Compte Rendu du projet de structures de données

Réalisation d'un logiciel de gestion de versions

David Pulido Cornejo, Ana Sofía Mora Villagómez

16 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 très 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 informatique.

2 Fonctionnement général :

Pour pouvoir gérer un projet et ses différentes instances nous devons tout d'abord comprendre comment nous allons représenter un ensemble de fichiers constituant un projet. Pour cela nous allons construire deux structures qui vont être comme les cellules et les noyaux de notre gestionnaire. Ces deux structures sont 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 créer 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 passée 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.

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 les 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 de la liste. 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 de la liste de commits sur laquelle le client travaille, pendant que HEAD va nous permettre de nous déplacer entre nos commits, HEAD contiendra le hash du commit sur lequel on travaille. Par la suite, la référence MASTER sera représenté par un fichier nommée ".current_branch" dans "./refs".

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. Dans ce cas il y aura 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 arborescences 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 contien-

dra aussi le hash de la feuille (dernier commit) d'une branche de du graphe. Une branch est une étiquette qui nous permet de suivre l'évolution de chaque branche et de nous repérer sur l'arbre. Pour manipuler ces branches nous devons savoir dans quelle branche le client travaille à un instant donné, nous allons écrire le nom de cette branche dans un fichier caché nommé ".current_branch". 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 qui 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. Pour sauvegarder son projet, l'utilisateur doit ajouter à git ".add" les fichiers qu'il veut enregistrer et utiliser 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 ajouté à la branche qui lui correspond, finalement cette branche 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 le client n'a aucun outil pour pouvoir consolider l'ensemble des instances de son projet dans un projet final.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. Dans 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 lui permettant 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 répertoires du projet :

Dans le projet nous avons reparti 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 structure 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 exécutables 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 leur contenu.

3.2 La répartition des fonctions :

list.h : Ce fichier est le header de list.c qui contient toutes les fonction de manipulation de listes.

file_hash.h : Il s'agit du header de file_hash.c. Ces documents contiennent les fonctions qui manipulent des fichiers de forme générique (c'est à dire qu'elles ne prennent pas en compte la nature du fichier) et manipulent les hash d'un fichier.

worktree_handler.h : C'est le header de work_tree.c, ces documents contiennent la déclaration des fonctions manipulant uniquement des WorkTrees et des WorkFiles.

commit_handler.h : Ce fichier est le header de commit_handler.c, dans ce fichier sont déclarées les fonctions manipulant la structure commits et leur contenu.

reference_handler.h : Les fonctions de reference_handler.c (qui a pour header reference_handler.h) vont manipuler les références du repertoire ./refs sans prendre en compte l'existance des autres branches. Ce sont les opérations plus simples que le projet effectue sur les références. C'est dans ce fichier que sont déclarées aussi les fonctions qui permettent au client de créer des commits sur une brache.

branch_handler.h : Il s'agit du header de branch_handler.c qui contient les fonctions gérant les déplacements sur les branches d'un projet.

merge_hadler.h : Header de merge_handler.c qui contient les fonctions permettant à l'utilisateur de fusionner deux branches du projet.

main_test.c : La fonction main de ce fichier ne prend pas d'arguments. Elle contient un jeu de test sur l'ensemble des fonctions de list.h, file_hash.c, worktree_handler.c et commit_handler.c.

main.c : Ce main prend au moins un argument et il sert comme un menu pour que l'utilisateur puise employer les fonctions du projet qui lui permettront de gérer les différentes versions de son projet qu'il veut enregistrer.

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 est une fonction généraliste qui va être employé pour produire les blobs de WorkTree et de Commit. Elle prend un objet de type void* obj, une chaine de caractères extension, et une fonction généraliste (void(*toFile(void*,char*)). La fonction toFile doit transférer l'information de la structure qu'elle traite (elle doit faire un cast de l'objet réçu) dans un fichier. La fonction blobContent est employée en blobWorkTree avec l'extention ".t" et dans blobCommit. avec l'extension ".c" .
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 calculer le sha du fichier 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 que le 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. La fonction blobContent est une fonction généraliste qui va être employé pour produire les blobs de WorkTree et de Commit. Elle prend un objet de type void* obj, une chaine de caractères extension, et une fonction généraliste (void(*toFile(void*,char*)). La fonction toFile doit transférer l'information de la structure qu'elle traite (elle doit faire un cast de l'objet réçu) dans un fichier. La fonction blobContent est employée en blobWorkTree avec l'extention ".t" et dans blobCommit. avec l'extension ".c" .

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 ses WorkFiles et enregistrer les hash de leur blobs dans la partie hash du WorkFile 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 répertoire. Nous ne pouvons pas faire 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 obtenir son blob en utilisant saveWorkTree. Le blob de ce nouveau WorkTree contiendra l'information pour accéder aux copies du contenu du répertoire. Par conséquent le WorkTree original aura les hash des aux de tout le contenu interne du répertoire qu'il représente. Lors de la sauvegarde d'un WorkTree, saveWorkTree peut rencontrer des documents qui ne sont pas des fichiers ou des répertoires, comme par exemple des tubes ou des links. Dans notre implémentation de saveWorkTree, nous avons décidé d'ignorer

ces cas spéciaux.

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 aux 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 dans les paramètres de la fonction. La fonction va créer une liste avec tous les noms des commits existant dans le répertoire de références. Puis, cette liste sera filtrée 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. Finalment la référence HEAD est actualisée 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 va 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 fonction 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 restoreWorkTree. Maintenant l'utilisateur peut accéder à la dernière version enregistrée du projet dans la branche qu'il a choisit.

4.6 myGitAdd() :

myGtiAdd 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 nommée."add". C'est ainsi que ".add" est actualisé et maintenant il contient l'information d'un nouveau fichier.

4.7 myGitCommit() :

La fonction myGitCommit prend en paramètres le nom d'une branche branch et une chaîne de caractères qui constituera le message du commit. La fonction commence par vérifier que le dossier de références soit accessible, si le répertoire n'est disponible un message d'erreur sera affiché et l'utilisateur devra initialiser le dossier de références. La fonction continue par vérifier que la branche existe, si elle n'existe pas un message d'erreur est aussi affiché. Le troisième test de sécurité effectué par la fonction constitue à vérifier que le dernier commit de la branche choisie correspond au commit du HEAD, la fonction s'achève et affiche a nouveau un message d'erreur si les hash de ces commits diffèrent. Si nous n'effectuons pas ce dernier test, le client aurait la possibilité de créer des arborescences incohérentes où les documents d'une chronologie de travail seraient dans une branche qui ne leur correspond pas. Une fois que les test de sécurité sont effectués, la fonction va extraire le WorkTree décrit par ".add" afin de sauvegarder une copie les fichiers enregistrés par l'utilisateur avec la fonction saveWorkTree. La fonction continue par effacer le fichier ".add" pour qu'il soit prêt pour le suivant commit. La fonction procède par créer un commit dans lequel le WorkTree est celui qu'elle vient de créer, son message est celui saisi par l'utilisateur en paramètres et finalement le prédécesseur est le dernier commit de la branche branch. Un blob du commit est effectué et la fonction actualise HEAD et branch pour qu'elles pointent sur ce blob.

4.8 merge() :

La fonction merge prend en paramètres deux chaînes de caractères, l'une représente une branche nommée "remote_branch" et l'autre un message. Souvenons nous que deux fichiers d'un commit sont en conflit s'ils ont le même nom mais un hash différent, c'est à dire que les deux documents représentent deux instances différentes d'un même fichier. La fonction va employer une fonction auxiliaire pour avoir accès à une liste des fichiers des commits de la branche actuelle et la branche "remote_branch" qui posent des problèmes de cohérence et un WorkTree de leurs fichiers qui ne posent pas des conflits. La fonction se termine si la liste de fichiers problématiques n'est pas vide. Si les deux commits n'ont pas de conflits, la fonction va sauvegarder l'arbre de fusion des commits avec saveWorkTree, un nouveau commit ayant pour WorkTree le blob du WorkTree de fusion et le message des paramètres est crée. Ce commit aura comme particularité le fait d'avoir pour prédécesseur la branche de ".current_branch" et en plus, sa table d'hachage contiendra aussi un élément nommé "merged_predecessor" auquel sera attribué le hash du dernier commit de la branche "remote_branch". Finalement la référence de "remote_branch" est éliminée car les deux branches manipulées par la fonction maintenant forment qu'une seule, la "remote_branch" était devenue redondante. En fin Merge se termine et ne renvoie rien.

4.9 createDeletionCommit() :

La fonction createDeletionCommit prend en paramètres une branche, un message et une liste. La fonction va accéder au WorkTree du dernier commit de la branche mise en paramètres, et elle va ajouter dans le fichier ".add" tous les fichiers du WorkTree qui ne sont pas dans la liste des paramètres. Avec l'aide de la fonction myGitCommit, un nouveau commit est crée sur la branch mise en paramètres. Le noueveau commit portera le message introduit par l'utilisateur en paramètres. Ce commit aura un WorkTree avec presque tous les documents que son commit précédent avec l'exception des fichiers présents dans la liste mise en paramètres.

4.10 main() :

La fonction main de main.c prend plusieurs arguments, le premier d'entre eux doit correspondre au nom du programme "main.c". Si elle reçoit qu'un seul argument, elle va demander d'être exécutée avec au moins deux arguments et elle affichera aussi un message proposant de lui fournir la commande help. Le deuxième argument donné a la fonction main doit correspondre à une instruction. La fonction main aura des différents comportements selon la nature de cette instruction. Voici quelques instructions particulières :

help : La fonction main affiche le menu d'instructions auxquelles elle répond et elle se termine.

merge : Pour effectuer un merge main a besoin d'au moins trois arguments, le troisième argument argv[3] va correspondre à la branche avec laquelle on veut fusionner la branche actuelle. Le quatrième argument, s'il existe, sera le message du commit crée par merge. S'il n'y a pas un quatrième argument le message du commit de merge sera vide. Main va alors créer le worktree de fusion de la branch argv[2] et la branche actuelle et en plus elle va créer leur liste de conflits. Si la liste des conflits est vide le merge entre les deux branches est effectué automatiquement et main se termine. Si non, main offre à l'utilisateur trois solutions. La première option est de saisir 1, en ce cas un merge où les fichiers de la liste des conflits de la branche actuelle sont conservés est fait. Dans le cas où l'utilisateur choisit l'option 2, un merge où les fichiers de la brache argv[2] sont conservés est effectué. La troisième option pour l'utilisateur est d'appuyer la touche 3, en ce cas il devra choisir manuellement quels éléments de la liste de conflits il conservera.

4.11 Mention honorable, print_color() :

La fonction print_color est une fonction supplémentaire d'utilities qui nous a permit de donner plus de vie à notre code en nous facilitant l'écriture de chaînes de caractères avec des différentes couleurs. La fonction prend trois arguments, le premier argument est output qui est un pointeur de FILE, il va indiquer à la fonction où elle doit écrire la chaine de caractères msg qui est en effet le deuxième argument de la fonction. Le troisième argument est une chaine de caractère color. Color peut prendre les valeurs "red", "yellow", "blue", ou "green", pour chaque option le texte écrit sera, respectivement, de couleurs rouge, jaune, bleu ou vert. Si color n'est pas égal à aucune des valeurs précédentes la couleur du texte sera blanche.