2023

RÉALISATION D'UN LOGICIEL DE GESTION DE VERSIONS

Sraieb Siwar

Fall Serigne Fallou

SOMMAIRE

01 Introduction

02 État de l’art : Gestion de versions de code

03 Conception et implémentation du logiciel de gestion de versions

04 Conclusion

05 Références bibliographiques

INTRODUCTION

Notre projet se concentre sur la création d'un outil de suivi et de versionnage de code, similaire à Git,
un logiciel de gestion de versions utilisé dans le développement logiciel. Nous visons à comprendre en détail son fonctionnement, en examinant les structures de données et les fonctionnalités clés.
Nous allons nous concentrer sur les fonctionnalités suivantes :

- Comment permettre à un utilisateur de créer des enregistrements instantanés de son projet?
- Comment permettre à l'utilisateur de naviguer facilement à travers les différentes versions ?
- Comment créer et maintenir une arborescence des différentes versions du projet ?
- Comment vérifier l'identité des utilisateurs ?
- Comment sauvegarder les modifications qui ne sont pas enregistrées dans une version instantanée

Ce projet nous permettra d'acquérir une compréhension pratique des principes du versionnage de code et de développer des compétences en conception d'un outil de gestion de versions robuste et efficace.

ETAT DE L'ART : GESTION DE VERSIONS DE CODE

La gestion de versions de code est une pratique essentielle dans le développement de logiciels, permettant de gérer les modifications, les versions et les collaborations sur un code source. Plusieurs outils de gestion de versions sont populaires sur le marché, tels que Git, Mercurial, SVN, et d'autres.
Dans cette section de l'état de l'art, nous allons examiner ces outils, analyser leurs fonctionnalités, les structures de données et algorithmes utilisés, ainsi que leurs avantages, limites et bonnes pratiques.

1/PRÉSENTATION DES OUTILS EXISTANTS DE GESTION DE VERSIONS

Les outils de gestion de versions les plus couramment utilisés sont :
-Git : un système de gestion de versions distribué largement utilisé, offrant une grande flexibilité, une gestion avancée des branches et une facilité de collaboration.
-Mercurial : un autre système de gestion de versions distribué, offrant des fonctionnalités similaires à Git, mais avec une approche différente.
-SVN (Subversion) : un système de gestion de versions centralisé, qui a été largement utilisé dans le passé mais a perdu en popularité ces dernières années.

2/ANALYSE DES FONCTIONNALITÉS OFFERTES PAR CES OUTILS

Les outils de gestion de versions offrent un ensemble de fonctionnalités pour gérer les modifications, les versions et les collaborations sur le code source. Ces fonctionnalités peuvent inclure :
-Stockage des versions du code source dans un référentiel central ou distribué.
-Suivi des modifications apportées au code, avec la possibilité de revenir à des versions antérieures.
-Gestion des branches pour permettre le développement parallèle et la fusion de code.
-Gestion des conflits lors de la fusion de modifications provenant de différentes branches ou de différents contributeurs.
-Suivi des auteurs, des dates et des messages de modification pour faciliter la collaboration et l'attribution de responsabilités.

3/REVUE DE LA LITTÉRATURE SUR LES STRUCTURES DE DONNÉES ET ALGORITHMES UTILISÉS

La gestion de versions de code repose sur des structures de données et des algorithmes spécifiques pour stocker, gérer et fusionner les modifications du code. Il existe différentes approches utilisées dans les outils de gestion de versions, telles que :
-Systèmes de fichiers : certains outils utilisent des systèmes de fichiers spécifiques pour stocker les versions du code, avec des opérations de copie, de suppression, etc.
-Graphes de révisions : d'autres outils utilisent des graphes de révisions pour représenter les versions et les relations entre elles, avec des opérations de fusion, de propagation, etc.
-Algorithmes de fusion : les outils de gestion de versions utilisent également des algorithmes de fusion pour combiner les modifications provenant de différentes branches ou contributeurs, avec des approches telles que la fusion à trois voies, la fusion automatique, etc.

4/PRÉSENTATION DESAVANTAGES, LIMITES ET BONNES PRATIQUES

Chaque outil de gestion de versions a ses avantages et ses limites. Par exemple, Git est largement reconnu pour sa flexibilité, sa performance et sa gestion avancée des branches, mais peut avoir une courbe d'apprentissage abrupte pour les nouveaux utilisateurs. SVN, quant à lui, peut-être plus simple à utiliser mais peut manquer certaines fonctionnalités avancées. Il est important de noter que le choix de l'outil de gestion de versions dépend des besoins spécifiques du projet et de l'équipe de développement.
Voici quelques bonnes pratiques à considérer lors de l'utilisation d'un outil de gestion de versions de code :
-Utiliser des branches pour permettre un développement parallèle et isoler les fonctionnalités ou les corrections de bugs.
-Effectuer des commits fréquents avec des messages de commit clairs pour garder une trace des
modifications et faciliter la collaboration.
-Utiliser des outils de comparaison et de fusion pour résoudre les conflits de manière appropriée et éviter la perte de code.
-Sauvegarder régulièrement le référentiel de code pour éviter la perte de données en cas de problème technique.
-Former les membres de l'équipe sur l'utilisation de l'outil de gestion de versions et établir des
conventions de nommage et de gestion des branches pour garantir une utilisation cohérente et efficace de l'outil.

En conclusion, la gestion de versions de code est une pratique essentielle dans le développement de logiciels, et il existe plusieurs outils de gestion de versions populaires sur le marché. Chaque outil a ses avantages, limites et bonnes pratiques, et le choix de l'outil approprié dépend des besoins spécifiques du projet et de l'équipe de développement. En suivant les bonnes pratiques de gestion de versions, il est possible de faciliter la collaboration, de gérer les modifications du code de manière efficace et de garantir l'intégrité et la disponibilité du code source.

CONCEPTION ET IMPLÉMENTATION DU LOGICIEL DE GESTION DES VERSIONS

1/Reformulation du sujet synthétique

Notre projet consiste à développer un outil de suivi et de versionnage de code similaire à Git, utilisé dans le développement logiciel. Nous mettons en place différentes structures de données, telles que WorkFile, WorkTree, kvp, HashTable et Commit, pour gérer les fichiers, les arbres de travail, les commits, les branches, les conflits, etc. Notre outil permettra de créer, supprimer, mettre à jour et fusionner des fichiers et des commits, ainsi que de gérer les branches et les conflits. Il sera capable de sauvegarder, restaurer et afficher les informations sur les commits, les arbres de travail et les branches. Tout au long du projet, nous nous assurons de respecter les bonnes pratiques de programmation, de gérer les erreurs et de garantir la robustesse et la fiabilité de notre outil.

2/Description des structures manipulées et de la structure globale du code:

- Les structures de données utilisées dans cet exemple sont conçues pour représenter un système de contrôle de version simplifié, similaire à Git. Elles comprennent plusieurs types de structures, notamment :
- La structure "Cell" qui représente un élément d'une liste chaînée. Elle contient un pointeur vers une chaîne de caractères (data) et un pointeur vers le prochain élément de la liste (next).
- La structure "List" qui représente une liste chaînée. Elle contient un pointeur vers le premier élément de la liste.
- La structure "WorkFile" qui représente un fichier ou un répertoire dans le répertoire de travail (work tree). Elle contient un pointeur vers le nom du fichier (name), un pointeur vers le hash du fichier (hash) et un entier représentant les autorisations associées au fichier (mode).
- La structure "WorkTree" qui représente le répertoire de travail (work tree). Elle contient un tableau de WorkFile (tab) représentant les fichiers du répertoire de travail, la taille du tableau (size) et le nombre de fichiers dans le répertoire de travail (n).

- La structure "kvp" qui représente une paire clé-valeur (key-value). Elle contient un pointeur vers une chaîne de caractères représentant la clé (key) et un pointeur vers une chaîne de caractères représentant la valeur (value).
- La structure "HashTable" qui représente une table de hachage. Elle contient un tableau de pointeurs vers des kvp (T), la taille du tableau (size) et le nombre d'éléments dans la table de hachage (n).
- La structure "Commit" qui représente un commit dans le système de contrôle de version. Elle est basée sur la structure HashTable et contient les mêmes éléments, à savoir un tableau de pointeurs vers des kvp (T), la taille du tableau (size) et le nombre d'éléments dans la table de hachage (n).
- Le code est organisé en cinq parties distinctes, chacune avec son propre objectif et fonctionnalité, et est orchestré par un point d'entrée principal (main) qui coordonne leur exécution.

Première partie :

La première partie du projet, qui se trouve dans le fichier file.c, a pour objectif principal d'écrire un programme permettant d'enregistrer un instantané. Elle comprend les éléments suivants : prise en main du langage Bash, implémentation d'une liste de chaînes de caractères et gestion de fichiers sous Git.

Deuxième partie

La deuxième partie du projet, qui se trouve dans le fichier workfile.c, se concentre sur la manipulation de fichiers et de répertoires dans une structure de données appelée WorkTree. Elle inclut la définition des fonctions de manipulation de base pour ces structures. Cette partie permet également d'enregistrer un instantané d'un WorkTree, en ajoutant une extension ".t" au nom du fichier d'enregistrement instantané pour le distinguer des fichiers classiques.

Troisième partie

La partie 3 dans "commit.c" concerne la gestion des commits pour organiser les enregistrements instantanés d'un projet. Les commits sont implémentés sous forme de tables de hachage contenant des clés et des valeurs, et doivent inclure des informations telles que l'auteur, le hash du WorkTree, le message et le hash du commit précédent. Les commits sont organisés en une liste simplement chaînée appelée branche, et des références sont utilisées pour pointer vers les commits, stockées dans un répertoire caché ".refs". Les références principales sont "HEAD", qui pointe vers le dernier commit, et permet de simuler des déplacements dans la chronologie du projet "master", qui représente la branche principale.

Quatrième partie :

La partie 4 du projet, implémentée dans le fichier "branche.c", concerne la gestion d'une timeline arborescente à partir de plusieurs branches. Dans la partie précédente, nous avons travaillé avec une seule branche, ce qui permettait de suivre l'évolution d'un projet de manière linéaire. L'avantage d'avoir plusieurs branches est de pouvoir considérer différentes versions d'un même code, ce qui permet à plusieurs personnes de travailler simultanément sur différentes parties/fonctionnalités d'un projet, de tester des versions alternatives du code, ou de réaliser des modifications sans affecter la branche principale.
Dans cette partie, nous allons mettre en place la gestion de plusieurs branches en utilisant un fichier caché appelé ".current branch" qui permettra de savoir à tout moment dans quelle branche nous nous situons. Ce fichier contiendra le nom de la branche courante. Avec plusieurs branches, il est important de pouvoir manipuler facilement une branche et de savoir comment naviguer entre les différentes branches de l'arborescence. C'est l'objectif de cette partie.
Le fichier "branche.c" contient donc les fonctions de base pour manipuler les branches, ainsi que la simulation de la commande "git checkout" pour permettre de changer de branche et naviguer entre les différentes branches de l'arborescence.

Cinquième partie

La partie 5 du projet "merge.c" implémente la gestion des fusions de branches dans un système de contrôle de version inspiré de Git. Elle permet de fusionner deux branches en créant un nouveau commit qui représente la fusion des contenus des derniers commits de ces deux branches. En cas de conflit, elle propose une méthode simple de gestion de conflits, en créant un commit de suppression pour supprimer les éléments conflictuels d'une des deux branches avant de réaliser la fusion sans conflit. La partie 5 met également en place la gestion des prédécesseurs pour le nouveau commit de fusion, en conservant les informations sur les deux branches fusionnées dans les métadonnées du commit.Le fichier header.h contient les définitions de structures de données, les constantes, les prototypes de fonctions et les macros pour un projet de gestion de version Git simplifié. Il définit les types de données et les opérations nécessaires pour la manipulation des fichiers, des commits et des branches.

MyGit :

MyGit est un outil de gestion de version simple en ligne de commande, développé en langage C, permettant à un utilisateur de suivre l'évolution d'un projet. Voici les principales fonctionnalités de MyGit :

- init: Cette commande initialise le répertoire de références de MyGit, en créant un fichier de références vide pour stocker les informations sur les commits et les branches.
- list-refs: Cette commande affiche toutes les références existantes dans le répertoire de références, telles que les noms de branches, les noms de tags, et autres références.
- create-ref <name> <hash>: Cette commande crée une nouvelle référence avec le nom <name> qui pointe vers le commit correspondant au hash <hash>. Cela permet de créer de nouvelles branches ou de créer des tags pour marquer des points spécifiques dans l'historique du projet.
- delete-ref <name>: Cette commande supprime la référence avec le nom <name>, ce qui peut être utilisé pour supprimer des branches ou des tags qui ne sont plus nécessaires.
- add <elem> [<elem2> <elem3> ...]: Cette commande ajoute un ou plusieurs fichiers ou répertoires à la zone de préparation de MyGit. Les fichiers ajoutés à la zone de préparation feront partie du prochain commit.
- list-add: Cette commande affiche le contenu de la zone de préparation, c'est-à-dire les fichiers et les répertoires qui ont été ajoutés avec la commande add.
- clear-add: Cette commande vide la zone de préparation, en enlevant tous les fichiers et les répertoires qui y ont été ajoutés.
- commit <branch name> [-m <message>]: Cette commande effectue un commit sur une branche spécifiée, avec un message descriptif optionnel. Un commit est une sauvegarde des modifications apportées au projet à un moment donné, et il est identifié par un hash unique qui représente son état.
- get-current-branch: Cette commande affiche le nom de la branche courante, c'est-à-dire la branche sur laquelle se trouve l'utilisateur actuellement.
- branch <branch-name>: Cette commande crée une nouvelle branche avec le nom <branch-name>, si elle n'existe pas déjà. Si la branche existe déjà, un message d'erreur est affiché.
- branch-print <branch-name>: Cette commande affiche le hash de tous les commits de la branche spécifiée, ainsi que leur message descriptif éventuel. Si la branche n'existe pas, un message d'erreur est affiché.
- checkout-branch <branch-name>: Cette commande effectue un déplacement sur la branche spécifiée, en changeant la branche courante de l'utilisateur. Si la branche n'existe pas, un message d'erreur est affiché.
- checkout-commit <pattern>: Cette commande effectue un déplacement sur le commit dont le hash commence par <pattern>. Si plusieurs commits correspondent à ce pattern, des messages d'erreur sont affichés. Cette commande permet de revenir à des états antérieurs du projet en se basant sur les commits.
- Merge <branch> <message> : Cette commande effectue la fusion de deux branches et deux cas peuvent se présenter : s'il n'y a aucun conflit entre les fichiers des deux branches, la branche courante fusionne avec la branche remote normalement sinon elle demande à l'utilisateur de choisir soit l'option de maintenir les fichiers de la branche courante soit de maintenir ceux de la branche remote soit de choisir manuellement la branche dans laquelle il veut maintenir le fichier.

3/Description des fonctions principales et des choix d’implémentation

blobFile

La fonction blobFile effectue les étapes suivantes :
Elle appelle la fonction sha256file pour générer le hash SHA-256 du fichier d'entrée file et stocke le hash dans la variable hash. Elle alloue de la mémoire pour une copie du hash dans la variable ch2 à l'aide de la fonction strdup. Elle modifie ensuite le troisième caractère du hash en '\0', pour obtenir un sous-ensemble de deux caractères du hash, qui sera utilisé pour créer un répertoire unique pour stocker le fichier. Elle appelle la fonction fileExists pour vérifier si le répertoire correspondant au sous-ensemble de deux caractères du hash existe déjà. Si le répertoire n'existe pas, elle crée le répertoire en utilisant mkdir avec le sous-ensemble de deux caractères du hash comme nom de répertoire.
Elle appelle la fonction hashToPath pour générer le chemin de fichier complet à partir du hash complet dans la variable ch. Elle appelle la fonction cp pour copier le fichier d'entrée file vers le chemin de fichier généré à partir du hash.
Elle libère la mémoire allouée pour les variables hash, ch2 et ch à l'aide de la fonction free.
Choix d'implémentation :
Utilisation de sha256file pour générer le hash SHA-256 du fichier d'entrée, permettant d'obtenir une empreinte unique et sécurisée du contenu du fichier. Utilisation de strdup pour allouer de la mémoire et créer une copie du hash, afin de manipuler une chaîne de caractères sans modifier l'original. Utilisation de fileExists pour vérifier si le répertoire correspondant au sous-ensemble de deux caractères du hash existe déjà, avant de créer le répertoire avec mkdir, pour éviter la création de répertoires redondants. Utilisation de hashToPath pour générer le chemin de fichier complet à partir du hash, permettant de créer un chemin de fichier unique basé sur le hash SHA-256 du fichier. Utilisation de cp pour copier le fichier d'entrée vers le chemin de fichier généré à partir du hash, permettant de stocker le fichier dans le répertoire correspondant au sous-ensemble de deux caractères du hash. Allocation dynamique de mémoire pour les variables hash, ch2 et ch à l'aide de malloc et strdup, permettant de gérer dynamiquement la taille des chaînes de caractères en fonction de la taille du hash et du sous-ensemble de deux caractères du hash. Il est important de penser à libérer la mémoire allouée avec free lorsque celle-ci n'est plus nécessaire pour éviter les fuites de mémoire.

blobWorkTree

La fonction prend en entrée un pointeur vers une structure WorkTree appelée wt. Elle crée un fichier temporaire sécurisé dans le répertoire "/tmp/" en utilisant un modèle de nom de fichier avec des caractères de remplacement ("XXXXXX") à l'aide de mkstemp, et stocke le nom de fichier généré dans un tableau de caractères fname. Elle appelle la fonction wttf pour écrire le contenu de la structure wt dans le fichier temporaire. Elle appelle la fonction sha256file pour générer le hash SHA-256 du fichier temporaire. Elle alloue de la mémoire pour un tampon de chaîne de caractères ch qui contiendra le hash SHA-256 généré, et copie le hash dans le tampon. Elle ajoute l'extension ".t" au hash dans le tampon ch à l'aide de strcat. Elle appelle la fonction cp pour copier le contenu du fichier temporaire dans un nouveau fichier portant le nom généré à partir du hash SHA-256 dans ch. Elle appelle la fonction setMode pour définir les permissions du nouveau fichier à 0777. Elle libère la mémoire allouée pour le tampon ch à l'aide de free. Elle retourne le hash SHA-256 généré.
Choix d'implémentation :
Utilisation d'un fichier temporaire sécurisé pour stocker le contenu de la structure wt à l'aide de mkstemp, garantissant ainsi la sécurité et la fiabilité de l'opération. Utilisation de wttf pour écrire le contenu de la structure wt dans le fichier temporaire. Utilisation de sha256file pour générer le hash SHA-256 du fichier temporaire, permettant d'obtenir une empreinte unique et sécurisée du contenu du fichier. Utilisation de strcat pour ajouter l'extension ".t" au hash dans le tampon ch, afin de créer un nom de fichier unique pour le nouveau fichier à partir du hash SHA-256 généré. Utilisation de cp pour copier le contenu du fichier temporaire dans un nouveau fichier portant le nom généré à partir du hash SHA-256 dans ch, permettant de créer une copie du fichier temporaire avec un nom unique basé sur son contenu. Utilisation de setMode pour définir les permissions du nouveau fichier à 0777, ce qui permet d'accorder les droits de lecture, écriture et exécution à tous les utilisateurs. Allocation dynamique de mémoire pour le tampon ch à l'aide de malloc, permettant de gérer dynamiquement la taille du tampon en fonction de la taille du hash SHA-256 généré. Il est important de penser à libérer la mémoire allouée avec free lorsque celle-ci n'est plus nécessaire pour éviter les fuites de mémoire.

saveWorkTree

La fonction saveWorkTree est utilisée pour générer un blob représentant l'état actuel d'un arbre de travail (WorkTree) à partir d'un chemin d'accès (path). Un arbre de travail est une structure de données qui représente l'état des fichiers et des répertoires d'un système de gestion de version dans un certain répertoire. Voici une description de la fonction et de son choix d'implémentation :
La fonction prend en entrée un pointeur vers un WorkTree (wt) et un chemin d'accès (path), qui spécifie le répertoire de travail dont l'état doit être sauvegardé. La fonction parcourt les éléments (WorkFile) du tableau wt->tab de wt à l'aide d'une boucle for. Pour chaque WorkFile (wf), la fonction construit le chemin d'accès absolu (absPath) en concaténant path et le nom de fichier (wf->name) à l'aide de la fonction concat_paths. La fonction utilise la fonction isDirectory pour vérifier si absPath représente un fichier (isDirectory(absPath) == 0) ou un répertoire (isDirectory(absPath) != 0). Si absPath représente un fichier, la fonction appelle les fonctions blobFile, sha256file et getChmod pour obtenir le hash, le mode de fichier et le mode de fichier en octal du fichier correspondant, respectivement. Elle met à jour les champs hash et mode de wf avec les valeurs obtenues. Si absPath représente un répertoire, la fonction crée un nouvel objet WorkTree (wt2) à l'aide de la fonction initWorkTree, et utilise la fonction listdir pour obtenir la liste des fichiers et répertoires sous absPath. Elle itère ensuite sur cette liste à l'aide d'une boucle for et appelle appendWorkTree pour ajouter les fichiers et répertoires à wt2, en ignorant ceux dont le nom commence par un point (pour exclure les fichiers cachés dans les systèmes de fichiers UNIX). La fonction appelle récursivement saveWorkTree pour wt2 avec absPath comme nouvel argument de chemin d'accès, pour obtenir le hash du répertoire représenté par wt2, et met à jour le champ hash de wf avec cette valeur. La fonction appelle freeWorkTree pour libérer la mémoire allouée pour wt2. Elle met à jour le champ mode de wf avec le mode de fichier du répertoire correspondant, en appelant getChmod sur absPath. Elle libère la mémoire allouée pour absPath à l'aide de free. Elle répète les étapes ci-dessus pour tous les éléments de wt->tab. Enfin, la fonction appelle la fonction blobWorkTree pour générer un blob représentant les données de wt, et renvoie un pointeur vers cette chaîne de caractères. Le choix d'implémentation de cette fonction utilise la récursivité pour traiter les répertoires et leurs contenus de manière efficace. Elle utilise également des fonctions auxiliaires (isDirectory, `blobFile, sha256file, getChmod, initWorkTree, listdir, appendWorkTree, freeWorkTreeetblobWorkTree) pour effectuer les opérations nécessaires sur les fichiers et les répertoires, ce qui permet de bien organiser le code et de le rendre plus lisible et maintenable. De plus, elle utilise des pointeurs pour manipuler les structures de données (WorkTree, WorkFile, List, Cell`) afin d'économiser la mémoire et d'optimiser les performances. La fonction affiche également des informations de débogage à l'aide de printf pour faciliter le suivi du processus d'exécution et le débogage en cas d'erreur. En somme, la fonction saveWorkTree est conçue pour sauvegarder l'état d'un arbre de travail dans un blob, en traitant efficacement les fichiers et les répertoires, en utilisant des fonctions auxiliaires et en fournissant des informations de débogage.
La fonction blobCommit s'implémente presque de la même manière que blobFile.

myGitAdd

La fonction myGitAdd permet d'ajouter un répertoire ou un fichier dans la zone de préparation du Git. Pour son implémentation, elle teste d'abord si le fichier « .add » existe sinon elle la crée avec l'appel système touch. Elle utilise ftwts pour lire le worktree dans le fichier « add » et si le fichier passé en paramètre n'existe pas la fonction s'arrête sinon elle ajoute son worktree associé dans le add du Git avec la fonction appendWorkTree et wttf. Ensuite, elle libère la mémoire.

MyGitCommit

permet de créer un point de sauvegarde du commit dans le Git. Pour cela, si la référence existe avec la fonction fileExists. Ensuite, elle récupère le hash de la branche HEAD et de la branche passée en paramètre, si ce ne sont pas les mêmes branches la fonction s'arrête sinon elle lit les workTree dans la zone de préparation en les sauvegardant tout en vidant le add. Elle créé le commit associé et le prédecesseur, enfin elle enregistre le commit avec blobCommit et met à jour les références HEAD et branche du paramètre qui vont pointer vers le nouveau commit.

MyGitCheckoutBranch

MyGitCheckoutBranch est une fonction qui permet de changer de branche. Elle écrit tout d'abord la branche à laquelle on veut accéder dans le fichier « .current-branch » ensuite elle fait pointer HEAD vers le hash de cette branche et restaure le commit associé en sauvegardant son workTree par la fontion restoreCommit.

MyGitcheckoutCommit

permet de retourner sur des anciennes versions de fichier grâce à leurs commits. Pour l'implémentation, elle récupère d'abord tous les commits des branches et filtre la liste obtenue par filterList et ensuite si la liste est vide elle renvoie un message d'erreur sinon si le commit est seul dans la liste, elle fait pointer la branche HEAD sur le commit et restaure le commit.

mergeWorkTrees

La fonction mergeWorkTrees réalise la fusion de deux arbres de travail (WorkTree) représentant deux états différents d'un même projet. Elle compare les fichiers présents dans les deux arbres de travail et crée un nouvel arbre de travail (fusion) qui contient les fichiers des deux arbres d'origine, en prenant en compte les différences entre les deux. L'implémentation choisie utilise une approche itérative pour parcourir les fichiers des deux arbres de travail. Pour chaque fichier dans wt1, elle vérifie s'il existe dans wt2. Si c'est le cas, elle compare les hashs des fichiers pour déterminer s'ils sont identiques. Si les hashs sont identiques, le fichier de wt1 est ajouté à fusion. Si les hashs sont différents, cela signifie qu'il y a un conflit entre les deux fichiers, et le nom du fichier est ajouté à une liste chaînée (conflicts) pour un traitement ultérieur. De même, pour chaque fichier dans wt2, la fonction vérifie s'il existe dans wt1, et si ce n'est pas le cas, le fichier de wt2 est ajouté à fusion. L'implémentation utilise également des fonctions d'insertion et de recherche dans une liste chaînée pour stocker et gérer les conflits détectés. Les fichiers en conflit sont ajoutés à la liste conflicts uniquement s'ils ne sont pas déjà présents dans cette liste, pour éviter les doublons. En résumé, la fonction mergeWorkTrees effectue une fusion des fichiers de deux arbres de travail en tenant compte des différences entre les deux arbres. Elle détecte les conflits entre les fichiers et les stocke dans une liste pour un traitement ultérieur, afin de permettre une gestion efficace des conflits lors d'une opération de fusion de branches ou de réconciliation de versions d'un projet.

merge

La fonction "merge" implémente la fusion de deux branches dans un système de gestion de versions. Elle prend en entrée les noms des deux branches à fusionner ainsi qu'un message de commit pour la nouvelle branche fusionnée. L'implémentation de la fonction commence par récupérer les informations sur les deux branches à fusionner, notamment les commits et les worktrees associés à chacune d'entre elles. Elle crée ensuite une nouvelle worktree vide qui servira de base pour la fusion des deux branches.
La fonction procède ensuite à la fusion des fichiers des deux worktrees d'origine en comparant les contenus des fichiers et en les combinant de manière appropriée pour générer les fichiers fusionnés dans la nouvelle worktree. Si des conflits sont détectés, c'est-à-dire si les deux branches ont apporté des modifications contradictoires au même fichier, les conflits sont enregistrés dans une liste. Si aucune erreur de fusion n'est détectée, la nouvelle worktree fusionnée est sauvegardée dans le système de gestion de versions en créant un nouveau commit. Les prédecesseurs de ce nouveau commit sont définis comme étant les commits des deux branches d'origine, pour conserver l'historique des deux branches fusionnées. Enfin, les références de branches sont mises à jour pour pointer sur le nouveau commit, indiquant ainsi que la fusion a été effectuée avec succès. Si des conflits sont détectés lors de la fusion, la liste des conflits est retournée à la fin de la fonction pour que l'utilisateur puisse les résoudre manuellement. L'implémentation utilise des structures de données telles que les commits, les worktrees et les listes pour gérer les différentes étapes de la fusion. Elle fait également usage de fonctions auxiliaires pour la manipulation des fichiers, la gestion des commits et des références de branches. Des allocations dynamiques de mémoire sont effectuées pour stocker les chemins des fichiers, les hashs associés aux commits et aux worktrees, ainsi que la liste des conflits. La mémoire allouée est libérée à la fin de la fonction pour éviter les fuites de mémoire.

createDeletionCommit

Voici une description de La fonction "createDeletionCommit" et de son choix d'implémentation :
La fonction prend en entrée trois paramètres : le nom de la branche à laquelle appliquer la suppression ("branch"), une liste de conflits à prendre en compte lors de la création du commit de suppression ("conflicts"), et le message de commit à utiliser ("message"). La fonction commence par récupérer la branche courante en appelant la fonction "getCurrentBranch()" et en stockant le résultat dans une variable "current_branch". Ensuite, elle utilise une fonction "myGitCheckoutBranch(branch)" pour se déplacer sur la branche spécifiée en paramètre, en effectuant un "checkout" de la branche avec le nom "branch". Cela peut être un choix d'implémentation fictif qui simule un déplacement dans une branche spécifique. Elle utilise également une fonction "getRef(branch)" pour récupérer le hash du dernier commit de la branche spécifiée, et stocke ce hash dans une variable "hBranch". La fonction alloue de la mémoire pour les chaînes de caractères "hPathC" et "pathC", qui seront utilisées pour construire le chemin du fichier de commit et du fichier de travail associés à la branche. Elle utilise ensuite ces chaînes de caractères pour construire les chemins complets du fichier de commit ("hPathC.c") et du fichier de travail ("ht.t") en utilisant la fonction "hashToPath()" pour convertir les hashes en chemins de fichiers. La fonction utilise les fonctions "ftc(hPathC)" et "ftwts(ht)" pour récupérer les structures de données "Commit" et "WorkTree" associées à la branche. Elle supprime la zone de préparation en appelant la fonction "deleteRef(".add")", ce qui peut être une façon fictive de supprimer une zone de préparation dans le SGV. La fonction itère ensuite sur les fichiers du "WorkTree" récupéré, en vérifiant si le nom de fichier figure dans la liste des conflits spécifiée en paramètre ("conflicts"), et si c'est le cas, elle appelle la fonction "myGitAdd(nom_fichier)" pour ajouter le fichier dans la zone de suppression. Ceci peut être un choix d'implémentation fictif qui simule l'ajout des fichiers de conflit dans la zone de suppression. Elle crée finalement le commit de suppression en appelant la fonction "myGitCommit(branch, message)" en passant le nom de la branche et le message de commit spécifiés en paramètres.Elle revient ensuite à la branche courante en appelant la fonction "myGitCheckoutBranch(current_branch)", ce qui peut être un choix d'implémentation fictif pour revenir à la branche courante après la création du commit de suppression. Enfin, elle libère la mémoire allouée pour les variables "current_branch", "hBranch", "hPathC", "pathC", la structure "Commit" "c" et la structure "WorkTree" "wt" en appelant les fonctions "free()" appropriées pour éviter les fuites de mémoire.

CONCLUSION

En récapitulation, le projet de gestion de version Git simplifié a abouti à la création d'un logiciel fonctionnel qui offre des fonctionnalités de base pour gérer les fichiers, les commits et les branches. Les principaux résultats obtenus sont la mise en place d'une structure de données appropriée, l'implémentation des opérations de base de Git (commit, checkout, merge, etc.) et la création d'une interface utilisateur minimaliste.
Perspectives d'amélioration et de développement futurs :
Pour améliorer le logiciel, plusieurs perspectives sont envisageables, telles que l'ajout de fonctionnalités avancées de Git (rebase, cherry-Pick, etc.), l'optimisation des performances pour gérer de gros dépôts, la mise en place d'un système de gestion des conflits plus sophistiqué, et l'amélioration de l'interface utilisateur pour la rendre plus conviviale et intuitive.
Retour d'expérience et leçons apprises :
Ce projet a été une expérience enrichissante qui a permis d'apprendre beaucoup sur les concepts de gestion de version, les structures de données, la manipulation des fichiers, ainsi que la coordination et la collaboration au sein d'une équipe de développement. Les principales leçons apprises sont l'importance d'une planification et d'une gestion efficace du projet, la nécessité d'une documentation claire et précise, ainsi que l'importance de la communication et de la collaboration entre les membres de l'équipe pour garantir le succès du projet.

RÉFÉRENCES BIBLIOGRAPHIQUES

projet-version-3avril.pdf