Rapport de Projet de Méthodologie de la Programmation

Développement d'un programme de gestion d'arbre généalogique

Manuel de :
M. Diego Rodriguez
AIT-AMEUR
le : 31/01/2021

Enseignants :
M. Yamine
M. Neeraj SINGH

Résumé

Objectif du rapport

Ce rapport a pour objectif de vous présenter le projet réalisé dans le cadre du cours de méthodologie de la programmation à l'ENSEEIHT à Toulouse. Il abordera toutes les parties de la réalisation du projet, allant de la présentation des choix réalisés à la présentation des principales méthodes, types et structures de données utilisées. Il présentera aussi la démarche adoptée, la conception choisie et les difficultés rencontrées. L'architecture sera aussi présentée avec pour finir un bilan technique et personnel sur ce projet et plus généralement ce module. Un manuel utilisateur est fourni en annexe : il décrit comment utiliser les programmes développés.

Introduction

Le projet consiste à développer un système de gestion d'arbre généalogique. Un arbre généalogique est une représentation graphique de la généalogie ascendante ou descendante d'un individu. Cet arbre est implémenté sous la forme d'un arbre binaire générique, une structure de données qui peut se représenter sous la forme d'une hiérarchie dont chaque élément est appelé nœud, et le nœud initial étant la racine. Dans notre arbre généalogique, chaque noeud sera appelé un individu qui possède un identifiant unique, un prénom, un nom, un sexe, une date de naissance, une date de décès si elle existe et la dernière adresse connue de l'individu. Le programme comporte des tests et un menu pour permettre à l'utilisateur d'interagir avec le programme. Le langage utilisé pour effectuer ce projet est Ada, qui a été utilisé pour tous les autres travaux pratiques du cours.

Table des matières

Principaux types de données et justification des choix de ces structures

Types de données

Pour pouvoir construire cet arbre généalogique, je me suis appuyé sur le TAD d'un arbre binaire que nous avons pu faire lors du TP16 de cette matière. Dans cet arbre, nous l'avions implémenter pour qu'il trie automatiquement l'arbre en fonction de la valeur insérée. J'ai dû modifier cela car un arbre généalogique n'a pas à être trié en fonction des nouvelles valeurs. Je tiens à préciser que tout le long de la documentation ou du code, un nœud signifie la même chose qu'un arbre, car chaque nœud est un petit arbre binaire. Le seul nœud qui se différencie des autres est l'arbre racine qui n'est le sous-arbre d'aucun autre nœud.

- Pour le module de l'arbre binaire:

J'ai créé le pointeur de type T_Arbre_Bin qui pointe sur l'enregistrement T_Noeud. T_Noeud est un enregistrement qui contient l'élément générique de l'arbre, un pointeur vers un sous-arbre droit et un pointeur vers un sous-arbre gauche.

L'élément du nœud est de type T_Element et il caractérise la donnée du nœud courant.

- Pour le module de l'arbre généalogique :

Le type T_Element est donc instancié avec le type T_Individu dont je parle ci-dessous. L'arbre généalogique est un arbre binaire

- Pour le module de l'individu :

J'ai créée un enregistrement d'informations de type T_Informations qui contient toutes les informations relatives à un individu telles que son nom, son prénom, son sexe, sa data de naissance, sa date de décès ou encore son adresse. Un pointeur de type PT_Informations pointe sur cet enregistrement. Une fois cet enregistrement créé, j'ai fabriqué aussi l'enregistrement d'un individu de type T_Individu qui est caractérisé par son identifiant de type T_Identifiant et par ses informations de type PT_Informations.

Justifications des choix de ces structures

Le module p_arbre_bin a été mis en place avec un élément, un sous-arbre droit et gauche afin de pouvoir mettre en place un chaînage entre tous les nœuds de l'arbre grâce aux pointeurs. Ceci est similaire à une liste chaînée, sauf que nous avons deux pointeurs par nœud.

Le type T_Element est générique afin de pouvoir instancier d'autres types d'arbres binaires qui ont chacun des utilisations uniques. Dans notre cas, nous utilisons un arbre binaire de type arbre généalogique.

Le type d'élément choisi T_Individu permet une modularité des informations stockées sur l'individu. Dans mon cas, j'ai instancié le type T_Identifiant avec un identifiant de type Integer, alors que l'on pourrait aussi avoir un identifiant de type String. J'ai choisi d'utiliser seulement l'identifiant de type Integer car cela me permettait de gérer l'unicité de l'identifiant de manière plus simple. L'utilisateur pourra aussi retenir plus facilement l'identifiant et aura seulement à l'inscrire afin de pouvoir interagir avec l'individu (le nœud) voulu. A partir de maintenant, je considère qu'un nœud peut aussi se traduire par le mot "individu", car dans notre cas, un nœud représente un individu.

Pour l'individu, j'ai choisis de pointer l'enregistrement des informations afin de pouvoir mettre ce pointeur à null s'il n'y a aucune information. Cela m'a permis de ne pas utiliser en mémoire de l'espace si les données ne sont pas renseignées. J'ai pris le soin de mettre beaucoup d'informations pour que chaque individu puisse vraiment être unique. Si nous n'avions que le nom et le prénom, nous aurions des probabilités d'avoir des doublons dans l'arbre et ce n'est pas très intéressant pour une personne souhaitant retrouver son arbre généalogique. L'identifiant de l'individu est unique et j'ai choisi de le mettre en Integer car il va me permettre de faire des recherches et comparaisons par identifiant, alors que, au contraire, on ne touche pas au pointeur d'informations pour rechercher un individu. On pourra donc faire référence à un individu sans devoir donner toutes ses informations.

Principaux algorithmes et raffinages

Algorithmes

Je vais poster ici seulement quelques algorithmes principaux car je ne peux pas tous les mettre, il y en aurait trop. Je vais les présenter et les expliquer. J'ai raffiné littéralement 100% du code dans mon projet, alors n'hésitez pas à aller voir les sources pour plus d'informations détaillées ou pour voir tous les raffinages existants. Les algorithmes les plus simples ne seront pas représentés. Seuls les algorithmes complexes sont assez intéressants pour être expliqués. Je ne vais pas réécrire les raffinages sur ce rapport car vous pourrez vous y référer dans le code. Je vais seulement expliquer le fonctionnement des algorithmes. Je ne vais présenter que les algorithmes les plus complexes de l'arbre généalogique qui est le principal intérêt de notre projet, sinon le rapport serait beaucoup trop gros.

Certaines méthodes de l'arbre généalogique sont simples et se font en quelques instructions voire une seule instruction qui consiste à appeler une méthode de l'arbre binaire. Je vais citer ces méthodes sans les expliquer dans le détail car la complexité de l'arbre binaire ne nous intéresse pas ici.

procedure creer(F_Arbre : out T_Arbre_Bin ; F_Individu : in T_Individu)
procedure ajouterParent(F_Arbre : in out T_Arbre_Bin ; F_Parent : in T_Individu ; F_PereOuMere : in Boolean)
procedure ajouterPere(F_Arbre : in out T_Arbre_Bin ; F_Pere : in T_Individu)
procedure ajouterMere(F_Arbre : in out T_Arbre_Bin ; F_Mere : in T_Individu)
function nombreAncetres(F_Arbre : in T_Arbre_Bin ; F_Individu : in T_Individu)
procedure supprimerNoeudEtAncetres(F_Arbre : in out T_Arbre_Bin ; F_Individu : in T_Individu)
procedure modifierIndividu(F_Arbre : in out T_Arbre_Bin ; F_Individu_Source : in T_Individu ; F_Individu_Cible : in T_Individu)
function equivalent(F_Individu1, F_Individu2 : in T_Individu) return Boolean

- Procédure *identifierAncetres* :

Cette procédure permet d'identifier les ancêtres d'une génération donnée pour un nœud donné. Il faut d'abord vérifier que l'arbre n'est pas vide. S'il est vide, on lève l'exception "arbre_null". Sinon, on peut continuer. On regarde si la génération indiquée est de 0. Si elle est de 0, cela veut dire que l'utilisateur recherche les ancêtres d'un individu correspond à lui-même, donc il n'y a pas d'ancêtre et on lève l'exception "pas_ancetre".

Sinon, on peut continuer. Cette procédure est toujours appelée avec un compteur qui démarre à 0 et qui s'incrémente à chaque appel récursif de cette fonction. On compare si le

compteur est égal à la génération demandée pour vérifier si l'on se situe à la génération qu'à demandé l'utilisateur. Si on est dans la bonne génération, on peut afficher l'individu courant. Sinon, on appelle récursivement la méthode pour le sous-arbre gauche puis pour le sous-arbre droit sans oublier d'incrémenter le compteur de 1 à chaque appel. Si ces appels récursifs renvoient un arbre vide, on lève une exception qui ne va rien faire afin de pouvoir continuer la recherche jusqu'au bout. La récursivité va répéter toutes ses actions jusqu'à avoir parcouru tout l'arbre et les ancêtres seront donc renvoyés.

- Procédure *ensembleAncetres* :

Cette procédure permet d'afficher l'ensemble des ancêtres jusqu'à une certaine génération d'un nœud donné. Tout d'abord on vérifie si l'arbre est vide. S'il est vide, on ne fait rien. J'ai choisi de ne rien faire pour gérer l'arbre vide de différentes manières afin d'utiliser toutes les possibilités de codage. S'il n'est pas vide, on vérifie si la génération est de 0. Si elle est de 0, cela veut dire que l'utilisateur recherche les ancêtres d'un individu pour sa propre génération, ce qui ne renverra rien, donc on renvoie une erreur "pas_ancetre". Sinon, on vérifie d'abord que si le compteur est à 0 (première occurrence de l'appel de la méthode) et qu'il est différend de la génération, alors on appelle récursivement la procédure elle-même pour le sous-arbre gauche puis pour le sous-arbre droit. Si nous ne sommes pas à la première occurrence de la méthode, alors on affiche l'individu courant et on vérifie si l'on se situe à la génération donnée. Si nous n'y somme pas, on continue d'appeler récursivement la procédure à gauche et à droite jusqu'à avoir parcouru tout l'arbre.

- Procédures *identifierDescendant* et *ensembleDescendants* :

Ces deux procédures fonctionnent de la même manière que les deux précédentes, mais font la recherche du sur-arbre. Cette manière de faire la recherche est définie dans la procédure recherche de l'arbre binaire. De plus, on appelle récursivement la procédure qu'une seule fois car il n'y a qu'un sur-arbre en comparaison avec les deux sous-arbres.

- Procédure *afficherArbreGen* :

Cette procédure fait un affichage spécial pour l'arbre généalogique, avec d'abord le père (sous-arbre gauche) puis la mère (sous-arbre droit). J'ai décidé de mettre un compteur qui s'incrémente à chaque appel récursif de cette procédure afin de décaler l'affichage. En effet, plus on avance dans l'arbre généalogique, plus l'individu est né il y a longtemps, et ainsi plus le décalage vers la droite est affiché. Cela permet d'avoir un affichage en escalier comme montré ci-dessous.

- Procédure *listeAucunParent* :

Cette procédure permet de lister l'ensemble des individus dont les deux parents sont inconnus. On vérifie d'abord si l'arbre est vide. S'il l'est, alors on affiche qu'aucun individu n'a aucun parent connu. Cela permet une nouvelle fois de tester l'arbre null d'une autre manière afin de vous montrer que je maîtrise toutes les possibilités. Si l'arbre n'est pas vide, on vérifie si le sous-arbre gauche et droit de l'individu courant sont vides. Si c'est le cas, on affiche l'individu car cela signifie qu'il n'a aucun parent (sous-arbre) de connu. Ensuite, on appelle récursivement cette méthode d'abord dans le sous-arbre gauche s'il n'est pas vide puis dans le sous-arbre droit s'il n'est pas vide.

- Procédures *listeUnSeulParent* et *listeDeuxParents* :

Ces deux procédures fonctionnent de la même manière que "listeAucunParent" sauf que l'on vérifie pour la première si un des deux sous-arbre est vide avec un ou exclusif. Pour la seconde, on vérifie que les deux sous-arbres ne sont pas vides.

- Procédures *identifierAncetrePaternel* et *identifierAncetreMaternel* :

Ces deux procédures sont équivalentes à la procédure "identifierAncetres" sauf que l'on affiche seulement les parents du côté paternel pour "identifierAncetrePaternel" et seulement les parents du côté maternel pour "*identifierAncetreMaternel".*

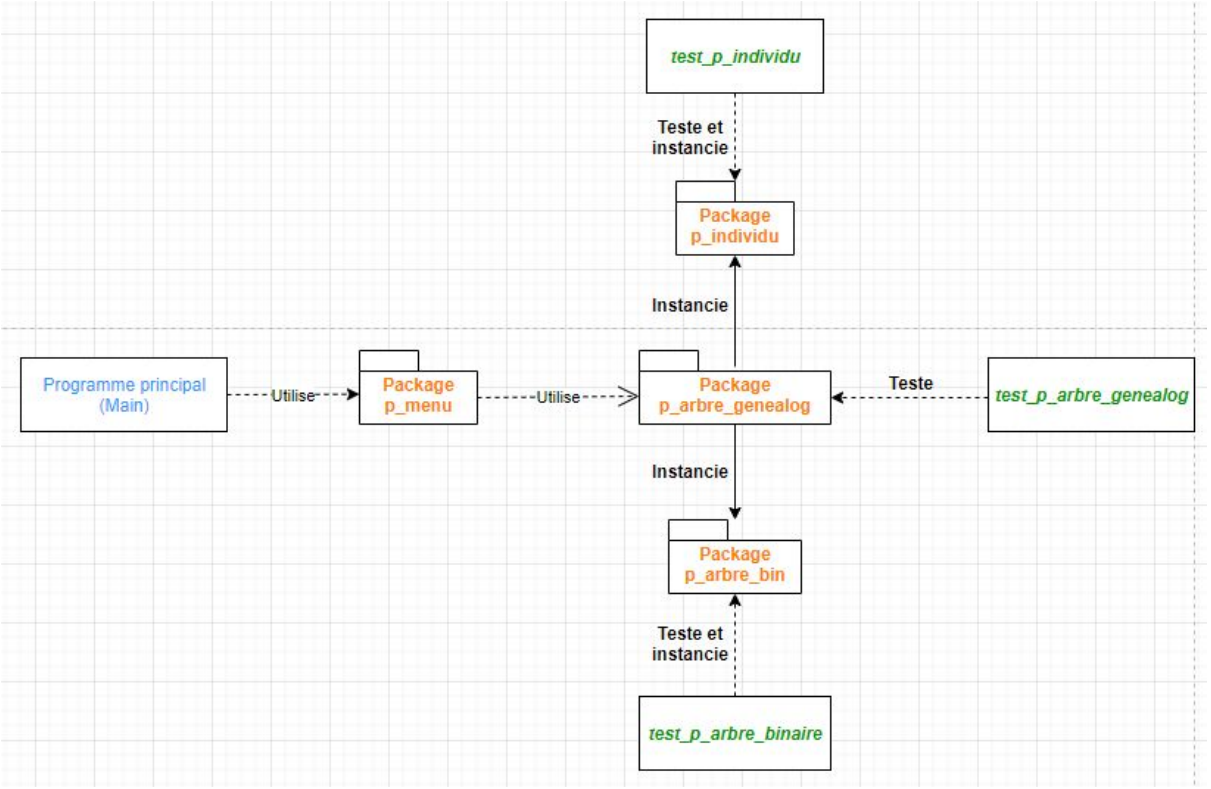
Pour conclure avec les algorithmes, il y a aussi de la complexité dans les autres méthodes et surtout dans l'arbre binaire, mais je ne peux pas tout détailler ici, cela dépasserait la limite de pages autorisée. Vous pouvez voir dans le code source l'intégralité des raffinages qui permet de vous faire comprendre en détail toutes les autres méthodes. J'ai ici seulement expliqué toutes celles de l'arbre généalogique, le module le plus important pour ce projet.

Raffinages

Comme je l'ai déjà dit, tous les raffinages sans exception sont présents sous forme de commentaire dans le code source des différents modules. Je vous invite à vous y référer pour pouvoir les regarder en détail et comparer avec mes explications ci-dessus. J'ai discuté avec mes enseignants, et ils m'ont autorisé à ne pas mettre mes raffinages car ils sont déjà présents dans le code. J'ai choisi dès le départ de faire les raffinages puis de rajouter le code Ada une fois ceux-ci terminés donc vous les trouverez entièrement dans tous les fichiers .adb. De même pour les spécifications des méthodes, elles se situent toutes dans les fichiers .ads.

Architecture de l'application

Ci-dessous un schéma représentant totalement l'architecture de l'application. En orange, vous pouvez voir les paquetages qui représentent le fichier de spécification ".ads" et le fichier contenant le corps des modules ".adb". Les tests sont représentés en vert et le programme principal en bleu. Les tests et le programme principal sont des programmes exécutables.



Le programme principal effectue un appel vers la seule méthode publique du menu qui permet d'initialiser le programme et de tout mettre en marche. L'utilisateur pourra ainsi interagir avec le programme. Ceci permet de faire en sorte que les appels à des méthodes non souhaitées soient impossibles.

Le paquetage *p_menu* utilise le paquetage *p_arbre_genealog* ainsi que ceux qu'il instancie. Chacune de ses méthodes demande à l'utilisateur une saisie. Ce paquetage fait donc office d'interface entre l'utilisateur et le programme qui s'exécute.

Le paquetage *p_arbre_genalog* instancie les paquetages *p_individu* (pour avoir la représentation d'un individu) et *p_arbre_bin* (qui représente la structure d'un arbre binaire). Ainsi, un arbre généalogique est un arbre binaire dont chaque élément est un individu. Ce paquetage possède aussi plusieurs procédures qui permettent de faire les traitements spécifiques à un arbre généalogique.

Le paquetage *p_arbre_bin* représente un arbre binaire non trié. Chaque nœud possède un élément générique. Cet arbre est donc instanciable de différentes manières. Dans notre cas, on l'instancie sous la forme d'un arbre généalogique.

Le paquetage *p_individu* représente l'élément d'un nœud, c'est-à-dire l'individu avec toutes ses informations associées et son identifiant unique. Cela permet d'ajouter et de modifier des informations à un individu plus facilement.

Tous les paquetages de test sont automatiques et ne nécessitent pas d'action de l'utilisateur. Il suffit de les compiler et de les exécuter comme expliqué dans le manuel utilisateur. Il suffit ensuite de regarder l'affichage sur la console pour voir s'ils passent tous. Chaque paquetage de test teste un module spécifique.

Durant l'entièreté du projet, l'objectif a été de faire en sorte que chaque paquetage dépende d'un nombre minimal d'autres paquetages, afin de favoriser la modularité et de permettre une bonne encapsulation des données et des méthodes.

Si vous voulez obtenir beaucoup plus de détails de chaque module, vous avez des captures d'écrans disponibles dans le dossier du projet remis en annexes dans le répertoire "Architecture_UML". Vous y trouverez le détails des types de données et des méthodes utilisées;

Choix réalisés

J'ai réalisé de nombreux choix durant ce projet que je vais vous énumérer sans ordre particulier. Cela va vous permettre de mieux comprendre mon programme et ma réflexion vis-à-vis de celui-ci.

Pour rechercher les descendants d'un individu, j'ai choisi de pouvoir indiquer via un booléen dans la méthode "recherche" de l'arbre binaire si on veut renvoyer le sur-arbre. J'aurais pu aussi développer une méthode "getSurArbre" dans l'arbre généalogique. J'ai décidé de ne pas mettre un pointeur vers l'arrière sur chaque nœud de l'arbre, cela ne correspondait pas à l'implémentation correcte d'un arbre binaire. Cette solution a été très bonne car le nombre d'ancêtre n'est pas immensément grand. Dans un autre cas, il aurait été préférable en rapidité d'exécution d'avoir un pointeur vers un sur-arbre, mais cela aurait été intéressant pour un autre projet (même si cette solution est gourmande en mémoire système).

Parfois, le code n'est pas logique et fait quelque chose de complexe pour pas grand chose. Ce choix a été fait afin de pouvoir utiliser toutes les notions travaillées au sein du module durant les cours, les TD et les TP.

Comme discuté avec Monsieur Neeraj Singh, j'ai décidé de laisser tous les raffinages dans le code source et de ne pas les mettre dans le rapport. J'en avais discuté avec un enseignant et il a validé cette manière de faire.

L'identifiant d'un individu est un entier. Cela m'a permis une facilité d'utilisation notamment pour la comparaison et la recherche d'individus. J'ai décidé de ne pas incrémenter automatiquement l'identifiant et de laisser le choix à l'utilisateur de mettre ses propres identifiants s'il préfère par exemple incrémenter l'identifiant de 10 pour chaque nouvelle génération. Il aurait été possible d'instancier l'individu avec un identifiant de type String comme vous pouvez le voir dans le code source. J'ai décidé de ne pas le faire, je l'ai donc laissé en commentaire.

Comme beaucoup d'exceptions sont levées et souvent pour la même raison (telle qu'un arbre vide). J'ai mutualisé le traitement de presque toutes les exceptions dans une procédure du menu afin de récupérer ici toutes les exceptions et de faire un traitement en conséquence. Parfois, j'ai choisi de traiter les exceptions plus à la racine afin de faire des traitements très spécifiques.

J'ai laissé la possibilité à l'utilisateur de ne pas saisir d'information pour un individu à part son identifiant. Les informations inconnues sont automatiquement remplacées par le champ "n/a" lors de l'affichage.

J'ai choisi d'utiliser le type "Unbounded_String" pour les informations de l'utilisateur afin que toutes les entrées utilisateur soient facilitées sans avoir à connaître au préalable la taille de la prochaine chaîne de caractères que l'utilisateur va rentrer.

Je n'appelle pas toujours l'exception "arbre_null" afin de pouvoir vous montrer toutes les manières possibles et imaginables de traiter les exceptions dans tous les endroits possibles du code. Cela était pour vous montrer que je maîtrise cette notion dans toutes ses formes. Par exemple, je l'utilise pour les méthodes pour modifier partiellement et totalement un individu, mais pas toujours pour les autres. De même pour d'autres exceptions, je ne les récupère pas toujours au même endroit pour bien vous montrer que je sais les récupérer dans plusieurs endroits du code pour vous montrer que je maîtrise le champ du domaine des exceptions et ce que ça implique.

J'ai fait en sorte que la saisie se fasse obligatoirement avec un entier (sauf si on demande la saisie d'informations d'un individu), sinon une erreur se produit et le programme se termine avec un message personnalisé. C'est un choix volontaire pour pouvoir laisser à GNAT lever ses propres exceptions concernant les "get_line".

Je n'ai pas testé le menu car tout se base principalement sur les entrées utilisateur que l'on peut vérifier directement lors de l'exécution du programme.

J'ai volontairement moins commenté les tests car c'est seulement des utilisations de méthodes en créant des valeurs souhaitées. Il n'y a quasiment pas d'algorithme derrière ceux-ci.

Je n'indique pas dans l'affichage à quoi correspond chaque valeur (Nom, Prénom, etc…) pour ne pas alourdir l'affichage.

J'ai absolument tout raffiné dans le code source. J'espère que vous le prendrez en considération.

J'ai choisi de coder intégralement en Français. Ce n'est pas une très bonne pratique mais nous avons fait tous nos cours en Français et je voulais que ce soit ressemblant à ce que nos enseignants nous ont appris. Les prochains projets que j'effectuerai dans ma vie seront en Anglais.

J'ai essayé d'envisager tous les cas possibles qui pourraient induire des erreurs lors de l'exécution du programme. Afin de rendre le programme robuste, j'ai décidé d'utiliser uniquement la programmation défensive. Un utilisateur ne peut pas faire buguer le programme car une erreur ou une interdiction lui sera toujours levée. Cette partie a été très longue à développer mais c'est le plus important pour qu'un programme soit fonctionnel. J'ai énormément été minutieux pour la conception de ce projet pour qu'il soit le plus lisible possible et extrêmement facile à comprendre et impénétrable en termes d'erreurs. J'espère que ces efforts seront pris en compte.

Je n'ai pas réglé tous les gnat warnings car parfois ça me posait plus problème qu'autre chose de les réparer. J'en ai donc laissé quelques-uns (mais très peu).

Pour finir, j'ai choisi de ne pas faire de liste chaînée car cela était inutile (à part pour les tests) pour la réalisation du projet. Il n'est pas demandé de récupérer l'affichage des résultats. J'en ai discuté avec les enseignants du module, et ils m'ont dit que cela n'était pas obligatoire et pas pénalisant. Bien sûr, pour une amélioration future du projet, il pourrait être intéressant d'en implémenter une, mais j'arrive à répondre à tout le sujet sans liste chaînée.

Démarche adoptée pour les tests

Grâce aux tests, j'ai pu premièrement savoir si les sous-programmes développés fonctionnent tels que prévu. Une fois cela fait, ces tests servent maintenant de tests de non-régression. Cela m'a permis de vérifier que la refactorisation ou l'amélioration de mon code ne générait pas de nouvelles erreurs. Je n'ai pas développé les tests très tôt car je me suis plutôt focalisé sur la bonne réalisation du programme qui était le point principal du projet. Néanmoins, je me suis noté de côté tous les cas de test que je devais effectuer, afin de pouvoir coder ceux-ci le plus rapidement possible.

Voici un tableau de correspondance qui montre quels tests sont exécutés pour chaque paquetage.

Paquetage testé	Programme de test
p_arbre_bin	test_p_arbre_binaire
p_arbre_genealog	test_p_arbre_genealog
p_individu	test_p_individu

Avant de faire ce projet, j'avais déjà eu l'occasion de faire des tests unitaires, notamment en java, j'ai donc essayé de reproduire ce concept en Ada. Les outils pour tester ne sont pas aussi développés qu'en Java, mais j'ai fait avec et cela s'est tout de même très bien passé.

J'ai donc décidé de tester les trois paquets "p_individu", "p_arbre_binaire" et "p_arbre_genealogique" qui sont les principaux programmes à tester. Il n'a pas été

nécessaire de tester le paquetage “p_menu” car il fonctionne grâce aux trois autres paquets et fait seulement des interactions avec l’utilisateur, qui se vérifient lors de l’exécution du programme. Étant donné les trois autres paquets testés, le paquetage “p_menu” par conséquent ne peut pas avoir d’erreur d’algorithmique. Pour chaque test, j’ai d’abord saisi des jeux de valeurs puis j’ai vérifié que le programme s'exécute correctement pour chaque manière d’utiliser les procédures disponibles. L’utilisateur verra à l’écran “PROCEDURE OK” pour chaque procédure ou “EXCEPTION OK” pour chaque exception.

J’ai aussi testé les exceptions intervenant dans chaque paquetage une à une afin de vérifier que les levées d’exceptions s’effectuent au bon moment.

Pour cela, je me suis servi partout où cela était possible de l’outil “pragma Assert”. Le test de l’arbre binaire n’utilise pas cet outil car nos enseignants nous ont dit de ne pas le tester, j’ai donc préféré passer moins de temps à le faire et j’ai récupéré les tests manuels que j’avais déjà fait pour le TP16 concernant les arbres binaires triés.

Étant donné que le programme avait pour but principal d’interagir avec l’utilisateur comme indiqué dans les choix réalisés, la plupart des procédures affichent le résultat directement à l’écran et ce n’était pas possible de tester ceci automatiquement. J’ai donc décidé d’écrire le résultat attendu à l’écran, et l’utilisateur doit vérifier manuellement que le résultat de l’appel d’une procédure correspond bien au résultat attendu. De même, lorsque le programme lit directement des informations auprès de l’utilisateur, je n’ai pas trouvé de moyen d’automatiser les tests de cette partie. Ces tests sont donc à faire par l’utilisateur durant l’exécution du programme principal. Bien sûr, j’ai tout de même veillé à faire en sorte que l’exécution du programme principal fonctionne parfaitement.

Difficultés rencontrées et solutions adoptées

Globalement, j’ai eu peu de difficulté pour mener à bien ce projet car j’étais déjà à l’aise dans la conception d’un programme informatique et je connaissais déjà un peu l’Ada.

Lors de l’écriture des tests, je me suis rendu compte que pour pouvoir tester les procédures qui font un affichage, il aurait été pertinent de fabriquer un module de type liste chaînée afin de pouvoir stocker ses affichages puis de les renvoyer sous forme d’une chaîne de caractères afin de pouvoir les comparer et faire des tests plus automatiques. Ici, l’utilisateur doit vérifier à la main que le résultat attendu correspond au résultat obtenu. Par manque de temps, j’ai décidé de ne pas mettre en place cette liste chaînée et de passer plus de temps sur les tests pour la comparaison manuelle (donc des tests moins automatisés).

J’ai trouvé cela difficile de faire un projet en entier de A à Z car nous n’avons pas forcément vu la démarche de conception en cours. Heureusement, grâce au travail d’équipe, nous avons pu beaucoup progresser ensemble pour une bonne conception du projet.

Il a été difficile de tester mes programmes car il n’existe pas de framework vraiment adapté pour ceci. J’ai donc dû beaucoup réfléchir lors de l’écriture des tests pour savoir comment les implémenter le plus proprement possible. Lors de l’écriture des tests, je me suis rendu compte que pour les modules qui affichent une liste d'individus. Une utilisation du module Liste Chainée aurait été pertinente. Le programme se serait comporté de la même manière, et l’ajout de complexité aurait pu permettre d’utiliser les tests automatiques plutôt qu’une comparaison manuelle entre deux chaînes de caractères affichés à l’écran. Par manque de temps, j’ai tout de même choisi de ne pas mettre en place la liste chaînée et d’utiliser des tests pas entièrement automatisés.

Il a été difficile pour moi de finir le projet dans les temps car je pense qu'il manquait des séances et peut-être une semaine pour avoir plus de temps pour le finaliser. Nous étions dans une période d'examen à la fin. Cela a rendu la fin du projet très compliquée. Pour réussir à surmonter tout ça, j'ai dû énormément travailler et redoubler d'effort.

Bilan technique

Etat d'avancement du projet

Finalement, je suis assez fier de moi car j'ai pu énormément avancer dans le projet, j'y ai mis vraiment énormément d'énergie et de temps. Le projet est totalement terminé et toutes les implémentations demandées dans le sujet ont été respectées. J'ai réussi à tout finir à temps. J'ai raffiné et commenté l'intégralité du code.

Il me manque seulement environ une semaine pour pouvoir peaufiner à 100% le projet et le rendre complètement lisse sans aucun détail qui traine, mais cela m'aurait pris un temps exponentiel et j'ai décidé de ne pas le faire.

Perspectives d'amélioration ou d'évolution

Il reste quelques détails à régler comme l'amélioration des pré et des post conditions à respecter afin que l'utilisateur ne puisse en aucun cas tricher dans le programme ou faire des choses non souhaitables. J'ai sûrement laissé passer des erreurs sans faire exprès. Ma factorisation du code n'est pas parfaite car je ne maîtrise pas encore cette notion en Informatique mais j'ai factorisé du mieux que je pouvais. Quelques méthodes sont publiques alors qu'il serait sûrement préférable de les laisser en privée, il faudrait revoir ce point pour être sûr que toutes les méthodes sont bien encapsulées. Il aurait été possible d'ajouter des frères et sœurs dans l'arbre mais ce n'était pas demandé et cela aurait pris trop de temps.

Pour une utilisation future, il aurait été intéressant de mettre en place une liste chaînée pour stocker les données des résultats des opérations sur l'arbre généalogique. En effet, comme expliqué précédemment, ce n'était absolument pas nécessaire pour ce projet, mais si on voulait le développer avec une plus grande ampleur, il aurait été important de mettre en place cette structure de données.

De plus, il est aussi possible d'ajouter des fonctionnalités différentes, mais cela serait pour continuer le projet dans une plus grande ampleur si jamais on voulait le sortir publiquement pour des utilisateurs externes depuis par exemple une plateforme open source telle que GitHub.

Bilan personnel

Intérêt

J’ai beaucoup aimé le projet. Le sujet était très intéressant car il s'applique sur un arbre généalogique, et nous avons tous déjà essayé de reconstruire notre arbre généalogique. J’ai donc eu un peu de nostalgie pendant ce projet. C’était très intéressant de pouvoir remettre tout ce que nous avons utilisé dans les travaux pratiques à profit afin de pouvoir concevoir et implanter ce projet. Nous avons été plusieurs au début pour faire la conception et la réflexion.

Ensuite, nous avons dû nous séparer pour chacun coder de son côté. À partir de ce moment, j’ai un peu stressé car j’ai eu peur de ne pas réussir mais au final j’ai réussi plutôt facilement le projet. En effet, je ne maîtrisais pas bien l’Ada et je préfère le Java donc j’ai eu un peu peur. Plus le projet avançait, plus je voulais traiter de cas, plus je voulais tester, plus je voulais changer des choses pour les améliorer dans les détails. Je suis un peu perfectionniste et cela m’a coûté beaucoup de temps et d’énergie. De ce fait, j’ai moins apprécié faire ce projet à la fin car j’avais en parallèle des partiels et d’autres projets à faire. Mais je ne me suis pas démotivé pour autant et mon travail acharné m’a permis de finir complètement ce projet.

Je suis maintenant extrêmement satisfait de voir que mon projet marche à la perfection, la sensation est très agréable.

Répartition du temps

Chaque étape du projet prend un temps plus ou moins grand que je vais détailler ci-dessous.

- Conception : 50% du temps
    - 15% sur les types de données
    - 35% sur la conception des modules et sur le raffinage des algorithmes
- Implantation : 25% du temps
- Tests : 10 % du temps
- Mise au point, amélioration, petites touches personnelles : 5% du temps
- Rédaction du rapport et du manuel utilisateur : 10% du temps

Je n’ai pas compté mes heures, mais j’ai passé énormément de temps en plus des heures de cours afin de faire ce projet. Je pense avoir passé presque 80 heures sur ce projet. Je pense qu’il aurait été préférable de le faire à 2 pour pouvoir déléguer les tâches longues et rébarbatives telles que certains bouts de code. J’ai tout fait pour suivre au mieux les enseignements que l’on a eu sans négliger la conception et les raffinages. C’est pour cela que le projet m’a pris autant de temps mais au moins il est très bien conçu.

Malheureusement, j’ai beaucoup plus travaillé les deux dernières semaines que les deux premières semaines, j’aurai dû mieux répartir mon temps afin de me sentir moins stressé.

Enseignements tirés

Beaucoup d'enseignements et de connaissances sont à tirer de ce projet. Tout d'abord, toutes les connaissances que j'ai pu accumuler durant le cours de Méthodologie de la Programmation. C'est grâce à ces cours que j'ai développé ma capacité à concevoir un projet de A à Z et de comprendre l'importance d'une bonne conception. J'ai aussi découvert la méthode des raffinages que j'ai pu appliquer un maximum et gagner beaucoup de temps lors de l'implémentation du projet.

Techniquement, j'ai pu découvrir en profondeur le langage Ada et plein de notions telles que l'instanciation de modules, la généricité, la factorisation de code, gestion des exceptions et plein d'autres encore...

Dorénavant, je ferai toujours de mon mieux pour respecter la rigueur que j'ai pu développer au sein de ce projet. Les enseignements que j'en ai tirés me suivront tout au long de mon apprentissage et même tout au long de ma vie professionnelle.

Je vous remercie pour la lecture de ce rapport.