Rapport de projet SGF

Ce rapport de projet a pour but de synthétiser tout le travail effectué lors du projet s'inscrivant dans le cadre de l'unité d'enseignement « Méthodologie de la programmation ». Ce document retracera les grandes étapes du projet, les choix effectués et leurs motifs. Il traitera également des problèmes rencontrés et comment ils ont été surmontés, de l'architecture globale du projet et de la réflexion derrière cette architecture. Enfin, un bilan de ce projet sera établi à la fois sur le plan technique que personnel, le tout dans le but de tirer des conclusions sur les points forts et les points faible de la réalisation de ce projet.

Introduction

L'objectif principale de ce projet était la création d'un SGF (Système de Gestion de Fichier) utilisable en ligne de commande (CLI). Le projet imposait l'utilisation d'une structure de donnée de type « arbre » sur laquelle notre SGF allait reposer. Le langage imposé pour la réalisation de ce projet était l'Ada. Ce projet proposait également des objectifs secondaires comme l'implémentation d'un menu utilisable en plus de l'interpréteur de commande ainsi que la gestion de la mémoire occupée par les différents fichiers et répertoires.

Nous allons dans un premier temps voir toutes la modélisation et la conception autour de ce projet (type de données, algorithmes, raffinages et modules). Dans un second temps, nous détaillerons et justifierons des choix réalisés lors du développement ainsi que des difficultés rencontrées. Enfin, nous dresserons un bilan d'une part technique et d'autre part personnel sur la réalisation de ce projet.

Table des matières

Modélisation et conception

Les types de données

La première chose à déterminer pour réaliser ce projet était la structure des types de données qui seront utilisés au sein du SGF. J'ai tout d'abord fait le choix de faire un type unique pour les fichiers et dossiers. En effet un dossier et relativement similaire à un fichier, il possède seulement quelques spécificités mais dans l'ensemble c'est la même chose. Il me paraissait donc plus judicieux de créer un seul et même enregistrement pour les « feuilles » de mon arbre plutôt que d'introduire de la redondance et de la complexité en créant deux types à part entière.

J'ai alors créé un enregistrement nommé « T_Fichier » qui avait pour éléments : un nom, une taille, des permissions, un contenu, un parent, un frère et un fils. Le parent, le frère et le fils ne sont rien d'autre que des pointeurs vers « T_Fichier » que j'ai défini avec un type nommé « T_Arbre ». Le nom et le contenu sont des « Unbounded_String » qui ont l'avantage d'avoir une taille plutôt souple. La taille est un entier et les permissions sont un tableau de 10 caractères. Pour les permission, je me suis inspiré du système de droit utilisé par Linux, le première caractère correspond à la nature de l'élément (« d » pour un dossier « - » pour un fichier). Les neuf autres correspondent respectivement aux droits en lecture écriture et exécution de l'utilisateur, du groupe et des autres. J'ai créé un type nommé « T_Tab_Perm » pour les permissions.

```
type T_Fichier is
    record

        Nom : Unbounded_String; -- Nom du fichier/dossier
        Taille : Integer;           -- Taille du fichier/dossier
        Permission : T_Tab_Perm;    -- Permissions du fichier/dossier

        Parent : T_Arbre;    -- Pointeur sur le parent
        Frere : T_Arbre;     -- Pointeur sur le prochain frère
        Fils : T_Arbre;      -- Pointeur sur le fils
        Contenu : Unbounded_String;    -- Contenu du fichier

    end record;
```

Figure 1 – Définition du type T_Fichier

Maintenant que j'ai défini les feuilles de mon arbre, je dois commencer à définir et écrire les premiers algorithmes qui me permettrons de construire mon arbre.

Les algorithmes

Il me fallait avant tout une première fonction qui allait créer un fichier ou un dossier. J'ai donc défini la procédure « Ajouter » qui prend en paramètre un pointeur sur fichier (T_Arbre), un nom, un booléen pour différencier la création de fichier et de dossier ainsi qu'un autre

pointeur sur fichier (T_Arbre) pour correctement construire l'arbre. À l'intérieur de cet algorithme je distingue deux cas différents. Le cas où je crée un fichier et le cas où je crée un dossier. Pour le premier j'initialise les permissions comme celle d'un fichier ainsi que la taille à 0. Dans le second j'initialise les permissions comme celle d'un dossier ainsi qu'une taille à 10240 comme le sujet le demandait. Dans tous les cas, le fils, le frère et le contenu sont initialisé à « null ».

Une fois fait, je devais tester ma procédure, pour cela j'ai écrit une seconde procédure qui prenait en paramètre un pointeur sur fichier et qui affichait simplement le nom du T_Fichier. Tout fonctionnait bien mais ce n'était pas très pratique de vérifier chaque élément individuellement chaque élément du dossier. J'ai alors écrit une procédure récursive qui affichait le nom d'un T_Fichier ainsi que celui de tous ses frères. Ça, couplé à une procédure nommée « Afficher_dos » qui affiche tous les éléments d'un fichier en utilisant la procédure précédemment créée et j'avais un début de « ls ».

J'ai ensuite fait de même pour pouvoir créer un « T_Fichier » en spécifiant en paramètre seulement un dossier peu importe le nombre d'éléments qu'il possède. J'ai nommé cette procédure « Ajouter_Dand_Dos ». Tout cela m'a permis de facilement créer et lister des dossiers/fichiers.

J'ai également pressenti le besoin de savoir si un « T_Fichier » était un dossier ou un fichier ce qui m'a poussé à écrire la fonction « Est_Dossier » qui prend en paramètre un « T_Arbre » (pointeur sur T_Fichier) et renvoie un booléen. Vrai si le premier caractère des permissions est « d », faux sinon. Simple mais efficace et surtout très parlant, on comprend très bien ce que la fonction fait.

À ce stade je n'avais plus vraiment d'idée d'algorithme à implémenter pour l'arbre, j'ai donc décidé de commencer à implémenter des commandes pour avoir un meilleur aperçu des prochains algorithmes à implanter. Cependant pour exécuter une commande il faut dans un premier temps la découper pour avoir d'un côté la commande en tant que telle et d'un autre les arguments. Pour réaliser cela je me suis aidé du paquage générique « P_Liste_Gen » que j'avais développé lors des TP. J'ai instancié ce paquage avec comme élément une « Unbounded_String ». J'avais donc à ma disposition une liste de « Unbounded_String » qui allait m'être utile pour la découpe des commandes.

La fonction pour le traitement des commandes s'appelle découpage, elle prend en paramètre une « String » ainsi qu'un caractère et renvoie une liste chainée de « Unbounded_String ». Le découpage se fait en fonction du caractère passé en paramètre afin de rendre la fonction plus générique.

Le découpage s'opère de la manière suivante :

J'initialise deux entiers à zéro, je parcours chaque caractère de la « String » à découper. Dès que je tombe sur le caractère passé en paramètre, je passe le second entier à la valeur de parcours de la chaine. Je peux alors découper le premier « mot » de la chaine en fonction du premier et du second entier et l'insérer en queue de ma liste chainée. Je passe le premier

entier à la valeur du second et je recommence le processus jusqu'à atteindre la fin de la « String ».

```
function Decoupage (F_Chaine : in String ; F_Cible : in Character) return P_Liste_Ustring.T_Liste_Chainee is
    -- Copie de la chaine de caractère passée en paramètre + ajout du caractère de découpage pour être sur de tout récupérer
    Commande : String := F_Chaine & F_Cible;
    Element : Unbounded_String; -- Chaque élément de la chaine à découpé
    Debut, Fin : Integer := 0; -- Debut et fin des éléments à découpé
    Resultat : P_Liste_Ustring.T_Liste_Chainee ; -- Pointeur sur T_Cellule d'une liste chainée d'unbounded string

begin

    Debut := Commande'First - 1;
    Resultat := Creer_Liste_Vide;
    for I in Commande'Range loop
        if Commande(I) = F_Cible then
            Fin := I;
            Element := To_Unbounded_String ((Commande (Debut + 1 .. Fin - 1)));
            Inserer_En_Queue (Resultat, Element);
            Debut := Fin;
        end if;
    end loop;

    return Resultat;

end Decoupage;
```

Figure 2 - Définition de la fonction Decoupage

Le type « P_Liste_Chainee » étant privé j'ai dû créer une procédure pour tester la fonction découpage. Tout marchait bien, je pouvais commencer à implémenter des commandes tel que demandé dans l'énoncé du sujet.

J'ai commencé par implémenter les commandes les plus simples tel que « ls », « mkdir » et « touch » qui reposait essentiellement sur des fonctions et procédure déjà implémenter comme « Afficher_Dos » ou « Ajouter_Dans_Dos ». Cependant, lors de la définition de ces commandes, j'ai fait attention à ne pas créer d'erreur. En effet les procédures et fonction de l'arbre fonctionnent de manière offensive, c'est-à-dire qu'elles font confiance à l'appelant et ne vérifie pas le pré et post conditions. C'est pour cela que les préconditions et postconditions sont testées au sein des algorithmes des commandes. Il n'est pas possible de faire confiance à l'utilisateur, il faut donc prendre de soin de vérifier que tout est bon avant d'appeler les procédures de l'arbre ou lever des exception le cas échéant. Typiquement, la procédure « P_Ls » vérifie que le fichier n'existe pas déjà.

```
procedure P_Ls (P_Dossier : in T_Arbre) is
begin
    if P_Dossier /= null then

        if P_Dossier.all.Fils /= null then
            Afficher_Dos (P_Dossier);

        else
            raise Dos_Vide_Erreur;

        end if;

    else
        raise Fichier_Non_Trouve_Erreur;

    end if;

end P_ls;
```

Figure 3 – Implémentation de la procédure P_Ls

Afin de pouvoir tester ces premières commandes en direct et sans passer par des programmes de tests j'ai décidé l'implémenter l'algorithme principale qui allait être l'interface entre l'utilisateur et la machine, l'interface en ligne de commande. *Plus de détail sur cette partie dans la partie raffinage.* Je pouvais alors tester mes commandes dans une interface en ligne de commandes. Il était alors temps de s'attaquer aux commandes plus complexes tel que « cd », « cp », « mv ».

Pour faire simple, j'ai décidé d'implémenter la commande « cd » de la sorte à ce qu'on puisse faire uniquement un déplacement d'un saut dans l'arbre (soit monter, soit descendre). Cela m'a donc fait écrire deux fonctions, l'une monter et l'autre descendre. La fonction monter était plutôt simple à implanter car chaque fichier, à l'exception de la racine, possède un parent. Je n'avais alors qu'à renvoyer un pointeur sur le parent si celui-ci n'était pas « null ». Cependant pour la fonction descendre, ça n'a pas été aussi simple. Je me suis d'abord dit qu'il fallait que je vérifie si le dossier où l'utilisateur veut se déplacer existe bien. Pour cela j'ai défini une fonction nommée « Existe_Fils » qui vérifie l'existence d'un élément dans un dossier. J'ai donc pu terminer ma fonction descendre qui renvoie un pointeur sur « T_Fichier » si le fichier/dossier est trouvé et renvoie « null » sinon.

```
function Descendre (F_Courant : in T_Arbre ; F_Fils : in String) return T_Arbre is
    -- Il faut vérifier que c'est bien un dossier
    Dest : T_Arbre := F_Courant;
begin
    if Existe_Fils (Dest.all.Fils, F_Fils) then
        Dest := Dest.all.Fils;

        while Dest.all.Nom /= F_Fils loop
            Dest := Dest.all.Frere;
        end loop;
    else
        Dest := null;

    end if;
    return Dest;

end Descendre;
```

Figure 4 – Définition de la fonction Descendre

Il ne me restait plus qu'à implémenter ma fonction « cd » en utilisant la fonction descendre et en vérifiant qu'il s'agissait bien d'un dossier afin d'empêcher l'utilisateur de se déplacer dans un fichier. J'ai également créé un cas particulier dans le cas où l'utilisateur souhaite monter avec l'argument « .. ». Dans ce cas c'est la fonction « monter » qui est appelée avec une exception levée lorsque l'utilisateur se trouve à la racine.

```
procedure P_Cd (P_Courant : in out T_Arbre ; P_Chemin : in String) is
    Dest : T_Arbre := null;
begin
    -- Cas particulier :
    if P_Chemin = ".." then
        Dest := Monter (P_Courant);
        if Dest /= null then
            P_Courant := Dest;
        else
            raise Racine_Atteinte_Erreur;
        end if;

    -- Cas général :
    else
        Dest := Descendre (P_Courant, P_Chemin);
        if Dest /= null then

            if Est_Dossier (Dest) then
                P_Courant := Dest;
            else
                raise Pas_Un_Dossier_Erreur;
            end if;

        else
            raise Fichier_Non_Trouve_Erreur;
        end if;
    end if;


end P_Cd;
```

Figure 5 - Définition de la procédure P_Cd

J'ai poursuivi en définissant d'autres commandes comme « vi », « cat », « tar », « mv », « cp » en essayant de rester le plus simple possible afin de ne pas avoir d'algorithme chaotique.

Le raffinage

Dans l'optique de répondre au problème principal posé par le sujet, j'ai écrit un raffinage en choisissant comme R0 : « *Permettre à l'utilisateur d'utiliser un SGF avec des lignes de commande.* ». J'ai raffiné ce R0 en 3 étapes. Premièrement déclarer exhaustivement les commandes définies et utilisables. Puis initialiser l'arbre en créant la racine. Et enfin laisser la main à l'utilisateur tant qu'il ne décide pas de quitter. Les deux premières étapes se règlent en une ligne de code. Pour la troisième j'ai choisi de faire une boucle « tant que » avec comme condition « quitter = vrai ».

L'interaction avec l'utilisateur se divise en deux étapes. Premièrement il faut demander et récupérer la commande de l'utilisateur puis il faut la traiter. Pour récupérer une chaine de caractère, un « Get_Line » suffit. Pour traiter la commande de l'utilisateur, un opérateur « case » me paraissait bien adapté à la situation. J'ai donc récupéré le premier argument de la commande avec la fonction « Recup_Arg » que j'avais défini au préalable. Et en fonction de ce premier argument (« ls », « cd », « pwd », etc.) j'allais exécuter la fonction associée à chaque commande. Le tout en prévoyant bien sur un cas ou l'utilisateur tape une commande qui n'existe pas.

Les modules

Mon programme principal, nommé « main_sgf.adb », utilise trois modules. « P_Arbre » qui contient toutes les commandes de l'arbre (« ajouter », « supprimer », « monter », « descendre », « copier », « renommer » etc.). « P_Commande » qui contient la définition de toutes le commandes (« ls », « cd », « pwd », « vi », « cp », « cat », « touch », « mkdir » etc). Ces fonctions et procédures font appel à des procédures de « P_Arbre ». Enfin le module « P_Liste_Gen » qui permet de contenir le découpage des commandes.

Déroulement de la réalisation du projet

Mes choix

Durant la réalisation du projet j'ai été amené à faire des choix. J'ai, de manière générale, essayé de privilégier la simplicité afin de ne pas me sentir perdu dans mon propre projet. Ce module se nomme « Méthodologie de la programmation », je devais donc programmer avec méthodisme.

Mon premier choix a été de définir qu'un seul « fils » par dossier. Au début je pensais qu'il serait plus simple de définir un tableau de pointeur sur « T_Fichier » pour chaque dossier mais le problème d'un tableau et que sa taille est fixe. Cela impliquait donc de l'initialiser complètement à « null » à chaque création de fichier ainsi que de gérer la création de la suppression de fichier en fonction de l'état du tableau. Ceci ne me paraissait finalement tout sauf simple et j'ai donc décider d'utiliser un pointeur vers un unique fils et un pointeur vers un frère.

J'ai également fait le choix de ne pas traiter de chemin absolu ou relatif. En effet, par manque de temps sur le projet, j'ai préféré implémenter toutes les commandes plutôt que de passer du temps à traiter les chemins absolu et relatif dans le risque de ne pas finir le projet ou de rendre quelque chose de mal fait. Je suis conscient que ce que j'ai fait n'est pas exactement ce qui est attendu. Mais ce que j'ai fait, je l'ai bien fait. Chaque fonction et chaque procédure sont claires et concises. Mes fonctions de sont pas un méli-mélo de « while », de « if » et de « elsif ». Tout cela a pour gros avantage de pouvoir facilement et simplement éditer une fonction ou procédure dans prendre le risque de détruire toute l'architecture du programme. Il sera toujours temps l'implémenter le traitement des chemins si le besoin s'en fait sentir.

Les tests

Sous les conseils de nos professeurs lors des TP, j'ai choisi de tester mes programmes avec le module « pragma assertion ». Pour chaque paquage j'ai alors créé un fichier nommé « test_ » suivi du nom du paquage. J'ai testé chaque fonction d'un paquage en effectuant d'abord une opération simple et en vérifiant avec des assertions le bon déroulé de la fonction puis avec un cas un peu plus complexe. Certaines fonctions peuvent difficilement être testées car elle ne renvoie rien mais font seulement un affichage. « Pragma assert » ne peut alors pas vérifier leur bonne exécution.

Les difficultés

La principale difficulté que j'ai rencontré lors de ce projet aura été de ne pas savoir par où commencer. En effet ce projet me paraissait assez complexe et je ne savais pas par quel côté l'attaquer. À chaque fois que je voulais commencer, je me heurtais à trop de cas particuliers à traiter. J'ai alors décider de voir le projet sous un autre angle. Mon erreur était de vouloir tout traiter d'un coup ce qui n'est clairement pas possible. Je devais me fixer des micros objectives atteignable et petit à petit atteindre ce que le projet demande quitte à simplifier certaines exigences par moment. Cela m'a permis de réellement avancer dans le projet mais a eu pour

effet de bord de me faire prendre du retard et cela m'a obligé à simplifier certaines fonction pour terminer à temps.

Cependant, une fois lancé, je ne me suis pas heurté à de grosses difficultés. Le fait d'avoir des fonctions qui font une seule chose mais qui le font bien m'a permis d'avancer rapidement et de rattraper en partie mon retard. J'ai été satisfait des choix fait précédemment concernant le type unique « T_Fichier » ainsi que l'architecture par « frère » qui m'ont, in fine, simplifié la tâche.

Bilan du projet

Côté technique

Mon programme simule un SGF s'appuyant sur un arbre. Il propose un mini interpréteur de commande qui implémente la quasi-totalité des commandes demandées. Il manque cependant la partie récursive de certaine commande tel que le « ls -r », le « cp -r » ou bien le « rm -r ». L'implémentation d'une suppression récursive d'un dossier ne demanderait pas beaucoup de travail car la procédure de suppression existe déjà. Cependant, il faudrait repenser le système d'exécution des commandes afin qu'une même commande puisse prendre un nombre variable d'arguments (ex : « ls » et « ls -r »).

De plus, il pourrait être bien de traiter des chemins relatifs et absolus afin de pouvoir augmenter la portée des commandes. Actuellement les commandes ne fonctionnent que dans le dossier courant. Cette implémentation risque de demander un peu de travail mais une fois faite, elle pourra être implanter facilement grâce à la simplicité et à la modularité des algorithmes.

Enfin, une autre évolution serait d'implémenter une gestion de la mémoire en fonction des actions effectués par l'utilisateur. Comme dit précédemment, cela demandera du travail mais son implémentation sera aisée grâce à la simplicité des algorithmes.

Coté personnel

Personnellement, j'ai beaucoup aimé ce projet. Je l'ai trouvé à la fois ambitieux mais pas infaisable. Le thème est bien choisi car on utilise tous les jours un SGF sans même s'en rendre compte et j'ai trouvé très intéressant de comprendre et de construire notre propre SGF. Beaucoup de notions d'Ada ont été revues mais je n'ai jamais eu l'impression d'avoir à sortir du cadre du cours pour devoir faire ce projet, ce qui est plutôt appréciable au vu de l'envergure du projet.

Au sujet du temps passé sur le projet, je ne pense pas m'être bien organisé. Comme dit plus tôt dans le rapport, je pense avoir passé trop de temps à la conception, à essayer de réfléchir infructueusement à chaque détail, ce qui m'a fait prendre du retard. Après réflexion critique, je pense que j'aurais dû commencer petit, afin de mieux cerner le projet, quitte à me tromper sur certaines procédures. J'ai été trop ambitieux en voulant tout planifier du premier coup. J'en tiendrai compte pour le projet avenir.

D'un autre côté, une fois parti, j'estime avoir été assez efficace. Je n'ai jamais eu de gros blocages. Hormis quelques erreurs d'inattention, les compilations se passaient bien, et le programme s'exécutait comme je l'attendais. Tout cela ne m'a pas laissé beaucoup de temps pour le rapport que j'ai écrit en peu de temps.

Tables des illustrations