Rapport de Projet de Méthodologie de la Programmation

Objectif du rapport

Ce rapport présentera le projet réalisé pour le module "Méthodologie de la Programmation" de l'ENSEEIHT Toulouse.
Il abordera les différentes parties de la conception de ce projet. Il y sera présenté les choix techniques, les algorithmes utilisés, la structure du programme et les difficultés rencontrée.

Introduction

Le projet proposé consiste à créer un système de gestion d'arbre généalogique d'un individue. L'arbre généalogique permet à un individu de répertorier l'ensemble de ses ancêtres. On peut représenter l'ensemble des ancêtres par un arbre où chaque ou chaque individu est représenté par un nœud qui peut avoir deux parents.

> Exemple :
>
> - Bernard
>   - pere : Jackie
>   - pere : Dominique
>   - mere : Jane
>   - mere : Dominique

On voudra pouvoir designer les individus en relation des uns par rapport aux autres avec la notion de génération d'ancêtre et de descendant

> Exemple :
>
> - Un individue est sont propre ancetre/decendant de generation 0
> - Le pere d'un individue est sont ancetre de generation 1
> - La grand mere d'un individue est son ancetre de generation 2
> - Le fils d'un individue est sont dedendant de generation 1

Plan

Les modules étant déjà une sous-division structurante du projet du programme nous aborderont chaque aspect de conception pour chaque module.

Le rapport suivra le plan suivant :

Architecture du programme :

Afin d'assurer une bonne lisibilité et une bonne modularité, le code du projet a été décomposé en plusieurs moules ada par rapport à leurs fonctions.

Dans cette partie les modules constituant ce projet seront présenté avec leur but. Nous nous intéresserons qu'aux modules appartenant au projet et nous omettrons les modules standard de ada.

Les modules développé sont :

- **`p_arbre_binaire`** : Ce module générique contient l'entièreté des fonctions et des types nécessaire pour faire fonctionner un arbre binaire non trié. On y trouve le type générique`T_AB` et les sous programmes permettant de le manipuler. *Paramètre de généricité* :
  - T : type des donnés
  - "=" : comparaison entre deux T
  - put(x :T) : procédure d'affichage de T
- **`p_linked_list`** : Ce module générique contient les sous-programmes et les types afin de faire une liste chainée. Pour éviter de répéter du travail. On utilise un module développé dans le cadre des TPs *Paramètre de généricité* :
  - T : type des donnés
  - afficher(x :T) : procédure d'affichage de T
- **`p_personne`** : Ce module contient le type `t_personne` qui représente un individu. Il y a aussi les différents sous-programmes permettant de le manipuler (init/getter/setter/comparaison/affichage).
- **`p_arbre_genealogique`** : Ce module instancie `p_arbre_binaire` avec `t_persone` et ses sous programme associé. Dans ce module on retrouve aussi toutes les procédures de manipulation d'un arbre généalogique qui ne n'ont pas de sens dans un arbre binaire non trié quelconque. Dans ce module on renomme `T_AB` en `T_AG`. Dans ce module on instancie aussi `p_linked_list` avec `t_personne` car certaines fonctions ont besoin de renvoyer une liste d'individus.
- **`p_front`** : dans ce module on retrouve toutes les procédures et fonction d'entré/sortie nécessaire pour le programme principal(`main`).
- **`main`** et le programme principal il contient la logique du menu et fait les appels à `p_arbre_généalogique` et à `p_arbre_binaire` en fonction des entré de l'utilisateur. `main` mets en relation le modèle et le frontend qui interagit avec l'utilisateur.

Types

Dans cette partie nous nous intéresserons aux types contenue dans les différents modules.

`p_arbre_binaire :`

Dans ce module on veut mettre en place un arbre binaire non trié. Un arbre binaire non trié structurellement n'est pas diffèrent de tout autre arbre binaire. Il est constitué de cellules contenant chacune deux références à d'autre cellule et un élément de donné

Pour ce faire il nous faut trois types :

- **`T_Element`**(privé) : Un type en paramètre de généricité du module.
- **`T_AB`**(visible privé) : Un pointeur vers une cellule de l'arbre(`T_AB_Cell`). Représente un arbre binaire.
- **`T_AB_Cell`**(privé) : Un record qui représente une cellule d'arbre :

```
- gauche : T_AB
- droite : T_AB
- valeur : T_Element
```

`p_personne`

Nous allons vouloir représenter des individus. Pour ce faire on utilisera un type record `T_Personne` :

```
- id : integer
- nom : unbounded_string
- prenom : unbounded_string
```

Le type `T_personne` n'est encore utilisé que pour contenir le nom et le prénom de l'individue mais le type pourrait être très facilement étendu en modifiant la déclaration du type.

`P_linked_list`

Certains sous programmes nécessiterons une liste pour fonctionner. Il faudra donc un type `T_linked_list` qui représente une liste

Dans ce module on a 2 types :

- `T_linked_list_cell` : Un record qui constitue la liste

```
- value : T
- next : T_linked_list
```

- `T_linked_list` : un pointeur vers `T_linked_list_cell`.

`p_arbre_genealogique`

Nous allons vouloir faire un arbre généalogique. On peut se servir de l'arbre binaire de `p_arbre_binaire` pour faire notre arbre généalogique. On utilisera le type `T_AG` qui est une implémentation de `T_AB` avec `T_personne` renommé.

Les sous programmes de ce module nécessite de faire des listes de personnes nous implémenterons donc le module `p_linked_list` avec `T_personne` pour obtenir une `t_linked_list` qu'on renommera en `L_personne`.

Choix de conception et choix technique :

Séparation des fonctionnalité en package et généricité:

Dans tout projet le code est divisée en plusieurs modules afin de pouvoir améliorer la réutilisation du code, faciliter la lisibilité et pour la généricité.

Idéalement

Idéalement le programme devrais être séparé en module à responsabilité unique chacun ne laissant apparaitre que des structures de donnés privé et des sous programmes pour interagir avec les structure. Les paquets ne devrai contenir que des méthodes en rapport direct avec la méthode. Il est aussi important d'appliquer la généricité à ce qui pourrait être réutilisé pour plusieurs types de donnés.

Pour chaque fonctionnalité/sous programme, on se demande s'il correspond a un module existant ou si elle ne serait pas logique dans le module.

> Ex : supprimer un element est generale a tout arbre binaire trouver un element qui est l'encetre d'un autre n'est pas logique

Compromis

Typiquement on utilise l'orienté objet d'atteindre ces objectifs. Mais ce n'est pas

dans le domaine du projet et ada n'est pas le langage idéal pour l'orienté objet.

L'orienté objet aurait pu nous permet d'implémenter et d'étendre des classes générique. Sans briser la contrainte de n'exposer que ce qui est nécessaire à l'utilisateur finale. On aurait pu nous utiliser le protected pour étendre et implémenter `p_arbre_binaire` avec des nouvelles méthodes dans `p_arbre_genealogique`.

Pour essayer de respecter le mieux possible ces objectifs nous allons rendre les structures de donnés privé et nous allons rendre disponible des getter et des setter pour `T_arbre_binaire` afin que des packages basé sur `P_arbre_binaire` puissent accéder à l'arbre.

UX / UI : Expérience utilisateur et interface utilisateur :

On sait que le projet doit être un programme menu mais il y a plusieurs moyen de répondre à ces contraintes.

Apparence ciblée :

Cette application va prendre inspiration d'application textuelle comme top/htop, vim et nano.

L'utilisateur pourra designer par les nœud par leur id mais ce n'est pas pratique pour l'utilisateur car ça nécessite de la mémorisation. Pour résoudre le problème on gardera l'arbre affiché de manière permanente.

On mettra aussi des menus contextuels pour chaque action plutôt que d'utiliser un affichage linaire dans le terminal.

On pourra aussi utiliser des couleurs pour différencier les erreurs du reste des sorties.

NCurse :

Une des méthode qui a été considéré est d'utiliser la bibliothèque ncurse qui fournie de nombreux outils pour créer des interfaces textuel.

Cependant ncurse pose un problème de dépendance et de surcharge du projet. ncurse est bien plus gros que ce qui est nécessaire pour l'objectif. De plus ncurse est une dépendance qui n'est pas facilement disponible donc elle pourrait compliquer grandement l'installation de notre projet qui n'utilise que la bibliothèque standard.

ANSI escape character :

L'alternative à ncurse qui a été choisi est l'utilisation des character d'échappement ANSI. Ces characters sont des characters qui transmis au terminal permettent de donner des commande qui permettent de bouger un curseur, supprimer des parties du terminal, changer les couleur du texte...

Cette option bien que moins étendu nous permet d'avoir une interface textuelle sans rajouter de dépendances.

Division pseudo MVC (Modèle, Vue, Contrôler) :

Une bonne pratique appliqué dans plusieurs projet est de séparer les structure de donné les programmes principaux et les interface. Dans le patron MVC le contrôler fait les appels au modèles et a l'interface et est celui qui fait le lien entre les deux.

Pour ce projet on essaie de respecter cette philosophie en mettant l'interface dans des package qui leur sont spécifique. Similairement les modèles sont représentés par les packages qui contiennent les types.

Fonctions Et Procédures

Dans cette partie nous nous intéresserons aux procédures et fonctions importantes de chaque module.

Nous omettrons certaines fonctions et procédures qui ne sont pas complexe. La spécification complète des packages avec la spécification des sous-programmes seront disponible en annexe.

Fonctions et procédures de `P_arbre_binaire` :

Calculer la taille (`calcul_taille`)

Pour calculer la taille d'un arbre binaire se calcule avec un algorithme de récursion. En effet on peut décomposer le problème en sommer la taille des deux sous arbre + 1.

Spécification

```
-- nom : calcul_taille
-- sémantique :  renvoie nombre de nœuds de l'arbre
-- paramètres :
  -- arbre : in  T_AB -- Arbre dont on veut le nombre de nœuds
-- retour : integer --nombre nœud de l'arbre
-- pré-condition :
-- post-condition :
-- Tests de la procédure :
function calcul_taille(arbre : in T_AB) return integer;
```

Rafinage

```
R0 : calculer le  nombre de nœuds de l'arbre
R1 : comment "calculer le  nombre de nœuds de l'arbre"  --arbre : in
T_AB
ajouter le noeud courant + sosu arbre gauche + sous_arbre droit
R2 : comment "ajouter le noeud courant + sosu arbre gauche + sous_arbre
droit"
  if(arbre = null) then
    return 0;
  else
    return 1+ calcul_taille(arbre.all.gauche) +
calcul_taille(arbre.all.droite)
  end if;
```

Rechercher un nœud (`rechercher`)

Rechercher fais partie des fonctions qui change entre un arbre binaire trié et non trié. Dans un arbre non trié nous n'avons aucune information pour trouver l'élément. Il nous faut donc parcourir récursivement l'arbre pour trouver l'element.

Specification

```
-- nom : rechercher
-- sémantique :  rechercher un noeud dans un arbre
-- paramètres : racine : in T_AB --l'arbre où on fait une recherche
  -- valeur : in T_element -- le noeud à rechercher
  -- arbre : in  T_AB -- Arbre dont on veut le nombre de nœuds
-- retour : T_AB -- le noeud
-- pré-condition : l'arbre est initialisé
-- post-condition : retourne le noeud ou null si le noeud existe pas
-- Tests de la procédure :
function rechercher(racine : in  T_AB ; valeur : in T_element)
  return T_AB;
```

Raffinage

```
R0 : "rechercher un noeud dans un arbre"
R1 : comment "rechercher un noeud dans un arbre"  -- arbre : in  T_AB ;
valeur : in
  if(arbre =null or else arbre.all.element = valeur) then
    return arbre;
  end if;
  result := rechercher(arbre.all.gauche);
  if(result = null)
      result := rechercher (arbre.all.droite);
  end if;
  return result;
```

afficher arbre (`afficher`)

On cherche à reproduire la trace ci-dessou. Pour ce faire on va utiliser un algorithme récursif. Mais cependant on voit qu'afficher un sous arbre n'est pas exactement la même chose qu'afficher un arbre en effet la tabulation varie en fonction de la profondeur il nous faudra donc passer la profondeur dans les paramètres. Afin de respecter la trrace, on fera aussi une récursion avec l'ordre `racine->sous arbre gauche->sous arebre droit`.

Trace

```
[1] Minet Bernard
   Pere : [2] Minet Jack
      Pere : [4] Minet Pierre
      Mere : [5] Brosse Anne
   Mere : [3] Legrand Jean
```

plus généralement :

```
<element>
   <label-gauche> : <element>
      <label-gauche> : <element>
      <label-droit> : <element>
   <label-droit> : <element>
```

Spécification

```
-- nom : afficher
-- sémantique :  affiche un arbre
-- paramètres :
  -- arbre : in  T_AB -- Arbre qu'on veut afficher
  -- profondeur : in integer
        -- Profondeur de cette arbre pour affichage
        -- (déterminé le décalage de l'arbre)
  -- etiquette_gauche : in unbounded_string
        -- Étiquette pour la partie gauche
  -- etiquette_droite : in unbounded_string
        -- Étiquette pour la partie droite
-- pré-condition :
-- post-condition :
-- Tests de la procédure :
procedure afficher(arbre : in T_AB;
                   profondeur : in integer;
                   ethiquette_gauche,
                   etiquette_droite : in unbounded_string;
```

Rafinage

```
R0 : "affiche un arbre"
R1 : Comment "affiche un arbre"  -- arbre : in  T_AB; profondeur : in
integer;
                                 -- etiquette_gauche : in
unbounded_string
```

```
                                    -- etiquette_droite : in
unbounded_string
  if(arbre = null) then raise arbre_exception; end if;
  afficher racine;
  if(arbre.all.droite /= null )then
    afficher sous arbre droit
  end if;
  if(arbre.all.gauche /= null )then
    afficher sous arbre gauche
  end if;
R2 : Comment “afficher racine”  -- arbre : in  T_AB;
  afficher_element(arbre.all.element);
  new_line;
R2 : Comment afficher sous arbres droite
                    -- arbre : in  T_AB;
                    -- profondeur : in integer;
                    -- etiquette_gauche : in unbounded_string;
                    -- etiquette_droite : in unbounded_string;
  for i in 1..profondeur loop
    put(“   ”);
  end loop;
  put(etiquette_droite);
  put(“ : ”);
  afficher(arbre.all.droite, profondeur+1, etiquette_droite,
etiquette_gauche)
R2 : Comment afficher sous arbres gauche
                    -- arbre : in  T_AB;
                    -- profondeur : in integer;
                    -- etiquette_gauche : in unbounded_string;
                    -- etiquette_droite : in unbounded_string;
  for i in 1..profondeur loop
    put(“   ”);
  end loop;
  put(etiquette_gauche);
  put(“ : ”);
  afficher(arbre.all.gauche, profondeur+1, etiquette_droite,
etiquette_gauche)
```

Supprimer élément(`supprimer`)

La suppression d'un élément est très similaire à la recherche. On fait juste

attention à rechercher avec une profondeur d'avance afin de pouvoir mettre le bon pointeur de sous arbre à null.

Spécification

```
-- nom : supprimer
-- sémantique :  suprime un noeud de l'arbre
-- paramètres :
  -- arbre : in  T_AB -- Arbre dont on veut suprimer un nœud
  -- element : in T_element -- Element à supprimer dans l'arbre
-- pré-condition :
-- post-condition :
-- Tests de la procédure :
procedure supprimer(arbre: in out T_AB;  element : in T_Element)
```

Raffinage

```
R0 : suprimer un noeud de l'arbre
R1 : comment "supprimer un noeud de l'arbre" -- arbre : in  T_AB;
element : in T_element
  if(suprimer_rec(arbre, valeur) = false) then
    raise noeud_absent
  end if;
R2 : comment "supprimer récursivement le noeud et ses antécédents" aka
"suprimer_rec"
          -- arbre : in  T_AB; element : in T_element
          -- return : boolean
  if(arbre=null) then
    return false;
  end if;
  if(arbre.all.gauche /= null and then arbre.all.gauche.all.element =
valeur) then
    arbre.all.gauche:=null;
    return true;
  end if;
  if(arbre.all.droit /= null and then arbre.all.droit.all.element =
valeur) then
    arbre.all.droit:=null;
    return true;
  end if;
  res := supprimer_rec(arbre.all.droit,valeur)
  if(not res) then
    res := supprimer_rec(arbre.all.gauche, valeur);
  end if;
  return res;
```

Fonctions et procédures de `p_arbre_genealogique` :

Récupérer les ancêtres de génération n d'un individu : (`get_ancetre_generation`) :

Afin de pouvoir obtenir les ancêtres de génération n d'un individu il nous faudra en premier lieu parcourir l'arbre généalogique pour se faire on utilisera `rechercher` de `p_arbre_binaire` afin de trouver la racine du sous arbre qui a l'individue comme élément. Une fois ce sous arbre trouvé on le parcours récursivement jusqu'à atteindre la profondeur attendue une fois la profondeur atteinte on ajoute l'élément à la liste de sortie et sort de la procédure.

Pour "récupérer la suite des ancêtres de génération n d'un individu" il nous faut simplement changer la structure de donné on passe d'une liste vers un arbre et on change le critère d'ajout à la structure de donné de sortie.

Spécification

```
--Nom : get_ancetre_generation
--sémantique : retourne la liste des ancêtres d'une
               --certaine generation d'un individue
--paramètres :
  -- arbre : T_AG
  -- persone : integer
  -- generation : integer
-- retour : L_Personne
-- préconditions : arbre/=null
-- postconditions :
function get_ancetre_generation(arbre : in T_AG;
                                persone : in integer;
                                generation : integer)
                                return L_Personne;
```

Raffinage

```
R0 : “retourne la liste des ancêtres d’une certaine generation d’un
individue”
R1 : Comment “retourne la liste des ancêtres d’une certaine generation
d’un individue”
    individu = find(arbre, get_dummy(personne));
parcourir récursivement jusqu’a la bonne profondeur et ajouter les
ancetre une fois a la bonne profondeur -- individu : in T_AG; resultat :
in out L_Personne; generation : in integer
R2 :  Comment “parcourir récursivement jusqu’a la bonne profondeur et
ajouter les ancetre une fois a la bonne profondeur ” aka
“get_ancetre_generation_rec”  -- individu : in T_AG; resultat : in out
L_Personne; generation : in integer
    if(individu = null) then return; end if;
    if(generation = 0) then
       add(resultat, get_element(individu))
       return;
    end if;
    get_ancetre_generation_rec(get_droit(individu), resultat,
generation-1);
    get_ancetre_generation_rec(get_gauche(individu), resultat,
generation-1);
```

Identifier le descendant d’une génération donnée pour un nœud donné (`get_descendant_generation`)

Pour récupérer le descendant d'un individue il va nous falloir trouver l'individue. On pourrait utiliser `rechercher` mais nous allons avoir besoin de garder le chemin emprunté pour trouver l'individue. On fait donc une recherche récursive. Une fois l'individu trouvé on fait redescendre sa profondeur. Quand la profondeur du descendant rempli le critère `profndeur_cible + profondeur_noeud  = profondeur_ancetre` on fait remonter l'élément du nœud.

Pour identifier la suite des descendants d'une génération donné pour un individu donné on utilise la même méthode il faut juste changer la structure des donnés de sortie et la condition de validité : `T_personne` devient `L_Personne` et `profndeur_cible + profondeur_noeud  = profondeur_ancetre` devient `profndeur_cible + profondeur_noeud  > profondeur_ancetre`

Spécification

```
-- Nom : get_descendant_generation
-- sémantique : retourne le descendant de n-ieme generation
            -- d'un individue
-- paramètres :
  -- arbre : T_AG
  -- personne : id
  -- generation : integer
-- retour : T_Personne
-- préconditions : arbre/=null
-- postconditions :
function  get_decendant_generation(arbre : in T_AG;
                                   persone : in T_persone;
                                   generation : integer)
                                   return T_Personne;
```

Rafinage

```
R0 : retourne le descendant de n-ieme generation d'un individue

R1 : comment "retourne le descendant de n-ieme generation d'un
individue"
    Parcourir récursivement l'arbre afin de trouver la profondeur de
l'ancetre et de trouver l'individue lors du depilement de la recursion -
-arbre : in T_AG, persone : in T_persone; descendant : out T_person;
generation, profondeur : integer
    return descendant

R2 : Comment "Parcourir récursivement l'arbre afin de trouver la
profondeur de l'ancetre et de trouver l'individue lors du depilement de
la recursion" aka
    Parcourir recursivement pour trouver la profondeur de l'ancetre
    Renvoyer le descendant si le noeud courrent est le descendant
R3 : Comment "Parcourir recursivement pour trouver la profondeur de
l'ancetre"
    if(arbre = null) then return -1; end if;
    if(get_racine_element(arbre) = persone) then
        return profondeur;
    end if;
    profondeur_ancetre:=
get_decendant_generation_rec(get_SA_droit(arbre), persone, descendant,
generation, profondeur+1);
    if (profondeur_ancetre = -1) then
        profondeur_ancetre:=
get_decendant_generation_rec(get_SA_gauche(arbre), persone, descendant,
generation, profondeur+1);
    end if;
R3 : Comment "Renvoyer le descendant si le noeud courrent est le
descendant"
            if (profondeur_ancetre = profondeur+generation) then
                descendant := get_racine_element(arbre);
            end if;
    return profondeur_ancetre;
```

Obtenir l'ensemble des individus qui n'ont qu'un parent connu. (`get_un_parent`) :

Cette fonction nécessite un algorithme qui parcours l'arbre et qui a chaque nœud vérifie les sous arbres pour avoir le nombre de parents. C'est le même principe

pour 2 ou 0 parents

Spécification

```
--Nom : get_un_parent
--sémantique : retourne la liste des individus ayant un
            -- seul parent
-- arbre : T_AG
-- retour : L_Person
-- préconditions : arbre/=null
-- postconditions :
function get_un_parent(arbre : in T_AG) return L_Person;
```

Rafinage

```
R0 : Obtenir l'ensemble des individus qui n'ont qu'un parent connu.
R1 : Comment "Obtenir l'ensemble des individus qui n'ont qu'un parent
connu" ? : in T_AB
    init_liste(liste)
    parcourir l'arbre récursivement aka "get_un_parent_rec" et remplir
liste
    return liste

R2 : comment "parcourir l'arbre récursivement aka "get_un_parent_rec""
if(arbre =null) then return; end if;

if(get_arbre_gauche(arbre)=null and get_arbre_droit /=null ) then
inserer_liste(fl, get_arbre_element(arbre))
end if;

if (get_arbre_gauche(arbre)/=null and get_arbre_droit =null ) then
inserer_liste(fl, get_arbre_element(arbre))
end if;

get_un_parent_rec(get_arbre_gauche, resultat);
get_un_parent_rec(get_arbre_droit, resultat)
```

Fonctions et procédures de `main` :

Le fonctionnements de main est assez simple. Les applications menu sont en

général constituée d'un loop qui contient un switch en fonction qui exécute des sous-programmes sur le modèle en fonction des entré de l'utilisateur

Raffinage

```
R0 : Faire un menu permettant de manipuler un arbre genalogique
R1 : Comment "Faire un menu permettant de manipuler un arbre
genalogique"? :
individu := Demander un individu;
Initialiser l'arbre( arbre : in T_AB; individu : in T_Personne)
loop
    choix := get_choix();
    exit when choix = choix_de_sortie;
    case choix is
        when un_choix =>
            demander les infos necessaire
            appeler la foctions correspondante
        when others =>
            afficher message d'erreur
    end case;
end loop
```

> Remarque : On ne detaillera pas le comment "demender les infos necessaire" et "appeler la fonction correspondante" car le principe est le meme pour chaque option et car ca rendrait le raffinage moin lisible

Tests

Méthodologie

Pour la réalisation des tests nous avons décidé de tester individuellement chaque package du programme avec sa propre procédure de test. On essaiera de suivre une méthode similaire aux tests unitaires.

Dans chaque programme de test on testera toutes les fonctions et procédure public associé au package. Cependant on exclura les getter et comparaison a null des tests en effet il est impossible de les tester correctement ces méthodes dans le langage.

On essaie de couvrir chaque cas d'utilisation des méthodes qu'il soit correcte ou non correcte. Cependant on considérera que les sous programmes seront appelé

avec respect pour les préconditions. On vérifiera les valeurs de retour des fonctions et les exceptions quand elles sont levé.

Typiquement sur les projet on utilise des libraire pour effectuer les tests. Elles permettent d'avoir un bon aperçu de la progression du test et de ce qui est passé ou pas. Afin de ne pas surcharger le projet et de ne pas poser de problèmes de dépendance, nous allons utiliser les instruction pragma de ada. Afin de faciliter la lisibilité du test on utilisera aussi ada text io qui nous permettra d'indiquer progressivement les progrès des tests.

Une autre fonctionnalité inclut dans les librairies de test est de pouvoir tester les exception en une instruction. Afin de pouvoir tester les erreurs nous avons du utiliser un bloc begin exception end afin de tester les erreurs.

Exemple

Test d'une fonction

```
tp := get_decendant_generation(arbre, get_id(p4),2);
put("descendant de 5 de deuxieme generation : "); put(tp); new_line;
pragma Assert(get_id(tp) = 1);
```

Test d'une erreur

```
begin
    l := get_sucession_decendant_generation(arbre, 90, 2);
    raise TEST_ECHOUE;
exception
    when  p_AB_Person.NOEUD_ABSENT_ERROR=>
        put_line("Erreur attendu levée");
end;
```

Couverture

package	test
`p_personne`	✓ : `test_personne.adb`
`p_arbre_binaire`	✓ : `test_arbre_binaire.adb`
`p_arbre_genealogique`	✓ : `test_arbre_genealogique.adb`
`p_front`	x : *Fait uniquement de l'IO est donc ne peut pas être testé*
`p_linked_list`	x : *On fait confiance au package développé dans les TP qui ont été testé à ce moment*

Difficultés :

Difficultés rencontrées :

Voici une liste des difficultés principales qui ont été rencontré dans la conception du projet :

- La gestion rigide des dimensions des structures de donné d'ada rends leur manipulation compliquée. Les strings sont demande beaucoup de manipulations pour faire des choses très simple et les arrays d'ada ne laisse pas d'autre options que les surdimensionner ce qui a des couts en ressource
- L'absence de bibliothèque de tests a été un problème pour structurer les tests correctement.
- Le passage des structures de donnés en privé à grandement compliqué les préconditions et postconditions en effet en ada elles n'ont pas accès aux structures.
- L'arbre binaire en privé a rendu la conception de l'arbre généalogique compliqué en effet le package n'a pas la capacité de modifier la structure. Et il n'existe de structures protégé comme dans un langage comme java.
- L'absence des fonction lambda et de l'orienté objet rend difficile la non-répétitivité du code dans certains cas.

Solutions appliquée :

Ci-dessous sont détaillées les solutions aux problèmes précédemment soulevé :

- Bien qu'il aurait été possible d'utiliser les types de base d'ada. Il a été décidé d'utiliser le `T_linked_list` qui avait été précédemment développé dans un TP et le type `unbounded_string` de la bibliothèque standard d'ada.
- La solution employée est détaillé dans la section dédiée aux tests.

- L'ajout de getter nous a permis d'évaluer beaucoup plus facilement les préconditions et postcondition.
- La mise en place de setter et de getter a permis aux algorithme d'arbre généalogique de parcourir l'arbre. Ce problème est de taillé dans la partie choix technique.
- Ce problème a nécessité plus de temps sur la conception afin de réduire au maximum les dupliquât. Cependant on a fait attention à ne pas trop sacrifier la lisibilité pour reduire le volume du code

État du projet :

État du projet :

Le projet dans l'état actuel est complet et répond aux attentes du sujet du projet. La couverture de tests couvre toutes les fonctions du modèle. Chaque fonctionnalité majeure a été raffiné et développée.

Amélioration possible :

- Étendre les commentaires dans le code.
- Ajouter un système de sérialisation qui augmenterais l'intérêt du projet.
- Ajouter des fonctionnalité supplémentaire pour la manipulation de l'arbre et des individus.

Bilan personnel :

Intérêt personnel :

Bien que le sujet lui-même ne soit pas extrêmement complexe, il apporte suffisamment de question et pour permettre de mettre en place et de travailler sur des méthodes de conception et des méthodes de gestion de projets.

J'avais déjà travaillé sur un certains nombre de projets en informatique qui m'ont appris à programmer, à concevoir et à gérer un projet. Cependant ces projets sont des fois pas assez encadrer ou trop cour pour mettre en place une rigueur dans la méthodologie. Ce projet nous permets et nous laisse le temps de mettre en place cette rigueurs et de nous entrainer à être rigoureux.

Ce projet m'a aussi permis de travailler sur les testes qui ont pour longtemps été ma bête noire.

Temps de travail

En addition des 32 heures de travail qui nous ont été alloué pour le projet durant les heures de j'ai rajouté à peu près 8/10 heures de travail personnels.

Répartition du temps

Voici la répartition du temps de travail pour ce projet :

- 55% : Conception
  - 25% : Conception des structures de donnés
  - 30% : Raffinage
- 25% : Programmation
- 20% : Rapport

On peut remarquer le rapport a été un charge de travail importante pour moi. Ce qui n'est pas particulièrement étonnant en considérant que cela fait partie de mes faiblesses.

Enseignement tiré du projet

Un des principale enseignement tirés de ce projet est l'importance du temps accordé à la conception du projet. En effet du temps qu'on pourrait considérer comme gaspiller est en fait du temps qui permet de faciliter l'implémentation et qui éviter les problèmes qui peuvent apparaitre lors d'un développement qui n'est pas guidé.