SAE S2.02 – Rapport POO

C4

Mangin MAXENCE, Loïse VIGNERON, Manon VILODA-RIVIERE

02/06/2025

La SAE 2.01-2.02 a pour but de nous apprendre à développer un outil d'aide à la décision pour résoudre des problèmes complexes, par le biais d'algorithme d'optimisation et d'interface graphique. Les ressources nécessaires à la réalisation de ce projet sont :

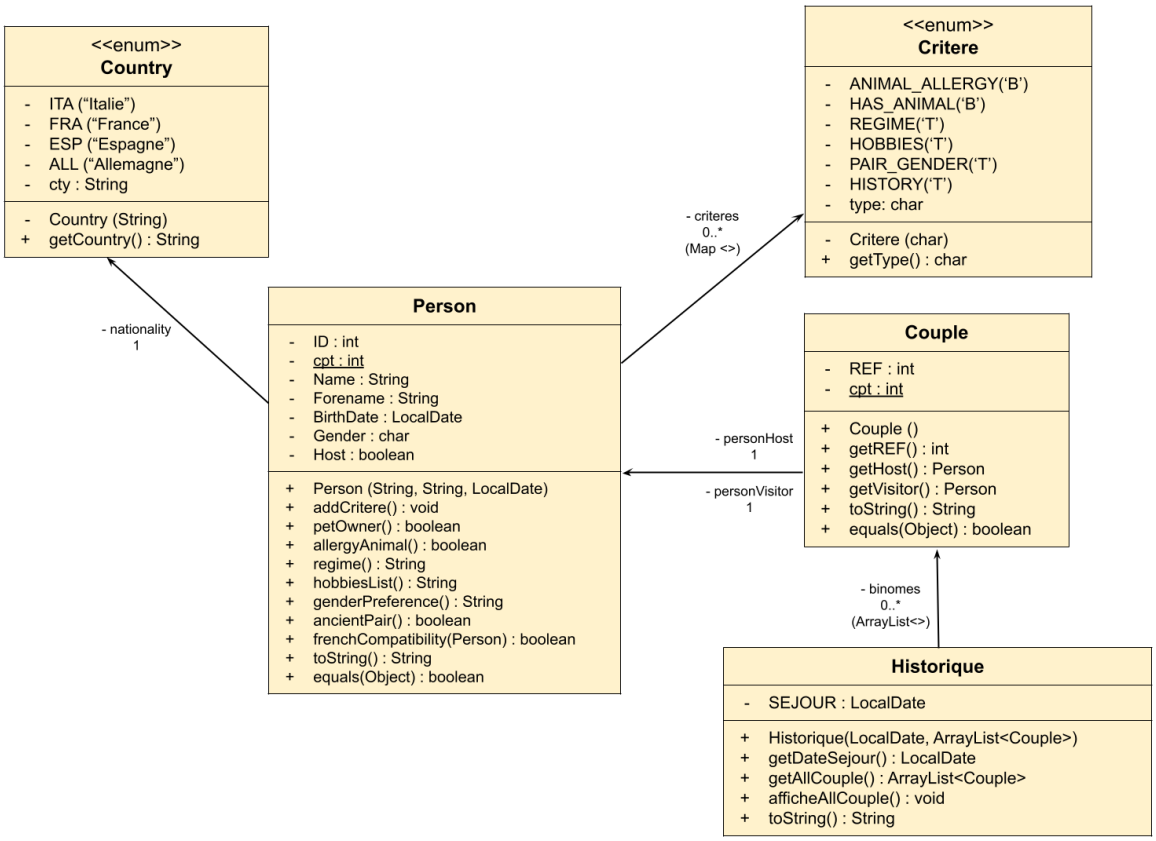
- R2.01 - Développement Orienté Objets
- R2.02 - Introduction à l'Interaction Humain-Machine
- R2.07 - Graphes

Le sujet de cette SAE est de permettre d'organiser des séjours linguistiques entre collégiens. Le besoin client est d'aider les professeurs à trouver les meilleures associations entre élèves visiteurs et élèves hôtes. Ces associations seront basées sur différents critères, comme par exemple : le régime alimentaire, les allergies, les loisirs, et d'autres encore.

Concernant la ressource R2.01, il nous a été demandé dans un premier temps de proposer un modèle UML répondant aux besoins décrits ci-dessus.

POO-v1

Modèle UML



Ci-dessus l'UML proposé lors de la 1ère semaine.

Choix des classes

Person

Nous avons décidé de créer une classe principale **Person** afin de réunir toutes les informations concernant un élève.
Dans cette classe nous avons comme attributs :
* *ID* : qui correspond à un numéro d'identifiant unique. * *cpt* : il correspond à un attribut statique utile à la création automatique d'un ID. * *name* : le nom de la personne. * *forename* : le prénom de la personne. * *birthDate* : Date d'anniversaire qui sera utilisé dans la réalisation des binômes en tant que critère d'affinité. * *gender* : le genre de la personne qui sera aussi considéré comme un critère. * *host* : **boolean** indiquant si la personne est un hôte ou non, s'il ne l'est pas, nous considérons qu'il est visiteur. * *nationality* : vient de l'énumération **Country** et indique la nationalité de la personne. * *criteres* : cet attribut est une **HashMap** prenant comme *Key* **Critere** (énumération) et comme *Value* un **String**.

En plus des *constructors*, *setters* et *getters*, notre classe **Person** possède différentes méthodes. Nous allons ci après les décrire une à une.

- *addCritere()* : permet d'ajouter un critère à la Map *criteres*.
- *petOnwer()* : renvoie un **boolean** qui indique si la personne possède ou non un animal de compagnie.
- *allergyAnimal()* : renvoie également un boolean, indiquant si la personne est allergique ou non aux animaux.

- *regime()* : permet de connaître le régime alimentaire de la personne.
- *hobbiesList()* : retourne une chaine de caractères comprenant les hobbies que l'élève à énoncé dans le formulaire.
- *genderPreference()* : renvoie un **String** contenant le genre voulu pour la création du binôme.
- *ancientPair()* : indique si l'adolescent souhaite être à nouveau avec un ancien camarade, ou au contraire, s'il ne veut surtout pas être à nouveau avec lui.
- *frenchCompatibility()* : cette méthode permet de savoir, lorsque l'élève est français, s'il y a au moins un loisir en commun avec son futur partenaire.
- *compatibility()* : cette méthode retourne un **int** indiquant un score d'affinité entre deux adolescents.

Country

Afin de faciliter la comparaison des nationnalités à l'avenir et d'éviter toutes erreurs de frappe lors de l'enregistrement des élèves, nous avons choisi de créer l'énumération **Country**.
Il s'agit ici d'un énumération enrichie car nous utilisons un label (*String cty*) spécifique pour chaque constante, reprenant le nom du pays.

Critere

L'équipe a décidé de réunir les critères dans une **HashMap**, et pour faciliter la recherche des *Keys* et *Values*, nous avons opté pour une seconde énumération enrichie, nommée **Critere**.
Celle-ci a pour constante les noms des différents critères utiles lors de la mise en place des duos. Chaque constante est enrichie d'un **char** qui correspond aux types de réponses fournies par le formulaire.
Comme indiqué par l'énoncé, nous avons respecté les dénominations suivantes : *'B'* pour **boolean** et *'T'* pour **String**. Nous n'avons pas encore ressenti le besoin d'utiliser le 'N'.

Couple

Afin de réunir deux élèves pour créer un binôme, nous avons opté pour la création d'une classe **Couple** encapsulant deux **Person**.
Dans cette classe nous retrouvons un numéro de référence unique (*REF*) incrémenté automatiquement, qui permettra à l'avenir de mieux différencier les binômes, ou encore de faciliter des possibles recherches de binômes (dans **Historique**, par exemple).
Les deux **Person** de cette classe correspont à l'hôte (*personHost*) et au visiteur (*personVisitor*).

La création de cette classe nous a paru également plus pratique dans la conception d'un historique, car ne nécessite de sauvergarder qu'un seul objet dans une **ArrayList**, à défaut de deux objets dans un tableau classique ou une **HashMap**.

Historique

Lors du remplissage du formulaire, les élèves ont la possibilité d'indiquer s'ils souhaitent ou non retrouver leur ancien groupe. Il semble donc nécessaire de mettre en place un système de sauvegarde des binômes des anciens séjour.
Pour ce faire, nous avons choisi de créer la classe **Historique**. Chaque **Historique** est identifié grâce à sa date de début de séjour et possède une liste de tous les binômes constitués durant ce séjour (via une **ArrayList**). Ces deux informations nous parraissent pour l'instant suffisantes pour conserver toutes les informations nécessaires.

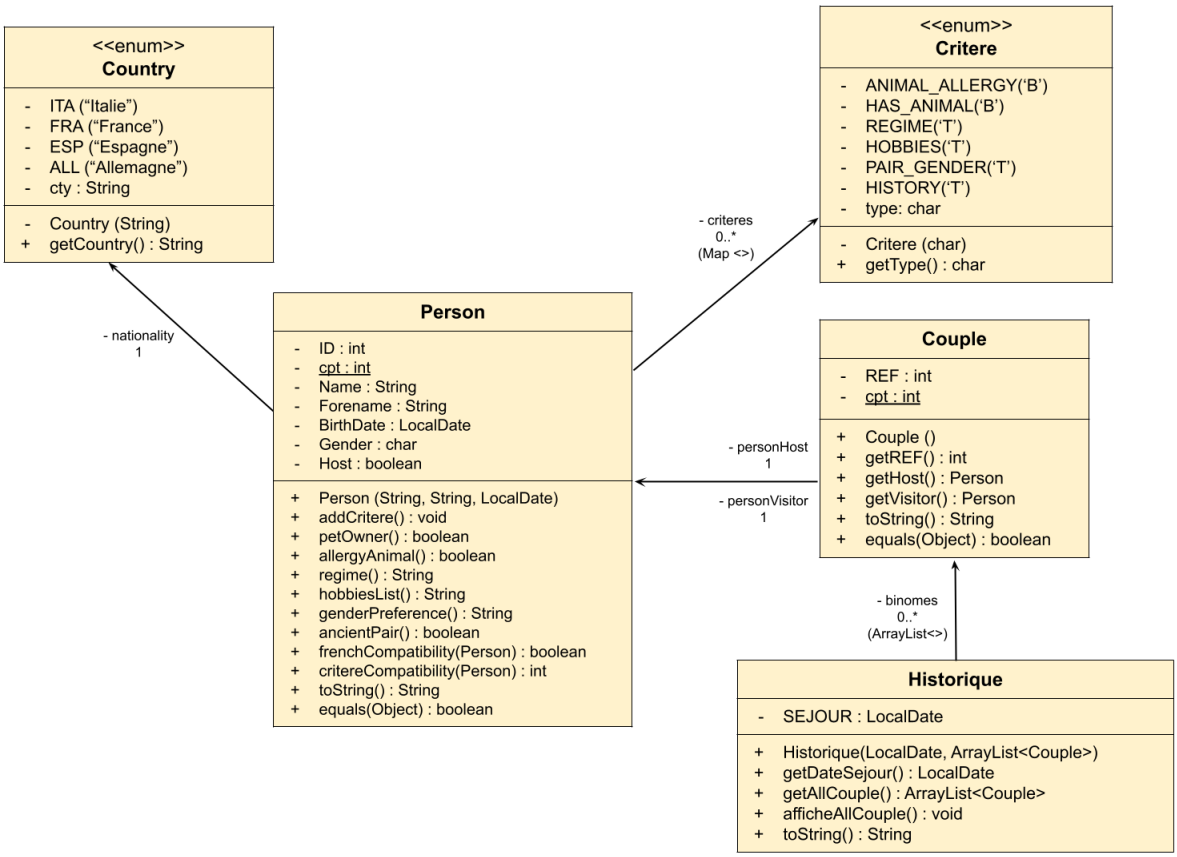
Remarques

Lors de la réalisation de cette première semaine de projet, nous avons rencontré des difficultés quant à la gestion du temps. Par souci de "bien-faire", nous nous sommes attardés sur la conception de l'UML, ce qui nous a ralenti dans l'écriture des différentes classes et tests.
Actuellement, nous n'avons pas pu achever toutes les méthodes et tous les tests que nous voulions présenter.

Néanmoins, nous allons changer pour les semaines à venir notre mode de communication et de travail afin de gagner en efficacité dans l'écriture du code.

POO-v2

Modèle UML



Le modèle UML de cette version n'a pas été beaucoup modifié. En effet, nous avons uniquement rajouté une méthode *critereCompatibility()* permettant de vérifier les points communs entre les élèves.
Suite à l'ajout de cette dernière, nous avons réalisé quelques tests afin de s'assurer de la bonne fonctionnalité d'un certain nombre de méthodes.

Modifications

Nous avons réalisé dans cette seconde version des modifications sur notre méthode *equals(Object)* de *Person*, afin de s'assurer de la fiabilité de notre code.
Nous avons également créer la méthode *critereCompatibility()*. Celle-ci est d'ailleurs basée sur le pseudo-code que nous avons réalisé pour le rapport de Graphes (R2.07).
La réalisation de cette méthode n'était pas aisée dans un premier temps. Effectivement, il était nécessaire de s'organiser et de faire les bons choix concernant le calcul d'affectation de chaque paire, aussi il fallait s'assurer que la compréhension des consignes était en raccord avec les résultats excomptés.

Tests

Comme évoqué plus haut, nous avons réalisé des tests concernant la classe *Person*.
Les tests concernent les méthodes suivantes :

- *equals()*
- *ancientPair()*
- *petOwner()*
- *allergyAnimal()*
- *frenchCompatibility()*
- *critereCompatibility()*

Ils ont été codé dans une classe spécifique : *PersonTest*, disponible dans le répertoire `R2.01-03-Dév/test/SAE`.
Deux *Person* ont été initialisée de manière à tester chaque méthode.

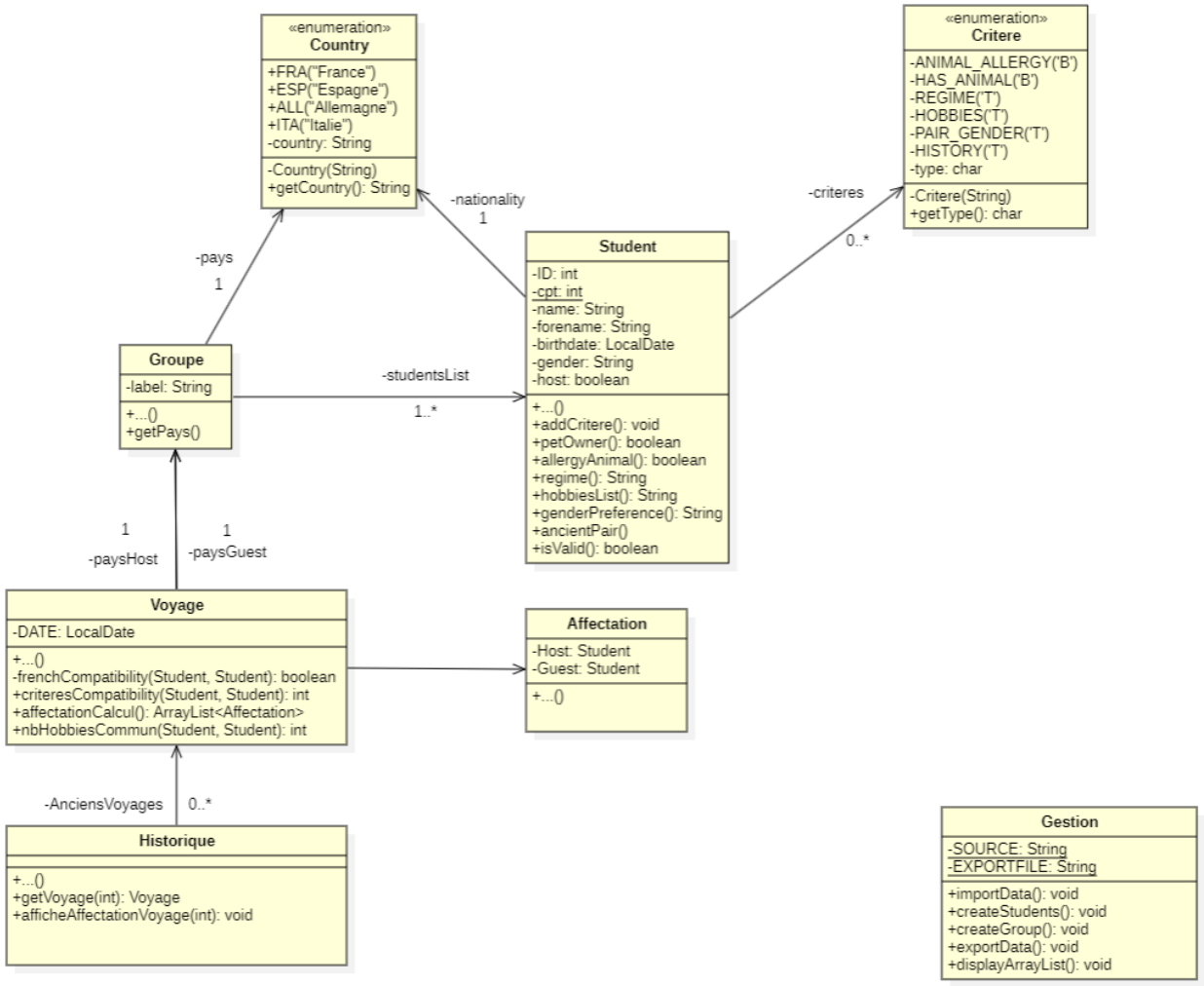
Suite à ces tests, nous avons du réécrire la dernière méthode (*critereCompatibility()*) qui ne renvoyait pas du tout les résultats attendus.

Remarques

Pour cette version, nous n'avons pas réussi à proposer des "try-catch" pertinents. En effet, l'équipe n'a pas encore saisi cette notion et malgré quelques essais, il était difficile de trouver un intêret par rapport aux méthodes créées et décrites plus haut.

POO-v3

Modèle UML



Quand on se réfère à notre UML ci-dessus, nous pouvons contaster que lors de cette version celui-ci a beaucoup changé.
En effet, suite aux échanges réalisés avec notre professeur référent du projet, nous nous sommes rendu compte que notre façon de modéliser n'était pas la bonne.

Modifications

Ajouts

Dans le but d'avoir une meilleure modélisation, nous avons ajouté les classes suivantes :

- *Groupe* : classe regroupant les élèves par pays d'origine.
- *Voyage* : qui rassemble un *Groupe* d'élèves d'un pays hôte et un *Groupe* d'élèves d'un autre pays visiteur. Un voyage est identifiable grâce à une date (de début de séjour). Cette classe permet également de calculer les affectations pour créer les binômes d'adolescents.
- *Affectation* : la classe *Affectation* permet de créer les binômes d'élèves.
- *Gestion* : Cette dernière classe a pour but d'importer des données, de créer les instances d'élèves et de groupes, et d'enfin d'exporter des données.

Nous avons également ajouté les méthodes qui suivent :

- *isValid()* : dans la classe *Student* qui permet de vérifier si les profils d'élèves sont cohérents.
- *affectationCalcul()* : calculant l'affectation des élèves dans la classe *Voyage*. Cette affectation est basée sur le score de compatibilité obtenus grâce à la fonction *criteresCompatibility()*. Elle reprend la logique de l'algorithme hongrois.
- *importData()* et *exportData()* : qui sont utilisées pour l'importation et l'exportation des données.
- *createStudent()* et *createGroupe()* : qui créent respectivement des élèves et des groupes d'élèves grâce aux différentes données.
- *displayArrayList()* : qui sert simplement à afficher des listes.
- *nbHobbiesCommuns()* : dans *Voyage*, renvoyant le nombre d'**Hobbies** en communs.

Adaptations

Comme nous avons ajouté des classes, il a fallu adapter les autres classes déjà présentes.
Parmi ces classes il y avait :

- *Student* (anciennement *Person*) : qui permet de créer uniquement un élève et qui ne contient que des constructeurs, getters et setters.
- *Historique* : qui permet actuellement d'enregistrer une liste de voyage (donc d'avoir une trace des différents voyages).

Des fonctions ont également étaient déplacées, notamment : *frenchCompatibility(Student, Student)* et *criteresCompatibility(Student, Student)* qui sont désormais dans Voyage et prennent en paramètres des *Student*.

Suppressions

Suite à tout ces changements, nous avons décidé de supprimer la classe suivante Couple.
En effet, celle-ci ne nous parraissait plus cohérante, sachant que la classe *Affectation* reprend ce principe de binômes.

Tests

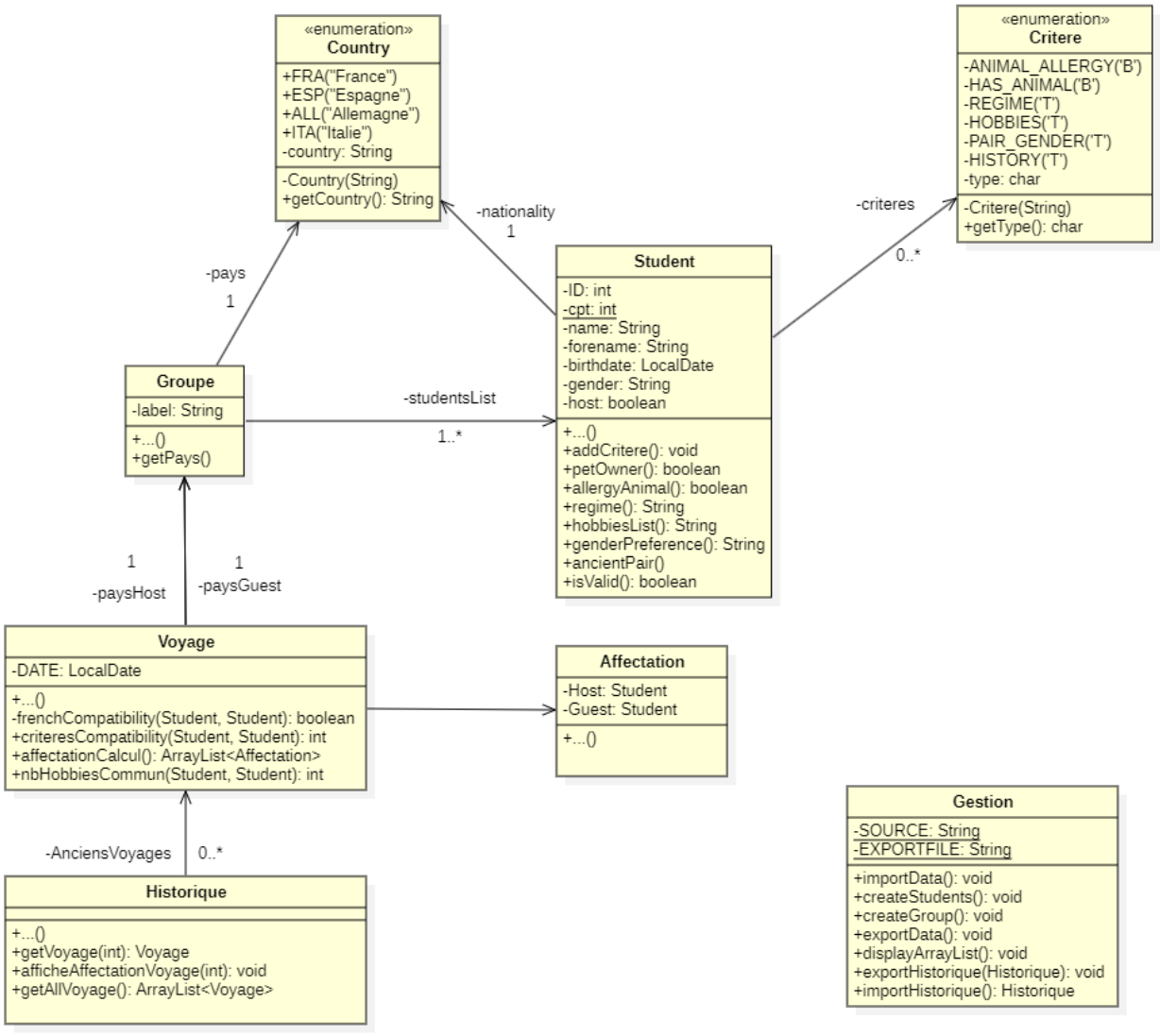
Différents tests ont été réalisés afin de valider ces modifications, ces tests portaient notamment sur les méthodes suivantes :

- *affectationCalcul()*
- *isValid()*
- *frenchCompatibility()*
- *nbHobbiesCommuns()*

Remarques

Il a été particulièrement compliqué dans cette version de remodéliser correctement notre UML, afin de progresser vers une proposition de programme plus pertinente et plus complète. Nous avons pour cela demandé conseils à notre professeur référent pour qu'il puisse nous guider dans les modifications.
De même, la conception de la méthode de calcul d'affectation a été une tâche difficile à réaliser. En effet, la compréhension de l'algorithme hongrois n'a pas été évidente et notre méthode actuelle a encore des limites, notamment sur les différences de nombre d'élèves entre groupes hôte et visiteur.

POO-v4



Lors de cette version, notre UML n'a pas réellement changé, nous avons uniquement rajouté des méthodes de sérialisation dans notre classe *Gestion* et nous avons modifié notre classe *Historique* afin qu'elle puisse être mieux adaptée à l'UML.
Pendant cette session de travail, nous nous sommes plutôt consacré à l'interface pour l'IHM. Nous avons ensuite associé notre code Java à notre code IHM, mais nous avons rencontré des difficultés.

Modification

Adaptations

Dans la classe *Historique* :
* L'attribut et son constructeur, qui sont désormais basés sur une **ArrayList**.
* Les getters qui ont également été modifiés et qui renvoient désormais une **ArrayList** ou un *Voyage*.
* La méthode *toString()*, pour mettre à jour l'affichage.

Dans la class *Voyage* : * Prise en compte des critères rédhibitoires (allergies alimentaires et animales) dans la méthode *criteresCompatibility()*.

Ajouts

Dans la classe *Gestion* : * La méthode *exportHistorique(Historique h)* a été ajoutée afin de pouvoir générer une sauvegarde/archive de l'historique de tous les voyages passés. Cette archive est enregistrée sous le format *.json* et sérialise l'entièreté des classes de notre UML.
* Sur la même logique, nous avons codé une méthode *importHistorique()* qui permet de récupérer une sauvegarde d'un fichier en *.json*.

Remarque

Pour ce rendu POO-v4, nous n'avons pas pu réaliser toutes les modifications que nous aurions voulu. En effet, la réalisation de la partie IHM a été très chronophage et nous n'avons pas réussi à gérer le temps correctement. Nous avons essayé de prioriser les méthodes qui nous semblaient les plus importantes.
Suite aux ajouts, nous avons également généré la documentation qui est disponible dans le répertoire `R2.01-03_Dév/doc/SAE/`.