Rapport
Projet PIM S5

Tom Pillot, Quentin Thuet

Décembre 2019

Le projet de Programmation Impérative du semestre 5 consiste en la conception d'un système de gestion d'arbres généalogiques. Codé dans le langage Ada, il permet la manipulation des modules et des types abstraits de données. La contrainte temporelle était d'environ 1 mois et demi, et le projet était à réaliser en binôme. Ce rapport présente les différents aspects de la réalisation du projet, de l'architecture mise en place à l'organisation du binôme.

1 Introduction

L'objectif du projet est donc de mettre en place un gestionnaire d'arbres généalogiques. La conception s'est effectuée en deux parties :

— Dans un premier temps seuls les liens fils/fille - père/mère (donc pas de liens de fraternité, ou bien de conjoints multiples) ont été implémentés, le tout associé à un registre d'état civil comprenant les informations sur les individus de l'arbre.

— Les liens plus complexes décrits précédemment ont été implémentés dans une seconde partie.

Cette implémentation a été effectuée selon un schéma précis :

1. Raffinage du programme principal
2. Description de l'architecture logicielle et spécifications des modules
3. Implémentation des programmes de tests des modules
4. Implémentation des modules
5. Implémentation de la première partie du programme principal
6. Implémentation de la seconde partie du programme principal

Fichiers rendus

Un dossier pour chaque partie a été rendu sous svn. La première partie présente l'avantage d'être robuste (notamment au niveau des vérifications des dates), mais le code n'a pas été mis en forme très proprement. En revanche, malgré quelques soucis au niveau des dates dont nous vous parlerons plus tard dans ce rapport, le code de la seconde partie est bien commenté (notamment les spécifications détaillées et la description de toutes les variables en jeu). De plus, la seconde partie reprend toutes les fonctionnalités de la première.

Table des matières

2 Architecture et types de données

Le principal défi de ce projet était de choisir une architecture et des types de données cohérents.

2.1 Architecture de l'application en modules

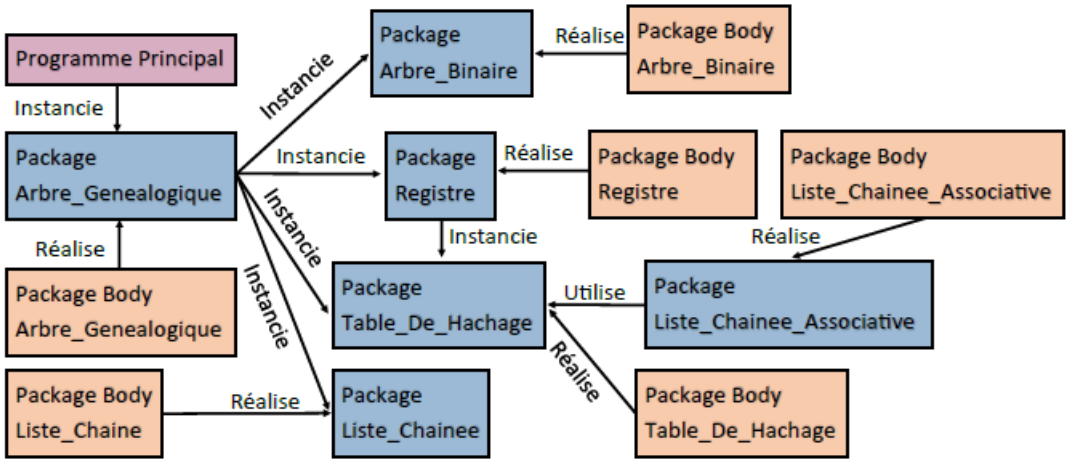

FIGURE 1 – Architecture logicielle du projet

Le module `Arbre_Genealogique` rassemble toutes les structures de données utiles au programme principal, c'est-à-dire :

- une liste chaînée d'arbres binaires, chaque arbre binaire contenant les relations entre les différents individus stockés,
- un registre contenant les informations de chaque individu,
- une table de hachage contenant pour chaque individu le pointeur vers le noeud de l'arbre binaire lui correspondant.

Une explication détaillée de ces trois structures vous est donnée section 2.3.

2.2 Choix réalisés

Il est important de mentionner certains choix qui ont été faits lors du développement de ce projet.

2.2.1 Gestion des identifiants

Pour générer des identifiants uniques pour chaque individu dans l'arbre généalogique, nous avons choisi de créer une variable `val_id` dans `Arbre_Genealogique` initialisée à 1. À chaque ajout d'un individu dans un arbre, cet individu a pour identifiant la valeur actuelle de `val_id`, puis `val_id` est incrémentée.

2.2.2 Table de hachage des noeuds

Pour éviter d'avoir à parcourir les arbres binaires pour trouver un identifiant à chaque fois que l'on veut ajouter, modifier ou supprimer un individu, nous avons décidé d'utiliser une table de hachage contenant les pointeurs vers tous les noeuds des arbres. L'élément associé à un identifiant dans la table des noeuds est un pointeur vers le noeud de l'arbre correspondant à cet identifiant. Cela permet d'accéder aux noeuds de l'arbre en temps constant.
Ce n'est pas une solution optimale du point de vue de la fiabilité, car cela autorise un utilisateur du module `Arbre_Binaire` à manipuler des pointeurs (`T_AB`) sans aucun contrôle de notre part. Cela peut causer les problèmes de gestion de la mémoire que nous avons vu en TP.

2.2.3 Arbre binaire doublement chaîné

Les éléments de type `T_AB` (arbre binaire) contiennent, en plus d'un pointeur sur le sous arbre gauche et droit, un pointeur sur le noeud parent. Cela nous permet de gérer les suppressions en temps constant, car nous n'avons pas besoin de chercher le parent de l'élément à supprimer dans tout l'arbre pour faire pointer son pointeur sur le sous-arbre à supprimer sur `null`.

2.2.4 Encapsulation

Pour chaque module il était important pour nous d'empêcher les autres modules qui en dépendent de faire des actions non voulues par le module. Ainsi nous avons utiliser des types `private` et `limited private` dès que cela était possible.

2.2.5 Exceptions

Dans le cadre de la programmation défensive, nous avons choisi d'utiliser les exceptions dans tous nos modules pour pouvoir facilement détecter les problèmes et les gérer dans le programme principal. Cette solution permet par exemple d'afficher un message à l'utilisateur lorsqu'il entre un identifiant absent dans l'arbre sans que le programme s'arrête.

2.2.6 Implémentation de la seconde partie

Nous avons fait le choix d'implémenter la deuxième partie en utilisant au maximum le travail effectué en première partie. Ainsi, la seule modification majeure est que le type `T_AG` (Arbre généalogique) est un enregistrement d'une liste chaînée d'arbres binaires (module `Liste_Chainee` que nous avons implémenté pour l'occasion, utiliser le module `Liste_Chainee_Associative` n'aurait pas eu de sens ici) au lieu d'un unique arbre binaire. Cela permet la manipulation de plusieurs arbres, mais aussi l'ajout de frères/soeurs et demi-frères/soeurs, par la création d'un nouvel arbre pour chacun, et par une procédure de liaison

d'arbres binaires entre eux. Par exemple, pour l'ajout d'un frère, le procédé est le suivant :

- Création d'un nouvel arbre
- Initialisation avec les informations sur le nouvel individu
- On appelle la procédure de liaison, qui va affecter comme sous-arbres du nouvel individu les parents de son frère

Malgré cette façon un peu grossière de définir la relation frère-soeur (un arbre créé pour chaque relation), le coût en mémoire n'est pas très important car c'est finalement un seul noeud d'arbre qui est créé pour chaque frère/soeur, et chaque individu n'est stocké qu'une seule fois. De plus, le registre et la table des noeuds uniques permettent de même de stocker une unique fois chaque individu. Surtout, ce choix nous a permis d'implémenter rapidement la seconde partie après avoir construit des bases solides durant la première.

2.3 Principaux algorithmes et types de données

Les types de données étaient une problématique centrale de ce projet, nous vous présentons ceux qui ont été utilisés.

2.3.1 Arbre Binaire

Le module `Arbre_Binaire` permet de créer un arbre binaire contenant des valeurs de type générique. Les accès aux éléments contenus par un arbre se font par le pointeur sur le noeud contenant l'élément. L'arbre est constitué de noeuds de types `T_Noeud` qui contient une valeur, un pointeur sur le sous arbre gauche, un pointeur sur le sous arbre droit et un pointeur sur le noeud parent.

2.3.2 Liste Chaînée Associative

Le module `Liste_Chainee_Associative` permet de créer une liste de taille variable où chaque élément de la liste est associée à une clé. La liste est constituée de plusieurs cellules de type `T_Cellule` qui contiennent une clé, une donnée et un pointeur vers la cellule suivante de type `T_LCA`.

2.3.3 Table de Hachage

Le module `Table_De_Hachage` permet de créer une table contenant un nombre variable d'élément, chaque élément est associé à une clé. La table de hachage est définit par le type `T_TH` qui est un tableau de `Capacite` éléments. Pour régler le problème des collisions généré par la fonction de hachage, nous avons choisi d'utiliser des listes chaînées associatives pour les éléments du tableau : les éléments associés à une même clé sont rangés dans une `T_LCA`. La fonction de hachage, la capacité et les types des données et des clés sont génériques pour permettre à ce module d'être réutilisé dans différentes situations.

2.3.4 Registre

Le module `Registre` permet de créer une table de hachage qui à un identifiant associe les informations lui correspondant, et de capacité variable. Ces informations sont définies par le type `T_Information` qui est un enregistrement du nom (`Unbounded_String`), du prénom (`Unbounded_String`), d'une date de naissance (`T_Date`), d'un lieu de naissance (`Unbounded_String`) et d'une date de décès (`T_Date`, si la personne n'est pas décédée, la date de décès vaut alors `01/01/10000`). Le type `T_Date` est lui-même un enregistrement de `Jour`, `Mois`, `Annee` qui sont tous de type `Integer`.
Le module Registre instancie donc une table de hachage et hérite donc de toutes les fonctions du module `Table_De_Hachage`, mais les fonctions `Enregistrer` et `Modifier` ont été surchargées pour permettre de contrôler la conformité des dates saisies. Ces outils de contrôle des dates ne sont pas fonctionnels pour l'ajout de frère/soeur ou demi-frère/soeur (par manque de temps) et cela constitue donc une perspective d'amélioration.

2.3.5 Liste Chaînée

Le module `Liste_Chainee` permet de créer une liste chaînée contenant des valeurs de type générique. La liste est composée de plusieurs cellules de type `T_Cellule` qui contiennent une valeur et un pointeur sur la prochaine cellule de type `T_LC`.

2.3.6 Arbre Généalogique

Le module `Arbre_Genealogique` implémente les sous-programmes nécessaires à la gestion d'un arbre généalogique. Il instancie `Liste_Chainee`, `Arbre_Binaire`, `Registre` et `Table_De_Hachage`. Le type `T_AG` est un enregistrement d'une liste chaînée d'arbres binaires `AB`, une liste chaînée contenant les identifiants des racines de chaque arbre `Id_Racine`, un registre `Registre` et une table de hachage `Noeuds`. Les arbres binaires stockent les identifiants des individus ajoutés dans les arbres. Nous pouvons rapidement connaître la racine d'un arbre grâce à la liste des identifiants racine. Le registre permet d'enregistrer les informations de chaque individu. La table de hachage des noeuds stocke les pointeurs vers tous les noeuds de l'arbre, c'est-à-dire que l'élément associé à un identifiant dans la table des noeuds est un pointeur vers le noeud de l'arbre correspondant à cet identifiant.

2.3.7 Programme Principal

Le programme principal permet une manipulation conviviale et fiable d'un arbre généalogique. Il s'appuie uniquement sur le module `Arbre_Genealogique` qui rassemble toutes les fonctions, procédures et exceptions utiles à la manipulation d'arbre généalogique.
Le fonctionnement du programme du point de vue de l'utilisateur est détaillé dans le manuel.

2.4 Mise à jour de l'architecture

Aucune modification majeure de l'architecture du projet n'a eu lieu après le jalon [J2] en ce qui concerne la partie 1 (l'architecture a été légèrement modifiée pour la partie 2, comme décrit section 2.2.6). Des modifications internes aux modules ont bien sûr été effectuées. En effet, il était parfois difficile de prévoir l'ensemble des procédures ou fonctions qui nous ont été nécessaires (voir section 3). Ces modifications mineurs sont donc principalement des fonctions ou procédures qui ont été soit ajoutées, soit modifiées car nous n'avions pas anticipé leur utilité, par exemple :

- Ajout de la fonction `Feuille` dans le module `Arbre_Binaire` qui renvoie un pointeur sur un noeud à partir de son noeud parent.
- Surcharge de la fonction `Taille` dans le module `Arbre_Genealogique` pour qu'elle renvoie également la taille d'un sous arbre à partir d'un identifiant donné.
- Ajout de l'option `PREMIER` au type énuméré `T_Direction` dans le module `Arbre_Binaire` qui permet d'ajouter le premier élément d'un arbre sans avoir à surcharger la fonction `Ajouter`.

2.5 Tests

Pour tous nos modules, nous avons créé des programmes de test qui permettent de vérifier que chaque module fonctionne correctement. Cela nous a permis de voir notre avancement au fur et à mesure de l'implémentation, jusqu'à ce que tous les tests passent. Cependant, même si tous les tests passent, il peut rester des cas qui posent problème auxquels nous n'avons pas pensé.

3 Difficultés

Malgré une mise en oeuvre du projet efficace, nous avons rencontré certaines difficultés que nous vous décrivons ici.

3.1 Spécifications et tests

La première difficulté que nous avons rencontré était l'écriture des spécifications et des tests. Nous avons rapidement compris quelle était la structure générale de chaque module (les principaux types, fonctions et procédures), mais nous avons eu plus de mal à anticiper toutes les fonctions et procédures moins centrales mais néanmoins nécessaires. De même, la structure de nos tests était bonne, mais au moment de l'implémentation des modules, nous avons du les modifier car nous n'avions pas anticiper certains comportements.

3.2 Choix de la méthode d'accès aux noeuds de l'arbre binaire

Nous sommes rapidement tombés d'accord sur l'architecture en modules du projet. Une autre décision qui a été prise dès le début est l'accès aux noeuds de l'arbre binaire directement par pointeurs. Cela permet un accès rapide à un noeud que l'on a exploité en stockant les noeuds associé à chaque identifiant (voir section 2.2.2). Nous avons réussi à nous en tenir à l'architecture décidé initialement, mais cela a mené à quelques difficultés. Le programme de test du module `Arbre_Binaire` a du être défini dans le fichier `arbre_binaire.adb` pour faciliter son implémentation tout en nous permettant de contourner le mode `private` du type `T_AB`.

3.3 Instanciation du module Table_De_Hachage par le module Registre

Dans un objectif évident de cohérence, nous avons voulu éviter d'implémenter tout le module `Registre`, celui-ci étant simplement une table de hachage spécialisée. Nous avons eu quelques difficultés à réaliser cet héritage tout en surchargeant tout de même les fonctions Enregistrer et Modifier pour que les informations saisies dans le registre soient contrôlées. Cependant, avec une bonne compréhension des structures et la syntaxe en jeu, nous avons trouvé une solution qui semble très bonne.

3.4 Détruire la liste d'arbres binaires

Dans la seconde partie, notre arbre généalogique possède une liste d'arbres binaires. La difficulté était de détruire les arbres binaires sans problèmes de mémoire. En effet, les arbres sont liés entre eux, ainsi quand on détruit le premier arbre, des noeuds des arbres suivants sont aussi détruits, si on essaye alors de détruire le second arbre on risque d'accéder à des noeuds déjà supprimés. Une

solution temporaire que nous avons trouvé pour éviter les problèmes de mémoire est de détruire entièrement le premier arbre et de supprimer seulement le premier noeud des autres arbres. Dans le cas où on utilise seulement `Ajouter_Frere` ou `Ajouter_Demi_Frere`, cela ne crée pas de problème car les frères sont ajoutés comme premier noeud de nouveaux arbres, ils sont ainsi détruits correctement. Cependant si l'utilisateur crée lui même un deuxième arbre, il ne sera pas détruit correctement et il y aura des fuites de mémoire. Il faudrait régler ce problème pour améliorer notre programme. La partie 1 n'est pas touchée par ce problème.

4 Organisation

Le travail a été réparti de manière très équilibrée. Chacun a travaillé sur un certain nombres de modules (voir la table 1), autant sur les spécifications que sur les implémentations. Si l'un de nous avait principalement travaillé sur un module, nous avons essayé de faire que ce soit uniquement lui qui apporte des modifications dessus. Par exemple, c'est Tom qui a principalement travaillé sur le module `Table_De_Hachage` et Quentin sur le module `Registre`, et si Quentin avait besoin pour le registre d'une nouvelle fonction dans `Table_De_Hachage` c'est Tom qui implémentait cette nouvelle fonction, car il connaît mieux le module `Table_De_Hachage`. Ainsi, Quentin n'avait pas besoin de se plonger dans la compréhension détaillée de l'implémentation de `Table_De_Hachage` juste pour l'ajout d'une fonction. Cela impliquait une bonne communication entre nous qui a été très efficace.

Module	Spécification et tests	Implémentation
`Programme_Principal` (Partie 1)	/	Quentin
`Programme_Principal` (Partie 2)	/	Quentin
`Arbre_Genealogique`	Tom	Tom
`Table_De_Hachage`	Tom	Tom
`Registre`	Tom	Quentin
`Arbre_Binaire`	Quentin	Quentin
`Liste_Chainee_Associative`	Tom	Tom
`Liste_Chainee`	Tom	Tom

TABLE 1 – Répartition du travail de programmation

Mis à part le travail de programmation, Quentin a écrit les raffinages du programme principal, et le manuel et ce rapport ont été écrits par nous deux de manière équilibrée.

5 Bilans

5.1 Bilan technique

Ce projet nous a amené dans un premier temps à travailler sur de nombreuses structures de données différentes, et à réfléchir à l'architecture de nos modules en amont de l'implémentation. C'est sans doute cette phase qui aura causé le plus de difficultés. Dans un deuxième temps, l'implémentation de nos modules à été rapide et efficace grâce à une bonne répartition du travail. La dernière phase, celle de l'ajout de fonctionnalité au programme principal et la gestion de plusieurs arbres, nécessitait une manipulation poussée de toutes les structures de données précédemment implémentés et une partie de notre temps était consacrée à la rédaction du manuel utilisateur et de ce rapport. En faisant le choix d'utiliser au maximum les structures implémentées en première partie, nous avons eu des résultats rapides sur cette dernière phase.
Nous sommes satisfait de notre travail car nous avons apporté des solutions à l'essentiel des problèmes énoncés dans le sujet. Toutes les solutions apportées ne sont sans doute pas optimales, mais nous pensons avoir fait des compromis satisfaisant étant donné le temps qui nous était donné pour la réalisation du projet.
Enfin, l'expérience de travail en équipe est positive. En effet, une communication efficace et une bonne répartition du travail permettait à chacun d'entre nous de savoir sur quoi il devait travailler sans interférer dans le travail de l'autre. Aussi, nous avons bien réussi à trouver une solution globale satisfaisante pour chacun de nous dès le début, ce qui nous a permis de ne jamais devoir revenir en arrière sur notre travail.
Pour améliorer notre programme, nous devrions rajouter la possibilité de pouvoir compter le nombre de frères et soeurs d'une personne et de les afficher. La robustesse du programme (notamment la vérification des dates) et la gestion de toutes les exceptions sont également une perspective d'amélioration.

5.2 Bilans personnels

5.2.1 Bilan de Tom Pillot

Ce premier projet m'a permis d'apprendre à travailler efficacement en équipe et à respecter des contraintes de temps. Il m'a permis de découvrir les arbres binaires et les tables de hachages, ainsi que d'approfondir mes connaissances sur la gestion des pointeurs. La rédaction du rapport et du manuel m'a aussi permis d'apprendre à utiliser LaTeX.
Temps passé à la conception : 3h
Temps passé sur l'implémentation : 35h
Temps passé sur le rapport et le manuel utilisateur : 6h

5.2.2 Bilan de Quentin Thuet

Les nombreux choix à faire, principalement sur l'architecture des différents modules, ont rendu ce projet très intéressant. D'autant plus intéressant que ces décisions étaient prises à deux, avec des avis parfois différents. J'ai donc appris à faire des choix en équipe et à les mettre en oeuvre de manière coordonnée, et notre collaboration s'est très bien passée. La manipulation de nombreuses structures de données différentes m'a permis de bien comprendre les avantages et inconvénients de celles-ci mais aussi le principe de la programmation modulaire. Enfin la rédaction du rapport et du manuel utilisateur était un exercice aussi intéressant sur le fond (retranscrire les idées que l'on a en codant n'est pas un exercice aisé) que sur la forme avec l'utilisation de LaTeX.
Temps passé à la conception : 2h
Temps passé sur l'implémentation : 40h
Temps passé sur le rapport et le manuel utilisateur : 8h