Méthodologie de la Programmation

N5AN02B

Projet : Un arbre généalogique

Réalisé par

Stéphane Loppinet & Ziuzin Nikita

I. Introduction

L'objectif de ce rapport est d'écrire en détail la façon dont nous avons abordé la réalisation de ce projet d'arbre généalogique. Dans un premier temps seront détaillés les différents choix techniques pris au début / pendant la réalisation de ce projet. Ensuite, plusieurs algorithmes faisant partie du code source seront présentés à l'aide des raffinages faits lors de la phase conception du projet. Pour finir, nous citerons les difficultés rencontrées ainsi que ferons un bilan pour revenir sur notre répartition des tâches, la gestion du temps ainsi que les enseignements tirés de ce projet.

II. Nota bene

A. Github

Nous avons utilisé la plateforme Github pour pouvoir travailler sur ce projet à deux et faciliter la gestion des versions ainsi que la mise en commun des différentes fonctionnalités. Vous pouvez retrouver notre répertoire en cliquant ici (vous pouvez nous demander l'accès au préalable).

B. Vocabulaire

Dans la suite de ce rapport plusieurs termes contradictoires seront utilisés, tels que "fils", "parent", "enfant" et "ancêtre". Afin d'éviter d'éventuelles confusions lors de la lecture du rapport, il est important de noter que :

- Un "fils" est le terme utilisé traditionnellement dans un arbre binaire pour caractériser un nœud descendant direct d'un autre nœud. Dans le cas de l'arbre généalogique, nous parlerons également de "enfant". Ainsi, lorsque nous citerons les fonctionnalités du module arbre binaire, nous parlerons de "fils". Le terme "enfant" sera quant à lui utilisé quand nous traiterons du sujet de l'arbre généalogique.
- De même, "parent" est le terme utilisé traditionnellement dans un arbre binaire pour caractériser le nœud directement au-dessus d'un autre nœud. "Ancêtre" est le terme logique dans le cas de l'arbre généalogique. Ainsi pour retrouver les ancêtres, on parcourt les "fils" de l'arbre.

C. Manuel d'utilisation

Afin de mieux comprendre comment utiliser le menu interactif qui permet de manipuler les arbres généalogiques, un manuel d'utilisation a été rédigé. Celui-ci se trouve dans l'archive rendue sur Moodle, mais vous pouvez également le retrouver ici.

III. Architecture et choix techniques

A. Diagramme des classes et types utilisés

Notre projet a une architecture d'application en modules. Au total, nous avons 4 modules ou "packages" principaux :

- Le module générique arbre binaire (**BinaryTree**) comportant l'ensemble des fonctionnalités liées à la création d'arbres, ajout/suppression/recherche des nœuds ainsi que l'affichage des arbres. Il se base sur un type record T_Node, comportant une clé, un élément générique (la data) ainsi que les pointeurs vers ses "fils". Une procédure générique "Put_Generic" est également présente pour gérer l'affichage de l'élément générique.
- Le module arbre généalogique (**FamilyTree**) qui se base sur le module BinaryTree et ajoute plusieurs fonctionnalités telles que la recherche d'orphelins, la recherche d'individus par génération et ainsi de suite.
- Le module **Person** qui introduit l'ensemble des fonctionnalités liées aux individus - c'est sur le type record **T_Person** (prénom, nom, sexe et date de naissance) de ce module-là que se basera **FamilyTree** pour instancier le modèle générique **BinaryTree**.
- Le module **utils** qui comporte des sous-programmes utilisés dans l'ensemble des modules précédents, par exemple pour formater certaines données. Dans ce module sont également créés plusieurs types enum qui permettent une gestion plus fine des arguments de sous-programmes des modules principaux. On y retrouve également les déclarations de certaines exceptions.

Voici le diagramme des classes qui en résulte (certains sous-programmes / types ne sont pas présents dans celui-ci, notamment le menu) :

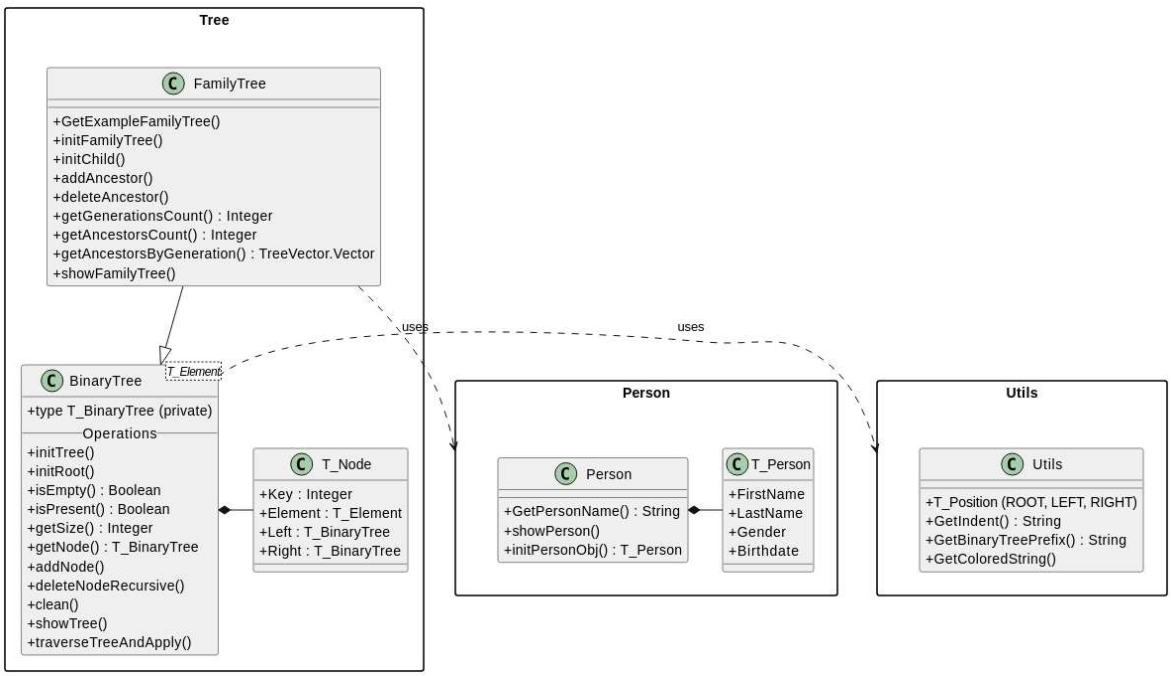

Figure 1: Diagramme des classes / modules

B. Présentation choix techniques

- Nous avons décidé de combiner les concepts de programmation offensive et défensive. Ainsi, nos modules de gestion d'arbres se basent sur des préconditions pour fonctionner, sans aucune vérification interne. Ces conditions sont quant à elles respectées par un ensemble de tests redondants dans le menu, après la saisie de l'utilisateur

- Nous avons opté pour la librairie standard ada pour gérer les vecteurs (plutôt que d'utiliser notre implémentation de vecteurs durant le tp8) pour plus de simplicité et éviter les bugs.

- Afin de supprimer un nœud (et ses ancêtres). Il faut mettre à "null" le pointeur de la génération -1 du nœud pour éviter d'avoir un pointeur vers un nœud qui n'existe plus. De plus, on pourrait être tenté de supprimer uniquement le pointeur du nœud pour supprimer le sous-arbre avec tous les ancêtres, mais on perdra de la mémoire correspondant au contenu d'un nœud :
    - 3 Strings encodés en ASCII (1 octet par caractère)
    - 1 type date (2 entiers et un tableau de 12 strings correspondant au mois)
    - 1 Pointeur (Sur une architecture 64 bits : 8 octets)

  On a donc décidé de parcourir chaque ancêtre pour libérer la mémoire pour éviter les pertes de mémoire même si cela prend plus de temps.

- Nous avons choisi de ne pas implémenter une structure de type Clé-Valeur pour notre arbre binaire. À la place, nous utilisons un type générique T_Element pour représenter les éléments stockés dans l'arbre. Cette décision simplifie l'architecture globale et permet une plus grande flexibilité dans l'utilisation de l'arbre pour différents types de données.

- Initialement, nous n'avions pas prévu d'implémenter une méthode générique pour l'affichage des éléments de l'arbre. Cependant, nous avons réalisé que cela était nécessaire pour permettre un affichage cohérent et personnalisé des éléments de type T_Element. La méthode Put_Generic a donc été ajoutée pour répondre à ce besoin, en fournissant un moyen standardisé d'afficher le contenu d'un nœud.

- Afin de factoriser le code, nous avons décidé d'implémenter une fonction générique pour parcourir l'arbre qui prend en paramètre une callback (actions à réaliser sur chaque nœud) visant à être appelée lors d'une suppression, d'un affichage, d'une recherche d'un ancêtre.
    - Volonté de séquencer le code en plusieurs petites fonctions (compliqué en Ada).
    - La fonction deleteNodeRecursive était à l'origine décomposée en une fonction et une procédure.
        - La fonction cherchait le pointeur vers le nœud visant à être supprimé
        - La procédure s'occupait de supprimer tout le sous-arbre avec le retour de la fonction

Problème : En Ada, il n'est pas possible de définir un type étant soit fonction soit procédure, comme ceci :

```
ActionCallback : not null access procedure or function (ABR : in out T_BinaryTree; Parent : in out T_BinaryTree; Stop : in out Boolean);
```

Nous avons choisi de rassembler la fonction et la procédure en une seule pour éviter ce problème et on en a profité pour améliorer la fonction générique de parcours en ajoutant une variable "Parent" qui se charge à chaque itération de garder le parent du nœud courant. Ce qui est utile notamment dans les cas de suppression d'un nœud, d'un sous-arbre.

C. Présentation des principaux algorithmes

1. Suppression de noeuds

```
-- Delete a node element and all his children
procedure deleteNodeRecursive
  (ABR : in out T_BinaryTree; Key : in Integer) with
  Post => getSize(ABR) = getSize (ABR)'Old - getSize(getNode(ABR, Key))'Old and not isPresent (ABR, Key);
```

```
R0 : Supprimer un noeud avec son identifiant et ses ancêtres
       arbre: T_BinaryTree in
       id_recherché : entier in

R1 : Comment "Supprimer un noeud avec son identifiant et ses ancêtres ?"

       - Rechercher l'enfant du noeud à supprimer (génération -1)
       R2: Comment "rechercher l'enfant du noeud à supprimer (génération -1)" ?
       fonction getNodeChild
       noeud_: T_BinaryTree out

              - Parcourir récursivement fg et fd jusqu'à trouver l'enfant du noeud
              R3: Comment "Parcourir récursivement fg et fd jusqu'à trouver l'enfant …" ?
                     if noeud = null alors retourner null
                     else if noeud.fg.id = id_recherche
                            alors
                                   Sauvegarder noeud.fg temporairement
                                   Mettre noeud.fg à null (supprimer pointeur)
                                   retourner noeud.fg
                     else if noeud.fd.id = id_recherche
                            alors
                                   Sauvegarder noeud.fd temporairement
                                   Mettre noeud.fd à null (supprimer pointeur)
                                   retourner noeud.fd
                     else
                            fg = noeud.fg
                            Parcourir les fils du fg récursivement jusqu'à trouver l'enfant
                            fd = noeud.fd
                            Parcourir les fils du fd récursivement jusqu'à trouver l'enfant

        - Supprimer le sous-arbre du noeud enfant
       R2: Comment "Supprimer le sous-arbre du noeud enfant"
       fonction deleteSubTreeWithoutOriginalPointer
              if noeud = null alors ne rien faire
              else
                     si noeud.fg existe alors
                            Sauvegarder noeud.fg temporairement
                            Mettre noeud.fg à null (supprimer pointeur)
                     si noeud.fd existe alors
                            Sauvegarder noeud.fd temporairement
                     Mettre noeud.fd à null (supprimer pointeur)
                     Supprimer noeud
                     fg = noeud.fg
                     Parcourir les fils du fg récursivement
                     fd = noeud.fd
                     Parcourir les fils du fd récursivement
```

2. Recherche d'ancêtres par génération

```
-- 4. Obtenir l'ensemble des ancêtres situés à une certaine génération d'un individu donné.
function getAncestorsByGeneration
  (ABR : in T_FamilyTree; Key : in Integer; Generation : in Integer)
   return TreeVector.Vector;
```

```
R0: Obtenir les ancêtres d'une certaine génération d'un individu
arbre: FamilyTree in
id: entier in
gen: entier in
R1: Comment : "R0" ?
       - Trouver noeud avec id (fonction: getNode)
       - Parcourir le fg & fd du noeud récursivement jusqu'à la bonne génération
       R2 : Comment "parcourir le fg & fd du noeud récursivement" ?
       fonction : getAncestorsAtDepth(noeud, gen)
       result: Vector out
       noeud: T_Noeud in

              if noeud = null ou gen < 0 alors retourner result
              else if gen = 0 alors ajouter noeud à result
              else
                     gen = gen - 1
                     fg = noeud.fg
                     Parcourir les fils du fg récursivement jusqu'à gen = 0
                     fd = noeud.fd
                     Parcourir les fils du fd récursivement jusqu'à gen = 0
```

3. Recherche de noeud par clé

```
-- Search Tree by Element
function getNode (ABR: in T_BinaryTree; Key : in Integer) return T_BinaryTree;
```

```
R0: Obtenir un sous-arbre correspondant à l'id de son origine
fonction: getNode
id_recherche: entier in
arbre: T_BinaryTree in
R1: Comment "Obtenir le noeud correspondant à un id" ?
       Vérifier si l'origine de l'arbre est le noeud cherché
       Parcourir le fg & fd du noeud récursivement jusqu'à trouver le noeud
       R2: Comment "Parcourir les fils du noeud récursivement jusqu'à trouver le
noeud" ?
              - Trouver l'enfant du noeud recherché grâce à getNodeChild(id)
              noeud: T_BinaryTree out

              - Vérifier lequel des fils est le noeud recherché
                     si noeud.fg.id = id_recherche alors retourner noeud.fg
                     si noeud.fd.id = id_recherche alors retourner noeud.fd
                     sinon retourner null
```

D. Démarche de tests

Afin de tester nos programmes au fur et à mesure de l'avancement du projet, nous avons suivi la démarche suivante :

- Création des cas de tests pour un sous-programme donné au tout début du projet, en se basant sur les signatures et les contrats définis dans les fichiers de spécification .ads.
- Implémentation de la fonction
- Implémentation des tests et correction des sous-programmes pour éliminer les éventuelles erreurs

Afin d'avoir une meilleure gestion de projet et éviter les régressions, nous avons ajouté un pipeline sur notre répertoire Github qui réalise tous les tests unitaires à chaque fois qu'une modification est réalisée sur la branche. Ces pipelines sont visibles ici (vous pouvez nous demander l'accès au préalable).

Figure 2: Pipeline Github pour les tests unitaires

E. Gestion des erreurs et robustesse

Le menu interactif permet à l'utilisateur d'utiliser toutes les fonctionnalités implémentées au cours de ce projet. Pour éviter toute erreur suite à une saisie incorrecte (qui résulterait en une pré condition non remplie), tous les edge case sont gérés directement dans le menu. Voici un exemple concret :

Plusieurs sous-programmes de FamilyTree s'attendent à recevoir une clé (correspondant à un individu) afin de réaliser une opération. C'est par exemple le cas du sous-programme deleteAncestor qui reçoit en argument la clé de l'individu à supprimer. Afin d'éviter qu'une clé inexistante soit passée à ce sous-programme, cette vérification se fait directement dans le menu de cette manière :

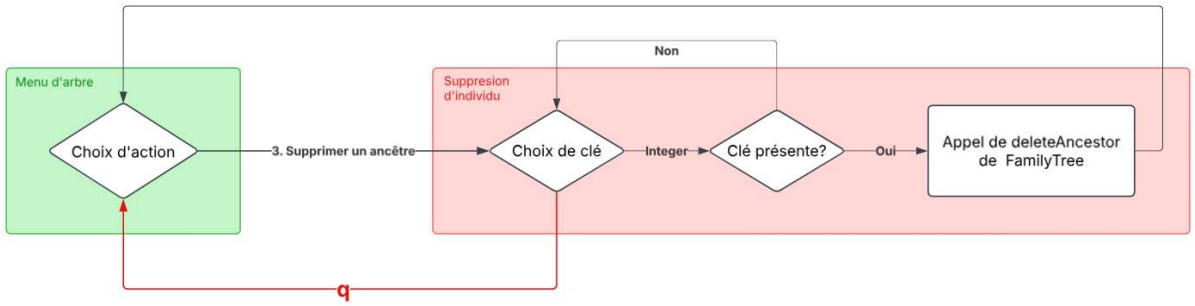

Figure 3: Vérification de la présence de clé

Le même concept est adapté pour toutes les autres fonctionnalités nécessitant une entrée utilisateur.

IV. Bilan du projet

A. Bilan technique

À la date du rendu de ce projet, soit le 2 février 2025, nous avons implémenté l'ensemble des fonctionnalités demandées dans le cahier des charges tout en ajoutant plusieurs fonctionnalités bonus, comme la possibilité de gérer plusieurs arbres généalogiques à partir du menu.

1. Pistes d'amélioration

Optimisation de l'exécution

Utilisation d'un dictionnaire :

- clé : identifiant du noeud
- valeur : pointeur vers le noeud

L'utilisation d'un dictionnaire pourrait permettre un accès direct et rapide aux nœuds en complexité O(1) via leur identifiant, évitant ainsi des parcours coûteux en O(n). Cette approche est particulièrement efficace pour les arbres dynamiques ou de grande taille, où les ajouts, suppressions et mises à jour sont fréquents. Cependant, dans notre cas, nous travaillons pour l'instant avec des petits arbres statiques, donc cela entraînerait une surcharge mémoire en doublant les références.

B. Difficultés rencontrées

1. Suppression d'un noeud avec son sous-arbre

La solution naïve à ce problème est de supprimer le sous-arbre à partir d'un nœud donné. Or, si on procède de la sorte, le pointeur du parent menant au nœud donné doit être mis à null pour éviter que ce pointeur mène à une mémoire ne contenant rien. La complexité se trouvait dans la volonté de factoriser le code et ainsi de créer des fonctions les plus génériques possibles. En l'occurrence, nous avions déjà fait une procédure de parcours de l'arbre qui réaliser une série d'opérations sur chaque nœud (traverseTreeAndApply). Il a donc fallu ajouter un paramètre pour garder le Parent du nœud courant au cas où il faut réaliser une action sur le Parent. De plus, nous avons opté pour un paramètre Stop qui a pour but d'arrêter le parcours récursif si l'objectif est réalisé. Ex : Si on cherche un nœud dans l'arbre et qu'on le trouve dès la 2e itération, pas besoin de continuer à parcourir tous les autres nœuds. Cela ajoute une couche de complexité en plus, mais qui est, selon nous, nécessaire notamment si on travaille avec des arbres contenant beaucoup de nœuds.

Pas d'utilisation de hashs car préférence de la simplicité des identifiants entiers. Pas besoin d'une confidentialité accrue non plus. On préfèrera utiliser un entier auto-incrémenté pour chaque nouvel individu de l'arbre. Pour gérer l'espace libéré après une suppression d'un individu, on utilisera une pile d'entiers pour les identifiants libérés (à utiliser en priorité). Si la pile est vide, on incrémente de 1.

C. Bilan binôme

1. Répartition des tâches

Afin de partir de la même base et de bien définir les attentes du projet et de chaque fonctionnalité, nous avons tenu à réaliser la rédaction des spécifications et des contrats de l'ensemble de nos modules à deux. De même, nous avons travaillé ensemble sur le module BinaryTree, qui allait nous servir de base pour la suite.

Ensuite, Stéphane a implémenté l'ensemble des sous-programmes du module FamilyTree tandis que Nikita s'est occupé de l'implémentation du menu interactif.

Enfin, nous avons réalisé respectivement les tests unitaires relatifs aux fonctionnalités que nous implémentons selon le modèle décrit dans la section Démarche de tests.

2. Gestion du temps

Lors de la première séance, nous avons fait un planning prévisionnel sur GANT pour répartir le travail entre les séances et se fixer des deadlines.

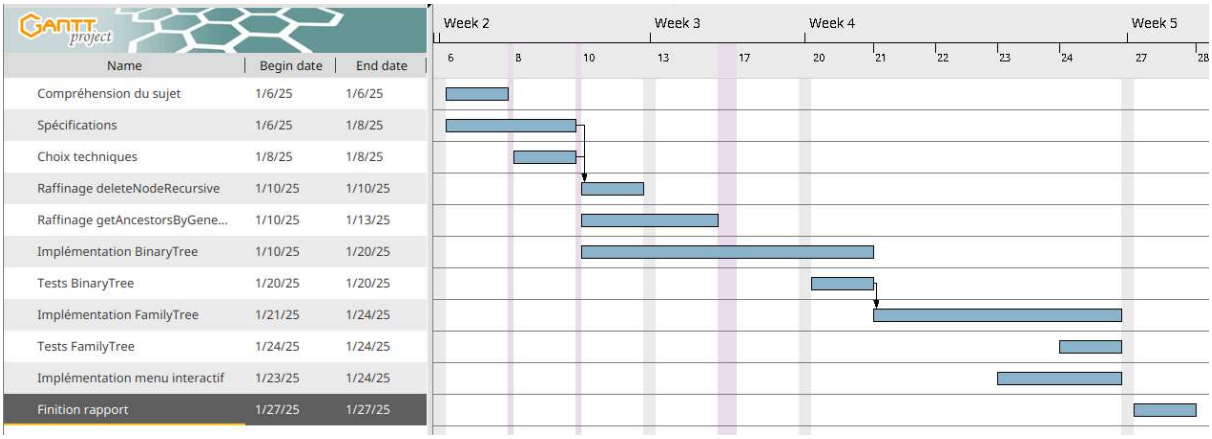

Figure 4: Planning prévisionnel établi lors de la première séance

Ce planning se basait sur le tableau ci-dessous. En rouge sont mentionnés les créneaux de travail / tâches que nous n'avions pas prévu lors de la phase de planification.

Date	Durée	Tâche à réaliser
Lundi 06/01	4h	Découverte du projet + rédaction des spécifications
Mercredi 08/01	2h	Rédaction des spécifications + Diagramme des classes
Vendredi 10/01	4h	Choix techniques + Raffinage des sous-programmes
Lundi 13/01	6h	Raffinage des sous-programmes + Implementation Binary Tree
Mardi 14/01	2h	Implementation Binary Tree
Vendredi 17/01	2h	Test Binary Tree

Lundi 20/01	6h	Implementation FamilyTree
Mercredi 22/01	2h	Implementation FamilyTree
- Jeudi 23/01	**4h**	**Implementation FamilyTree**
Vendredi 24/01	4h	Test FamilyTree + Implementation menu
Weekend du 25/01 au 26/01	**6h**	**Implementation menu + Test menu**
Lundi 27/01	4h	Relecture / Refactorisation + Manuel utilisateur + Rapport, **Implementation menu**
Weekend du 30/01 au 31/01	**4h**	**Léger refactoring, rédaction du rapport**

Ainsi, lors de la planification nous avions prévu de consacrer 36 heures à ce projet. Finalement, nous avons dû rajouter 12 heures supplémentaires pour le finir et rajouter des fonctionnalités bonus.

D. Bilan personnel

1. Nikita

Au début de ce projet, je n'étais pas assez à l'aise avec Ada et ses différences avec les langages de programmation auxquels j'étais habitué. Je me suis retrouvé plusieurs fois à vouloir utiliser des fonctionnalités qui n'étaient pas présentes nativement, ce qui m'a obligé à trouver des alternatives. Par exemple, la vérification des dates d'anniversaire que je pensais faire avec des regex s'est révélée impossible sans utiliser de packages GNAT externes. J'ai donc dû le faire moi-même d'une autre façon moins "propre".

Un des principaux problèmes liés au manque de connaissance de ADA et de gestion de mémoire en général était le problème de copie par référence / copie par duplication. En manipulant des variables access je confonds les deux ce qui faisait que je me retrouvais à modifier des données localement, sans modifier la donnée associée au pointer directement.

Une autre difficulté a été la gestion des **edge cases** dans le menu pour le rendre le plus simple et intuitif possible pour l'utilisateur. J'ai dû effectuer de nombreuses vérifications et gérer des exceptions précises afin de traiter au mieux les erreurs de saisie utilisateur et afficher des messages adaptés. Lors de l'implémentation de ce même menu, je me suis également retrouvé plusieurs fois à écrire du code redondant. Cela m'a obligé de réaliser plusieurs refactoring afin de rassembler des bouts de code similaires en un seul et même bloc.

2. Stéphane

Tout comme mon collègue, j'ai dû consacrer un temps significatif à comprendre ses spécificités. Cette phase d'apprentissage a ralenti mes premières avancées,
Dans un souci d'optimisation et de réutilisabilité, j'ai cherché à rendre certaines fonctions excessivement génériques. Cette approche, bien qu'intentionnellement ambitieuse, a rapidement montré ses limites. Le code est devenu difficile à lire et à maintenir, et j'ai passé beaucoup de temps à tenter de résoudre des problèmes de conception complexes qui, en fin de compte, n'apportaient pas

une réelle valeur ajoutée au projet. J'ai réalisé que la simplicité et la clarté devaient primer sur une généricité excessive, surtout lorsque celle-ci nuit à la compréhension globale du code.

V. Conclusion

Nous avons réussi à mener le projet à terme, en intégrant même des fonctionnalités bonus. Nous avons réussi à bien répartir les tâches. Le temps consacré à rédiger les spécifications ensemble s'est avéré payant : cela nous a permis de rester alignés tout au long du projet et d'éviter les changements d'architecture, qui ont été très rares. De plus, la documentation et le rapport ont été réalisés de manière proactive tout au long du projet, ce qui a facilité la gestion et la clarté du travail accompli.

Cependant, nous avons identifié des points à améliorer pour les futurs projets. Notre planification initiale s'est avérée trop optimiste, et nous avons dû ajuster notre calendrier pour tenir compte des tâches plus complexes que prévu. Bien que nous ayons su anticiper les parties les plus chronophages, il est essentiel d'être plus réaliste dans nos estimations pour éviter les retards.