Rapport de projet Ada - Enzo Furriel

Résumé

L'objectif du projet est d'appliquer les thématiques vues pendant les cours de **Méthodologie de Programmation** à travers la création d'un **Système de Gestion de Fichiers** virtuel (SGF) à l'aide d'un cahier des charges fourni. Le projet suivra donc trois étapes prioritaires : la **conception du projet** (langages algorithmiques et en suivant des mécanismes de raffinage vu pendant les TD), la **réalisation du projet** (langage Ada vu pendant les TP), les **procédures de tests** de notre programme. Dans ce rapport, nous décrivons la manière dont nous avons suivis ces méthode.

Introduction

La **problématique principale** est de créer un **Système de Gestion de Fichiers (SGF)**. Il s'agit d'un composant permettant de **manipuler des fichiers, des répertoires** et de **gérer l'espace mémoire occupée par ces fichiers**. Ce système de fichier ressemblera à celui de linux, il organise les répertoires de manière hiérarchique avec un répertoire racine appelé root et représenté par « / ». Un **répertoire** est défini par un nom, des droits d'accès, un répertoire parent et un contenu constitué d'autres répertoires et fichiers. Un **fichier** est caractérisé par un nom, une taille, des droits d'accès et plusieurs autres informations.

Le SGF que nous allons créer devra **permettre** :

- la création du SGF avec un répertoire racine
- la création d'un répertoire
- la suppression d'un répertoire
- la copie d'un fichier
- le déplacement d'un fichier
- la création de l'archive d'un dossier
- de lister le contenu d'un dossier
- l'affichage du répertoire courant
- de changer de répertoire courant.

Pour créer ce système de fichier, il y a eu **trois étapes principales**. Tout d'abord, nous avons **conçu le projet** en menant des choix d'architecture et d'algorithmie. Ensuite, nous avons **réalisé le projet** et nous dresserons les difficultés rencontrés ainsi que les solutions mises en place. Enfin, nous avons **effectué des tests** pour valider la pertinence des choix et l'implémentation du code. Nous conclurons avec un bilan technique et personnel sur les apports du projet.

I - Conception du projet

1. Présentation des principaux types de données

Après **analyse** du sujet et notamment de la structure d'un système de gestion de fichiers, nous avons décidé d'utiliser la structure **d'arbre** que nous avons pu mettre en place sur le TP des arbres binaires. En effet, un système de gestion de fichiers se compose de répertoires pouvant contenir eux-mêmes d'autres répertoires, les notions de père, fils et frères sont donc représentées. De plus, il n'y a pas de limites de répertoires contenus pour un répertoire parent. Nous sommes venus à la conclusion qu'il s'agit d'un arbre pouvant contenir jusqu'à **n branche** (n fils) avec n étant un entier naturel. Cela a donné naissance à **l'arbre nr** où arb_nr serait un **pointeur** vers un **enregistrement noeud** contenant une valeur T (généricité) et éventuellement un arb_nr en père, un arb_nr en frère et un arb_nr en fils.

Cela nous permettra d'accéder aux répertoires parents/frères/fils dans notre système de gestion de fichiers.

Ensuite, nous nous sommes penchés sur ce que la valeur T pourrait être dans l'enregistrement noeud. Il pourrait s'agir d'un **enregistrement Répertoire** contenant le nom du répertoire avec une chaîne de caractères, une taille en entier, des droits en entiers (0 à 777) et s'il s'agit d'un fichier ou non (non = dossier) avec un booléen.

Pour construire notre SGF, nous allons donc implémenter un **arbre de répertoires**.

2. Architecture de l'application en modules

Nous avons choisis de mettre en œuvre une architecture **MVC** (Modèle Vue Contrôleur).

Menu correspond à la **vue**. C'est l'interface qui formule la volonté de l'utilisateur et la transmet au contrôleur.

Le **contrôleur**, c'est l'ensemble des fonctions et procédures du paquet sgf. Elles sont appelées par la vue avec les paramètres de l'instruction. Il faut aussi spécifier sur quel objet du modèle on souhaite effectuer l'instruction.
Le contrôleur applique donc une instruction donnée, sur un objet donné.

Les **modèles** ici sont `arb_nr` et `Repertoire`, qui se trouvent dans les paquets suivants : paquet répertoire, paquet arbre.
Ce sont les données utilisées par le programme.

Par **contrainte de temps et d'encapsulation**, nous avons laissé les visibilités des différents en public. Une **amélioration envisagée** serait de modifier la visibilité des modèles en privé et la vue ferait seulement appel au contrôleur, mais il faudrait alors développer des fonctions accesseurs et des procédures de modificateurs.

3. Présentation des principaux algorithmes et raffinages

Pour réaliser le système de gestion de fichier, nous avons pensé que **trois fonctions** pourraient être indispensables pour notre contrôleur :

- **Vérification d'une destination** (permet de vérifier si une destination saisie par l'utilisateur existe et retourne l'arbre correspondant ou null)
- **Récupérer le nom du répertoire** saisie par l'utilisateur (récupérer le dernier argument saisie par l'utilisateur dans une destination et le retourne), il s'agit donc d'un parseur (délimitation par un caractère.
- **Rechercher par nom de répertoire** (permet de vérifier si un répertoire du nom saisie par l'utilisateur existe et retourne sa position correspondant ou 0)

Nous pensons qu'il est possible de **créer chaque commande** de manière plus **simplifiée** en implémentant ses 3 fonctions, voici les **étapes de raffinages** qui nous ont conduit aux **algorithmes** finaux.

Rechercher par nom :

Raffinage :

```
R0. Comment vérifier si un répertoire du nom saisie par l'utilisateur existe ou non parmi l'un 
R1. Comment R0 ?
    A1. Vérifier si un répertoire d'un fils de l'arbre actuel a même nom que celui saisi par l'
R2. Comment A1 ?
    A2.1. Vérifier si l'arbre actuel a des fils (si non alors le repertoire n'existe pas)
    A2.2. Vérifier si le premier fils de l'arbre actuel a le même nom de répertoire
    A2.3. Véifier si le frère du premier fils de l'arbre actuel a le même nom de répertoire
R3. Comment A2.1 ?
    A3. Si Premier fils de arbre_temporaire existe Alors
            A2.2
        Sinon
            Retourne 0
        Fin Si
R4. Comment A2.2 ?
    A4. arbre_temporaire => premier fils de arbre_temporaire
        Si nom_repertoire est égal à nom du répertoire de la valeur de arbre_temporaire Alors
            retourne 1
        Sinon
            A2.3
        Fin Si
        Retourne 0

R5. Comment A2.3 ?
    A5. Répéter jusqu'à que Frere de arbre_temporaire existe
            arbre_temporaire => frere de arbre_temporaire
            compteur => compteur + 1
            Si nom_repertoire est égal à nom du répertoire de la valeur de arbre_temporaire Alo
               Retourne compteur
```

```
            Fin si
         Fin répéter
```

Algorithme :

```
Paramètres : arbre : arbre nr IN; nom_repertoire : chaines de caractères IN
   arbre_temporaire => arbre
    Si Premier fils de arbre_temporaire existe Alors
       arbre_temporaire => premier fils de arbre_temporaire
      Si nom_repertoire est égal à nom du répertoire de la valeur de arbre_temporaire Alors
         retourne 1
      Sinon
         Répéter jusqu'à que Frere de arbre_temporaire existe
            arbre_temporaire => frere de arbre_temporaire
            compteur => compteur + 1
            Si nom_repertoire est égal à nom du répertoire de la valeur de arbre_temporaire Alo
               Retourne compteur
            Fin si
         Fin répéter
      Fin si
    Sinon
       Retourne 0
    Fin si
```

Récupérer nom répertoire (parser) :

Il s'agit donc d'un délimiteur de caractères permettant de retourner le dernier élément délimité après un '/'.

Raffinage :

```
R0. Comment délimiter les caractères '/' de telle sorte à récupérer le dernier argument d'une cl
R1. Comment R0 ?
    A1.1 Initialiser un tableau de chaîne de caractères
    A1.2. Vérifier chaque caractère et délimiter les '/'
R2. Comment A1 ?
    A2. Pour i allant de 0 à 9 Faire
               initialiser parametres(i)
          Fin Pour
R3. Comment A1.2 ?
    A3.1. Parcourir chaque caractère
    A3.2. Vérifier s'il s'agit d'un '/' si c'est le cas incrémenter le nombre de paramètres (ar
R4. Comment A3.1 ?
    A4. Tant que (indice <= longueur(destination_repertoire)) Faire
            A3.2
        indice => indice + 1
R5. Comment A3.2 ?
    A5. Si le caractère est un "/" Alors
```

```
            nombre paramètres => nombre paramètres + 1
        Sinon
            ajouter la nouvelle lettre a parametres(nombre parametres)
        Fin si
```

Algorithme : (parametres étant un tableau de chaines de caractères)

```
Parametres : destination_repertoire : chaines de caractères IN
Pour i allant de 0 à 9 Faire
   initialiser parametres(i)
Fin pour
nombre paramètres => 0
Tant que (indice <= longueur(destination_repertoire)) Faire
   Si le caractère est un "/" Alors
      nombre paramètres => nombre paramètres + 1
   Sinon
      ajouter la nouvelle lettre a parametres(nombre parametres)
   Fin si
   indice => indice + 1
Fin tant que

Retourner parametres(nombre_parametres)
```

Vérifier destination :

Raffinage :

```
R0. Comment vérifier si une destination saisie par l'utilisateur existe et si oui retourner l'a
R1. Comment R0 ?
    A1.1. Parser pour récupérer chaque nom de repertoire saisi par l'utilisateur
    A1.2. Vérifier si la référence est en absolue ou en relative et se déplacer si elle est en 
    A1.3. Vérifier si chaque paramètres est un répertoire présent dans l'arbre temporaire (arbr
R2. Comment A1.1 ?
    A2.1. Parser (vu à récupérer nom repertoire')
R3. Comment A1.2 ?
    A3. Si le premier caractère de destination_repertoire est un "/"
            Tant que Pere de arbre_temp existe Faire
                arbre_temp => pere de arbre_temp
            Fin Tant que
        Fin Si
R4. Comment A1.3 ?
    A4.1. Parcourir les différents arguments
    A4.2. Vérifier si l'argument parcouru désigne le père ou un dossier du répertoire et se dép
R5. Comment A4.1 ?
    A5. Pour i allant de 0 à nombre_parametres Faire
            A4.2
        Fin Pour
R6. Comment A4.2 ?
    A6. Si Parametres(i) est ".." Alors
            Si Pere de arbre_temporaire existe Alors
```

```
                arbre_temporaire => Pere de arbre_temporaire
            Sinon
                Retourne null
            Fin Si
        Sinon Si Parametres(i) est différent de "." Alors
            Utiliser rechercher_par_nom pour vérifier si un répertoire de nom Parametres(i) exi
            Si le repertoire existe Alors
                arbre_temporaire => arbre_temporaire correspondant au fils du répertoire reche
            Sinon
                Retourne null
            Fin Si
        Fin Si
```

Algorithme :

```
Paramètres : arbre arbre nr IN; destination_repertoire chaine de caractères IN)
--- PARSER (vu à 'recuperer nom repertoire')
arbre_temp => arbre
Si le premier caractère de destination_repertoire est un "/"
   Tant que Pere de arbre_temp existe Faire
      arbre_temp => pere de arbre_temp
   Fin Tant que
Fin Si
Pour i allant de 0 à nombre_parametres Faire
   Si Parametres(i) est ".." Alors
      Si Pere de arbre_temporaire existe Alors
         arbre_temporaire => Pere de arbre_temporaire
      Sinon
         Retourne null
      Fin Si
   Sinon Si Parametres(i) est différent de "." Alors
      Utiliser rechercher_par_nom pour vérifier si un répertoire de nom Parametres(i) existe
      Si le repertoire existe Alors
         arbre_temporaire => arbre_temporaire correspondant au fils du répertoire recherché
      Sinon
         Retourne null
      Fin Si
   Fin Si
Fin Pour
Retourne arbre_temporaire
```

II - Réalisation du programme

Présentation des principaux choix réalisés

Pour la liste des commandes et son implémentation, nous avons décidé de nous approcher le plus possible des mécanismes du SGF de **Linux**, les commandes et leurs utilisations se ressemblent beaucoup et l'utilisateur peut écrire les commandes en référence **relative** ou **absolue**.

Voici la liste des **commandes** que le SGF propose à l'utilisateur :

```
Voici les differentes commandes que vous pouvez ecrire :

=> cd : |destination| : Changer de direction de repertoire

=> cp |destination_source|[nom_fichier] [destination_cible] : Copier un fichier

=> ls |destination| : Afficher la liste des fichiers et dossiers

=> ls -r |destination| : Afficher tous les repertoires et tous les sous-repertoi
res

=> mkdir |destination|[nom_dossier] : Creer un dossier

=> mv |destination_source|[nom_fichier] [destination_cible] : Deplacer un fichie
r

=> pwd : Afficher le repertoire courant

=> quit : Quitter l'interface

=> reset : Creer un SGF avec que le repertoire racine

=> rm |destination|[nom_fichier] : Supprimer un fichier

=> rm -r |destination|[nom_dossier] : Supprimer un dossier

=> size |destination|[nom_fichier] [taille_fichier] : Changer la taille d'un fichier

=> tar |destination|[nom_dossier] : Archiver un dossier

=> touch |destination|[nom_fichier] : Creer un fichier
================================================================================
Legende : |param| => parametre facultatif [param] => parametre obligatoire
```

Pour en savoir plus sur une fonction, les spécifications et contrat sont décrites dans les commentaires des différentes fonctions.

Démarche adopté pour tester le programme

Nous avons décidé de **tester** les **fonctions principales** de notre programme en respectant la méthode **d'assertions** vérifiant si la valeur de retour est bien celle souhaitée lors du **jeu d'essai** préalablement choisie.

2 fichiers de tests existent pour tester le programme :

- `test_arbre` permet de tester les fonctions du package `arbre_nr`
- `test_sgf` permet de tester les fonctions du package `sgf`

Test_Arbre :

Arbre possède **7 fonctions** à tester dont **4 accesseurs** :

- `An_Get_Pere(arbre)` : retourne l'arbre père ou null
- `An_Get_PremierFils(arbre)` : retourne l'arbre premier fils ou null
- `An_Get_Frere(arbre)` : retourne l'arbre frère ou null
- `An_Get_Valeur(arbre)` : retourne la valeur de type T

Jeu d'essai :

- `arbre1` a comme valeur l'entier `1` n'a ni premier fils, ni frère, ni père
- `arbre2` a comme valeur l'entier `2` n'a pas de premier fils ni de frère mais a `arbre1` comme père
- `arbre3` a comme valeur l'entier `-5` n'a pas de père mais a `arbre2` comme premier fils et `arbre1` comme frère
- `An_Get_Pere(arbre1)` doit être `null` comme `arbre1` n'a pas de père
- `An_Get_Pere(arbre2)` doit être `arbre1` comme `arbre1` est le père de `arbre2`
- `An_Get_PremierFils(arbre1)` doit être `null` comme `arbre1` n'a pas de premier fils
- `An_Get_PremierFils(arbre2)` doit être `null` comme `arbre2` n'a pas de premier fils
- `An_Get_PremierFils(arbre3)` doit être `arbre2` comme `arbre2` est le premier fils de `arbre3`
- `An_Get_Frere(arbre1)` doit être `null` comme `arbre1` n'a pas de frère
- `An_Get_Frere(arbre2)` doit être `null` comme `arbre2` n'a pas de frère `An_Get_Frere(arbre3)` doit être `arbre1` comme `arbre1` est le frère de `arbre3`

```
arbre1 := new noeud'(1, null, null, null);
arbre2 := new noeud'(2, null, null, arbre1);
arbre3 := new noeud'(-5, arbre2, arbre1, null);

---------------------- TEST GETTER

-- Test An_Get_Pere : récupérer le père d'un arbre
Assert(An_Get_Pere(arbre1) = null);
Assert(An_Get_Pere(arbre2) = arbre1);

-- Test An_Get_Premier_Fils : récupérer le premier fils d'un arbre
Assert(An_Get_PremierFils(arbre1) = null);
Assert(An_Get_PremierFils(arbre2) = null);
Assert(An_Get_PremierFils(arbre3) = arbre2);

-- Test An_Get_Frere : récupérer le frère d'un arbre
Assert(An_Get_Frere(arbre1) = null);
Assert(An_Get_Frere(arbre2) = null);
Assert(An_Get_Frere(arbre3) = arbre1);

-- Test An_Get_Valeur : récupérer la valeur d'un arbre
Assert(An_Get_Valeur(arbre1) = 1);
Assert(An_Get_Valeur(arbre2) = 2);
Assert(An_Get_Valeur(arbre3) = -5);
```

Il existe aussi **3 autres fonctions** à tester :

- `An_Vide(arbre)` : retourne vrai si l'arbre est vide sinon faux
- `An_Fils(arbre, position)` : retourne le n-ième fils d'un arbre
- `An_Nombre_Fils(arbre)` : retourne le nombre de fils d'un arbre

Jeu d'essai : (avant d'insérer des fils)

- `An_Vide(arbre3)` doit être `faux` puisque `arbre3` n'est pas vide
- `An_Vide(arbre4)` doit être `vrai` puisque `arbre4` est vide
- `An_Nombre_Fils(arbre1)` doit être `1` puisque `arbre1` n'a pas de fils et le minimum est 1
- `An_Nombre_Fils(arbre2)` doit être `1` puisque `arbre2` n'a pas de fils et le minimum est 1
- `An_Nombre_Fils(arbre3)` doit être `1` puisque `arbre3` a 1 fils

On crée deux arbres `arbre4` et `arbre5` pour pouvoir les insérer en fils de `arbre1`

Jeu d'essai (après avoir insérer des fils) :

- `An_Nombre_Fils(arbre1)` est `1` après lui avoir inséré `arbre4` comme fils de `arbre1` car `arbre1` a qu'un seul fils
- `An_Nombre_Fils(arbre1)` est `2` après lui avoir inséré `arbre5` comme fils de `arbre1` car `arbre1` a deux fils

- `An_Fils(arbre1, 1)` est `arbre5` car `arbre5` est le premier fils de `arbre1`
- `An_Fils(arbre1, 2)` est `arbre4` car `arbre4` est le deuxième fils de `arbre1`

```
-- Test An_Vide : savoir si un arbre est vide ou non
Assert(An_Vide(arbre3) = false);
Assert(An_Vide(arbre4) = true);

-- si pas de fils retourne 1 sinon retourne le nombre de fils
Assert(An_Nombre_Fils(arbre1) = 1);
Assert(An_Nombre_Fils(arbre2) = 1);
Assert(An_Nombre_Fils(arbre3) = 1);

arbre4 := new noeud'(4, null, null, null);
arbre5 := new noeud'(5, null, null, null);

An_Inserer_Fils(arbre1, arbre4);
Assert(An_Nombre_Fils(arbre1) = 1);
An_Inserer_Fils(arbre1, arbre5);
Assert(An_Nombre_Fils(arbre1) = 2);

Assert(An_Fils(arbre1, 1) = arbre5);
Assert(An_Fils(arbre1, 2) = arbre4);

Assert(An_Fils(arbre1, 1) /= arbre4);
Assert(An_Fils(arbre1, 2) /= arbre5);
```

Test_SGF :

Nous allons tester les fonctions vues précédemment lors de l'étape de raffinage et d'algorithmie :

- `recuperer_nom_repertoire(arbre, destination_repertoire)`
- `rechercher_par_nom(arbre, nom_repertoire, est_fichier)`
- `verifier_destination(arbre, destination_repertoire, supprimer_dernier_mot_cle)`

Récuperer nom répertoire :

Nous commençons d'abord par **initialiser** un SGF avec un arbre et nous cherchons à savoir si le **répertoire récupéré** est bien le **dernier argument** saisi par l'utilisateur (récupérer le dernier argument après avoir délimité les "/").

Jeu d'essai :

- La chaîne de caractères `.` concerne le nom du répertoire (destination) `.`
- La chaîne de caractères concerne le nom du répertoire (destination)
- La chaîne de caractères `fichier1` concerne le nom du répertoire (destination) `fichier1`
- La chaîne de caractères `dossier1/dossier2/fichier` concerne le nom du répertoire (destination) `fichier`

- La chaîne de caractères `/dossier1/dossier2/fichier` concerne le nom du répertoire (destination) `fichier`
- La chaîne de caractères `/dossier1/dossier2/` concerne le nom du répertoire (destination) `dossier2`

```
-- initialisation du sgf
initialisation_sgf(arbre_actuel);

-----------------------------------------------------------------------------------------------------------------------
-- Test de la fonction recuperer_nom_repertoire (permet de récuperer le nom du répertoire dans une chaine de caractère envoyé (parse)
-----------------------------------------------------------------------------------------------------------------------
Assert(recuperer_nom_repertoire(arbre_actuel,To_Unbounded_String(".")) = To_Unbounded_String("."));
Assert(recuperer_nom_repertoire(arbre_actuel,To_Unbounded_String(" ")) = To_Unbounded_String(" "));
Assert(recuperer_nom_repertoire(arbre_actuel,To_Unbounded_String("fichier1")) = To_Unbounded_String("fichier1"));
Assert(recuperer_nom_repertoire(arbre_actuel,To_Unbounded_String("dossier1/dossier2/fichier")) = To_Unbounded_String("fichier"));
Assert(recuperer_nom_repertoire(arbre_actuel,To_Unbounded_String("/dossier1/dossier2/fichier")) = To_Unbounded_String("fichier"));
Assert(recuperer_nom_repertoire(arbre_actuel,To_Unbounded_String("/dossier1/dossier2/")) = To_Unbounded_String("dossier2"));
```

Rechercher par nom :

Pour tester la fonction **rechercher_par_nom**, il faut tout d'abord **créer** plusieurs dossiers et fichiers et l'on regarde ensuite si le **fichier cherché** se situe dans l'arbre actuel et retourne sa position (s'il ne se situe pas le résultat est 0).

Jeu d'essai :

- Le dossier `enzo` n'est pas l'un des répertoire fils de l'arbre actuel donc doit donner 0
- Le dossier `home` est l'un des répertoires fils et se trouve en position 6, il doit donner 6
- Le dossier `dossier3` est l'un des répertoires fils et se trouve en position 4, il doit donner 4
- Le fichier `enzo` n'est pas l'un des répertoire fils de l'arbre actuel donc doit donner 0
- Le fichier `f1` est l'un des répertoires fils et se trouve en position 3, il doit donner 3
- Le fichier `f3` est l'un des répertoires fils et se trouve en position 1, il doit donner 1

```
-- création des fichiers pour le prochain test
creer_repertoire(arbre_actuel, To_Unbounded_String("home"), 0, 777, false);
creer_repertoire(arbre_actuel, To_Unbounded_String("usr"), 0, 777, false);
creer_repertoire(arbre_actuel, To_Unbounded_String("dossier3"), 0, 777, false);

creer_repertoire(arbre_actuel, To_Unbounded_String("f1"), 0, 777, true);
creer_repertoire(arbre_actuel, To_Unbounded_String("f2"), 0, 777, true);
creer_repertoire(arbre_actuel, To_Unbounded_String("f3"), 0, 777, true);

-----------------------------------------------------------------------------------------------------------------------
-- Test de la fonction rechercher_par_nom (vérifie si un répertoire recherché existe et retourne sa position ou null)
-----------------------------------------------------------------------------------------------------------------------
-- sachant que quand on crée un répertoire la position de chaque répertoire incrémente de 1

-- Test dossier
Assert(rechercher_par_nom(arbre_actuel, To_Unbounded_String("enzo"), false) = 0);
Assert(rechercher_par_nom(arbre_actuel, To_Unbounded_String("home"), false) = 6);
Assert(rechercher_par_nom(arbre_actuel, To_Unbounded_String("dossier3"), false) = 4);

-- Test fichier
Assert(rechercher_par_nom(arbre_actuel, To_Unbounded_String("fichier"), true) = 0);
Assert(rechercher_par_nom(arbre_actuel, To_Unbounded_String("f1"), true) = 3);
Assert(rechercher_par_nom(arbre_actuel, To_Unbounded_String("f3"), true) = 1);
```

Verifier destination :

Pour tester la fonction **verifier_destination**, cela dépend de **l'arbre courant**, il y a donc un test en deux parties, l'un dans le **répertoire racine** et l'autre dans le **répertoire home** (appel de la fonction changer_destination_repertoire). Il faut savoir que le paramètre **supprimer_dernier_mot_cle** est un booléen permettant de savoir s'il faut faire la vérification sur la destination en supprimant le dernier argument ou non. La destination concerne **que des dossiers** et non des fichiers.

Jeu d'essai (répertoire courant est le répertoire racine) :

- La destination `home/dossier/dossier2` n'existe pas on doit avoir `null`
- La destination `/home/dossier` n'existe pas on doit avoir `null`
- La destination `home` existe on doit avoir `An_Fils(arbre_actuel,6)` car il s'agit du 6ème fils de l'arbre
- La destination `/home` existe on doit avoir `An_Fils(arbre_actuel,6)` car il s'agit du 6ème fils de l'arbre
- La destination `f1` n'existe pas on doit avoir `null`
- La destination `.` existe et concerne le répertoire courant, on doit avoir `arbre_actuel`
- La destination `/` existe et concerne la racine (même répertoire que répertoire actuel), on doit avoir `arbre_actuel`
- La destination `../` n'existe pas on doit avoir `null` (le répertoire racine ne peut pas avoir de répetoire père
- La destination `..` n'existe pas on doit avoir `null` (le répertoire racine ne peut pas avoir de répetoire père
- La destination `home/dossier/dossier2` en `supprimant le dernier argument` n'existe pas on doit avoir `null`
- La destination `home/dossier` en `supprimant le dernier argument` existe (il s'agit de `home`) on doit avoir `An_Fils(arbre_actuel,6)` car il s'agit du 6ème fils de l'arbre
- La destination `home` existe on doit avoir `An_Fils(arbre_actuel,6)` car il s'agit du 6ème fils de l'arbre
- La destination `home` en `supprimant le dernier argument` existe (il s'agit de `.`) on doit avoir `arbre_actuel`
- La destination `/home` en `supprimant le dernier argument` existe (il s'agit de `.`) on doit avoir `arbre_actuel`
- La destination `f1` en `supprimant le dernier argument` existe (il s'agit de `.`) on doit avoir `arbre_actuel`
- La destination `.` en `supprimant le dernier argument` existe (il s'agit de `.`) on doit avoir `arbre_actuel`
- La destination `/` en `supprimant le dernier argument` existe (il s'agit de `.`) on doit avoir `arbre_actuel`
- La destination `../` en `supprimant le dernier argument` existe (il s'agit de `.`) on doit avoir `arbre_actuel`
- La destination `..` en `supprimant le dernier argument` existe (il s'agit de `.`) on doit avoir `arbre_actuel`

Jeu d'essai (répertoire courant est le répertoire /home, on utilise maintenant arbre_temp = arbre du répertoire racine) :

- La destination `home` n'existe pas, on doit avoir `null`
- La destination `../home` existe, il s'agit du 6ème fils de arbre_temp ou arbre_actuel on doit avoir `An_fils(arbre_temp,6)`
- La destination `../` existe, il s'agit du répetoire racine, on doit avoir `arbre_temp`
- La destination `..` existe, il s'agit du répertoire racine, on doit avoir `arbre_temp`
- La destination `../../` n'existe pas, on doit avoir `null`
- La destination `../..` n'existe pas, on doit avoir `null`

```
-------------------------------------------------------------------------------------------------------------------
-- Test de la fonction verifier_destionation (vérifie si la destination existe et retourne l'arbre (pointeur) correspondant ou null)
-------------------------------------------------------------------------------------------------------------------

-- Test sans supprimer le dernier mot-clé et en partant de la racine
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("home/dossier/dossier2"), false) = null);
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("home/dossier"), false) = null);
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("home"), false) = An_Fils(arbre_actuel, 6));
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("/home"), false) = An_Fils(arbre_actuel, 6));
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("f1"), false) = null);
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("/home"), false) = An_Fils(arbre_actuel, 6));
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("."), false) = arbre_actuel);
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("/"), false) = arbre_actuel);
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("../"), false) = null);
Assert(verifier_destination(arbre_actuel, To_Unbounded_String(".."), false) = null);

-- Test en supprimant le dernier mot-clé et en partant de la racine
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("home/dossier/dossier2"), true) = null);
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("home/dossier"), true) = An_Fils(arbre_actuel, 6));
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("home"), true) = arbre_actuel);
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("/home"), true) = arbre_actuel);
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("f1"), true) = arbre_actuel);
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("/home"), true) = arbre_actuel);
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("."), true) = arbre_actuel);
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("/"), true) = arbre_actuel);
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("../"), true) = arbre_actuel);
Assert(verifier_destination(arbre_actuel, To_Unbounded_String(".."), true) = arbre_actuel);

-- Se déplacer dans un répertoire 'home' pour faire de nouveaux tests
arbre_temp := arbre_actuel;
changer_direction_repertoire(arbre_actuel, To_Unbounded_String("home"));
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("home"), false) = null);

-- Tests en partant du répertoire '/home' sans supprimer le dernier mot-clé
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("../home"), false) = An_Fils(arbre_temp, 6));
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("../"), false) = arbre_temp);
Assert(verifier_destination(arbre_actuel, To_Unbounded_String(".."), false) = arbre_temp);
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("../../"), false) = null);
Assert(verifier_destination(arbre_actuel, To_Unbounded_String("../.."), false) = null);
```

Toutes les **assertions** réalisées **fonctionnent**, on peut donc en déduire que les fonctions sont plutôt **fiables** avec le jeu de données que nous avons confectionné.

Difficultés rencontrées et solutions adoptées

- ## Chaîne de caractères :

Durant la réalisation du projet, nous avons eu des problèmes avec les chaînes de caractères. En effet, les chaînes de caractères ont une taille fixe et cela complexifie la saisie clavier de l'utilisateur. **Solution :** Afin de pallier ce problème, nous avons décidé d'utiliser des objets **unbounded_string** au lieu des

tableau de caractères, il s'agit de chaînes de caractères sans contraintes sur la taille, cela permet donc d'éviter pleins d'erreur ou de vérifications de la taille.

- ### Création d'une archive :

La commande tar pour créer une archive est assez complexe sur les systèmes de gestion de fichiers.
Solution : Nous avons décidé que notre commande tar permet de créer un fichier contenant le même nom que le dossier souhaité (succédé d'un ".zip") et de même taille dans le répertoire père.

- ### Implémentation de la mémoire :

Par soucis de complexité et de temps, nous n'avons pas exploité le mécanisme de mémoire contigu.
Solution : Nous avons donc décidé d'implémenter un système de gestion de mémoire simplifié :

- Un fichier créé a une taille de 5 Mo par défaut, il s'agit des métadonnées du fichier (sa taille minimale est de 5), elle peut être modifiée par la commande `size`.
- Un dossier créé a une taille de 0 Mo par défaut, sa taille dépend des fichiers présents à l'intérieur (il s'agit de la somme de la taille des différents fichiers)
- Par contrainte de mémoire sur la valeur maximale d'un entier, la mémoire maximale du SGF est atteinte lorsque le répertoire racine possède une taille de 10 Go (et non 1 To), si la taille est atteinte, il est impossible de créer un nouveau fichier.
- Les commandes pouvant influencer sur la mémoire sont : `touch / rm / rm -r / cp / mv / tar`.

- ### Tests du programme

Il a été assez complexe de prendre des mesures pour **tester le programme** étant donné que nous n'avons pas vu le **concept de tests** durant les cours et TP de cette matière. Il s'agissait donc d'une nouvelle thématique pour nous. Après de nombreuses recherches sur le sujet, j'ai pu voir qu'il était possible de réaliser des assertions pour vérifier que deux valeurs soient bien égales. Nous avons donc réaliser des assertions d'appel de fonction avec la valeur de retour attendue suite à notre jeu d'essai.

III - Bilans

Bilan technique

Répartition du temps :

Nous avions passé environ 50h par personne dans la **réalisation du projet**.

En ce qui concerne la **répartition du temps**, voici des pourcentages permettant d'estimer globalement le **temps passé** sur chaque partie :

- 60% à la conception
  - 20% sur les types de données

    - 40% sur la conception et le raffinage des algorithmes
- 5% sur les tests
- 25% sur l'implémentation
- 10% sur la documentation

Etat d'avancement du projet

Le système de gestion de fichier est **opérationnel** et toutes les commandes demandées par le cahier des charges sont **implémentées**. Il offre aussi la possibilité à l'utilisateur de **manipuler une destination** en référence absolue et relative. Le programme offre un **système de gestion de mémoire** (bien que simplifiée). On peut conclure que les **objectifs** du cahier des charges ont été **pleinement réalisés**.

Perspectives d'évolutions

Le projet possède de nombreuses possibilités **d'améliorations**. Effectivement, les modèles pourraient par exemple avoir une **visibilité privée** afin de garantir **l'encapsulation** et donc de restreindre l'accès direct aux états en empêchant la modification de l'objet hors de ses méthodes. On pourrait aussi améliorer le système de gestion de mémoire à l'aide d'une **bitmap** pour offrir au SGF le principe de **mémoire contigu** défini dans le cahier des charges.

Bilan personnel

Plusieurs **enseignements** sont à tirer de ce projet, et du cours de **"Méthodologie de la Programmation"** en général. Grâce à ces derniers, nous avons pris conscience de l'importance de la **bonne conception d'un programme**. Nous avons également découvert la **méthode des raffinages**, cette dernière nous a permis de **gagner un temps précieux** lors de **l'implémentation de notre projet**. C'est pour cela que nous avons tout d'abord longuement réfléchi sur la spécification, pendant 4 séances (8h) afin de s'assurer que notre modèle était viable.

Dans un cadre plus technique, nous avons appris à maîtriser certains aspects du **langage Ada**, ce qui est toujours intéressant pour notre bagage personnel. Nous nous sommes rendu compte que, même avec des *angages de bas niveau, il était possible de **développer des projets** d'envergure.

Le projet est fort intéressant et se prête plutôt bien à un travail en équipe. Les étudiants pourraient concevoir le sgf ensemble, se posant les bonne question d'architecture. Cela permettrait aussi d'atténuer les disparitées d'expériences RT/INFO et de transmettre les savoir-faire.