fonctions :

3.3 Les répertoires du projet :

Dans le projet nous avons repartit les fonctions de notre code principalement entre le répertoire src ("source") et le répertoire include. Dans source nous avons mis les codes source et dans include leur respectifs headers. Les noms des fichiers sont très descriptifs, ils sont nommées d'après la strurcture manipulée dans leur contenu, chaque structure employé dans notre projet a alors son propre code source et header dédié.
Dans notre projet il existe aussi le répertoire bin ou toutes les éxécutables sont gardées et le fichier tests où nous avons rangé les documents employés pour vérifier la validité de notre code. Le MakeFile est situé dehors de ces répertoires internes pour qu'il ait un accès plus direct aux répertoires du projet et ses contenus.

3.3.1 utilities.h :

Les fonctions d'utilities ne forment pas partie des fonctions répondant à un problème de l'énoncé du projet, mais elles correspondent à des fonctions auxiliaires qui facilitent la résolution d'un problème. Les fonctions d'utilities peuvent être aussi des fonctions qui généralisent des comportements ou préforment des actions employés fréquemment dans le projet comme la comparaison de deux chaines de caractères ou faire le blob d'un document.

4 Les fonctions au coeur du projet :

4.1 blobContent() :

La fonction blobContent prend un objet, une chaine de caractères correspondant à une extension et une fonction auxiliaire. La fonction va commencer par faire une copie temporaire de l'objet dans un fichier nommée "tmpXXXXXX". L'objet est transformé en fichier grâce à la fonction auxiliare toFile introduite en paramètres. Cette fonction auxiliaire doit faire un cast de l'objet que nous allons enregistrer et copier son contenu dans un fichier sous le format correspondant à l'objet traité. blobFile va continuer par calculant le sha du fchier temporaire. Ce hash deviendra le nom du fichier blob que nous allons créer. Tout d'abord un répertoire nommé à partir des premières deux lettres du hash est crée, au hash on ajoute entre son troisième et deuxième caractère un slash pour quele blob soit placé dans le répertoire crée. Puis on ajoute l'extension des paramètres à la fin du hash, cette extension nous permettra savoir le type d'objet dont nous avons fait le blob. Finalement une copie du fichier temporaire est faite et elle portera comme nom notre hash modifié et le fichier temporaire sera éliminé. blobContent finit par renvoyer le hash originel du fichier temporaire.

4.2 saveWorkTree() :

Dans saveWorkTree nous voulons faire une copie de tous les documents du WorktTree et avoir un blob décrivant le WorkTree des copies de l'ancien WorkTree. Pour cela il suffit de faire le blob de tous les fichiers dans les WorkFiles et eregistrer les hash des blobs dans la partie hash du WorkTree. Une fois que tous les WorkFiles du WorkTree sont traités, saveWorkTree fait le blob du WorkTree et retourne son hash. Mais la fonction contient une petite subtilité qui vient du fait qu'un répertoire peut être traité comme un fichier, alors un WorkFile peut contenir l'information d'un repertoire. Nous ne pouvons pas fair un blob d'un répertorie car en ce cas on ne ferait pas la copie de son contenu interne. Alors la fonction saveWorkTree va transformer ce répertoire en WorkTree pour s'appeler récursivement sur le WorkTree du répertoire. Dans cet appel récursif la fonction saveWorkTree récupère le hash représenant le hash du blob du WorkTree d'un répertoire, alors saveWorkTree récupère une référence aux nouvelles copies du contenu du répertoire traité. Le blob du fichier en ce cas répertoire est effectué et WorkTree peut continuer à traiter le reste de ses WorkFiles. C'est ainsi que saveWorkTree nous donne un repère pour accèder à un fichier décrivant un WorkTree ayant une copie de tous les documents que contenait le WorkTree mis en paramètres au moment de l'exécution de la fonction.

4.3 restoreWorkTree() :

Dans chaque fichier d'un un WorkTree, restoreWorkTree va chercher la copie du fichier décrite par le hash de son WorkFile et transférer le contenu de cette copie au document originel. Si restoreWorkTree rencontre un workFile représentant un autre WorkTree, restoreWorkTree va faire un appel récursif sur ce nouveau WorkTree. C'est ainsi que restoreWortkTree va faire le transfert du contenu d'une copie passée du projet au fichiers originels à partir desquels ces copies ont été faites.

4.4 myGitCheckoutCommit()

: Lorsque myGitChecoutCommit est employé l'utilisateur doit choisir un motif qu'il transmettra sous la forme d'une chaîne de caractères en dans les paramètres de la fonction. La fonction va créer une liste avec tous les noms des commits existants dans le répertoire de références. Puis, cette liste sera filtré et elle contiendra uniquement les commits dont le nom a pour préfixe le motif introduit par l'utilisateur. Si la liste est vide un message d'erreur indiquant l'inexistence du commit recherché est affiché. Si la liste contient un seul commit, le contenu de la version enregistrée par ce commit est restauré avec restoreCommit, une fonction qui fait appel à restoreWorkTree sur le WorkTree du commit. Finnalment la référence HEAD est actualisé et elle pointe sur le commit restauré. Enfin si la liste contient plusieurs éléments, la liste est affichée pour que le client puis introduire un motif plus spécifique lors du prochain emploie de la fonction myGitCheckoutCommit.

4.5 myGitCheckoutBranch() :

Dans cette fonction une chaîne de caractères branch représentant le nom d'une branche est mise en paramètres. La fonction commence par écrire sur ".current_branch" le nom de

la branch vers laquelle nous allons nous déplacer. Puis la fonction vas extraire le contenu du fichier branch. Ce hash correspond au nom du dernier commit de la branche, la fonction va vérifier que le hash ne soit pas null, s'il est null, le commit n'existe pas alors la fontcion affiche un message d'erreur et elle s'achève. Si le commit du hash existe, la fonction modifie le contenu de la référence HEAD pour qu'elle pointe sur ce commit, et le commit du hash est restauré avec WorkTree. Maintenant l'utilisateur peut accéder à la dernière version enregistré edu projet dans la branche qu'il a choisit.

4.6 myGitAdd() :

mygGtiAdd va être la fonction permettant au client d'indiquer au logiciel quels sont les fichiers du projet qu'il veut enregistrer. Dans notre logiciel il y a un fichier caché ".add". Ce fichier représente un WorkTree dont les WorkFiles sont des répertoires et des fichiers que le client a ajouté à ".add" avec myGitAdd. La fonction GitAdd prend en paramètres le nom d'un fichier (ou répertoire) et elle va commencer par ouvrir le fichier ".add", puis elle va reconstruire la structure informatique WorkTree en lisant ce fichier. Puis un WorkFile contenant le nom du fichier saisit en paramètres, son hash et son nom sera ajouté au WorkTree. Ce WorkTree sera transformé à nouveau en un fichier nomée ."add". C'est ainsi que ".add" est actualisé et maintenant il contient l'information d'un nouveau fichier.

4.7 myGitCommit() :

4.8 merge() :

4.9 createDeletionCommit() :

4.10 main() :