SAÉ S2.01/02

Ethan Robert, Jonas Facon, Antonin Marouzè

Mai 2025

Contents

1 Structure du projet

```
.
App/
    lib/
        Bibliothèques utiles au projet
    src/
        model/
            Bibliothèques back-end, partie POO
        ui/
            Interface utilisateur JavaFX, parie IHM
    test/
        Programmes de test
        data/
            Samples de test (fichiers CSV)
Graphes/
    Rapport de graphes
Doc/
    gen/
```

```
    assets/
        Images
        Diagrammes UML
    Fichiers sources & makefile des rapports
    Prototypes d'interface IHM
rapport.pdf
```

1.1 Pourquoi séparer les parties POO et IHM ?

Cette division vient d'une envie de propreté dans la structure principale du projet. Ainsi, le package dédié à l'interface utilisateur contient essentiellement les classes utiles au logiciel graphique, et ne fait donc qu'importer et utiliser les librairies contenues dans `model`.
Cela permet également d'isoler la partie back-end pour les tests.

2 Rendu des Semaines 1 et 2

Pour cette première semaine.

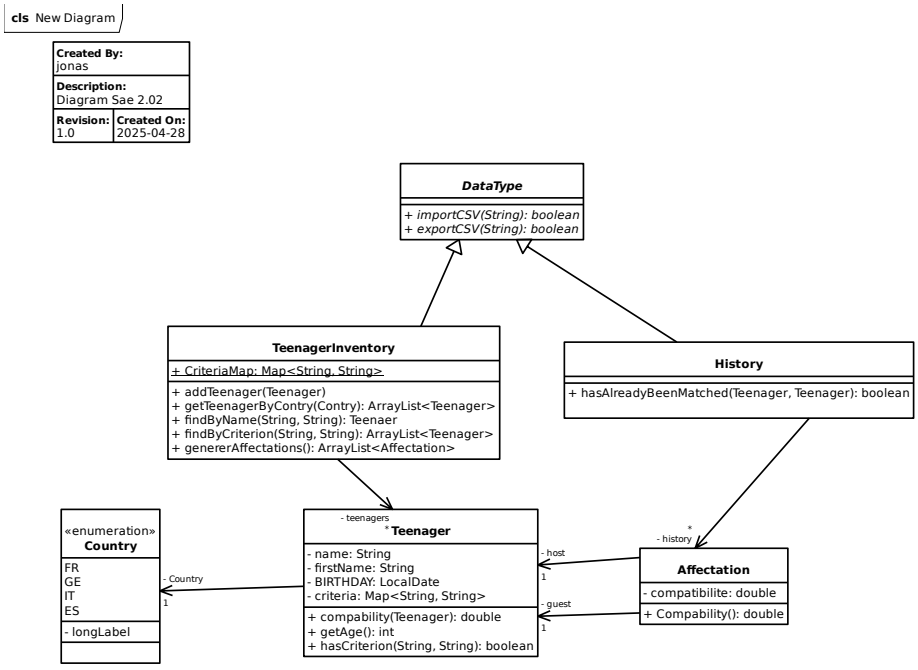

Figure 1: Diagramme UML du projet

2.1 Pourquoi ce diagramme UML ?

Nous avons décidé d'intégrer ce diagramme pour deux principales raisons : - **Plannifier** correctemen- `FOOD_ALLERGY` : La contrainte de régime alimentaire n'est pas respectée. Le régime alimentaire de l'invité n'est pas le même que celui de l'adolescent hôte'importation et exportation de ses données au format CSV (via deux méthodes abstraites `boolean importCSV(String)` et `boolean exportCSV(String)`), utilisées pour gérer les fichiers externes.

- La classe `TeenagerInventory` encapsule plusieurs objets de type `Teenager` au sein d'un `ArrayList<Teenager>`, permettant ainsi de gérer de manière centralisée tous les adolescents de la plateforme d'échanges. En dehors de cela, son attribut principal est `CriteriaMap<String, String>`, qui renseigne tous les critères existants, ainsi que leurs types (`T` pour du texte, `B` pour des booléens, etc). Elle possède également différentes **méthodes** :

  1. `addTeenager(Teenager)`
     Ajoute un objet **Teenager** à la `ArrayList`.
  2. `getTeenagersByCountry(Country)`
     Retourne la liste des **Teenagers** dont le pays correspond à celui passé en paramètre.
  3. `Teenager findByName(String, String)`
     Retrouve un **Teenager** à partir de son prénom et de son nom.
  4. `findByCriterion(String, String)`
     Renvoie tous les **Teenagers** correspondant à un même critère (ex. : "ville", "école", etc.).
  5. `genererAffectations()`
     Génère et retourne une `ArrayList` d'**Affectation**.

- La classe `Teenager` qui modélise chaque adolescent, à travers des **attributs** :

  1. `private String name` : le nom de l'adolescent
  2. `private String firstName` : le prénom de l'étudiant
  3. `private final LocalDate BIRTH` : une constante pour la date de naissance de l'adolescent
  4. `private Map<String, String> criteria` : une Map permettant d'inscrire les critères de compatibilité (rhédibitoires ou non), en suivant le modèle établi par `TeenagerInventory.CriteriaMap`

- Mais également des **méthodes** :

  1. `double compatibility(Teenager)` qui donne le score de compatibilité probable avec un autre adolescent, compris entre 0 et 1.
  2. `int getAge()` qui calcule l'âge d'un adolescent

3. `boolean hasCriterion(String, String)` qui retourne un booléan suivant si un adolescent possède un critère ou non.

- La classe `Affectation` qui permet d'encapsuler 2 objets `Teenager` (1 hôte et 1 visiteur) avec une méthode `double compatibility()`, donnant un score (compris entre 0 et 1) de compatibilité entre ces deux adolescents.
- La classe `History` permet d'ordonner le tout, en fournissant une `ArrayList<Affectation>` d'affectations déjà effectuées.

2.2 Enum et objets supplémentaires

- L'enumération `Country` contient les différents pays sous format abbréger elle contient comme méthode :
  1. `longLabel`Renvoie le nom complet d'un Pays.

3 Semaine 3 et 4

Voici tout d'abord le diagramme UML pour ces deux semaines.

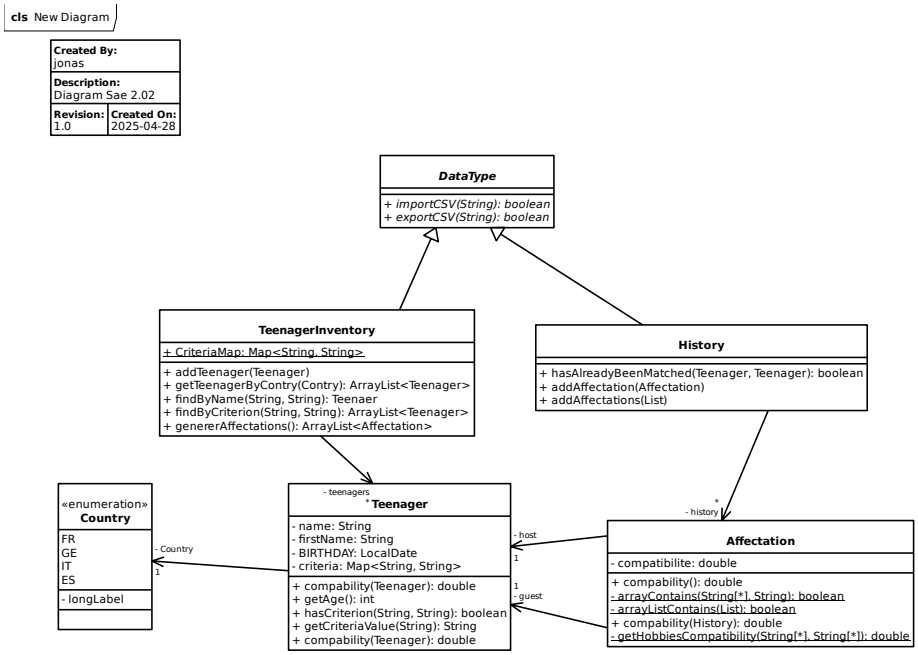

Figure 2: Diagramme UML du projet semaines 3 et 4

3.1 Changements principaux

Voici une liste des principaux changements dans la structure du code pour ces semaines.

3.1.1 Structure principale du code

- Le code est maintenant défini dans un package `model`. Cela permettra par la suite d'implémenter l'interface utilisateur dans un autre package, `model` ayant pour but de fournir le **back-end du projet** (une sorte d'API). Ainsi, une interface graphique peut **ne pas être le seul moyen d'interagir avec le logiciel**, et on pourrait imaginer d'autres types d'interfaces - comme une interface en ligne de commandes (TUI).

3.1.2 Package `model` : structure back-end du projet

- La classe `Affectation` contient maintenant une autre signature pour la méthode `compatibility()`, sans argument `History h`. L'utilisation de cette méthode sans cet argument implique que **l'historique d'affectation n'est pas pris en compte dans le calcul de la compatibilité entre deux étudiants**
- La méthode `compatibility()` est également appelée dans le constructeur de la classe `Affectation`. Le constructeur prend donc désormais ces deux signatures :
  - `Affectation(Teenager host, Teenager guest, History h)`
  - `Affectation(Teenager host, Teenager guest)` (l'historique n'est donc pas pris en compte dans le calcul de la compatibilité)
- Un nouveau type d'exception a été implémentée : `AffectationException`. Elle prend un argument `String reason`, qui représente la raison de la levée de l'exception. Une fonction `void printTrace()` permet d'afficher la raison de la levée de l'exception.
  Nous avons fait ce choix pour plus de flexibilité, permettant de lever n'importe quel type d'exception (toujours relative à la compatibilité entre deux adolescents)
- La méthode `compatibility()` de la classe `Affectation` lève désormais des `AffectationException` si certaines contraintes rédhibitoires ne sont pas respectées. Celles-ci peuvent avoir plusieurs raisons :
  - `FOOD_ALLERGY` : La contrainte de régime alimentaire n'est pas respectée. Le régime alimentaire de l'invité n'est pas le même que celui de l'adolescent hôte
  - `ANIMAL_ALLERGY` : La contrainte d'allergie aux animaux n'est pas respectée. L'adolescent invité est allergique aux animaux, alors que l'adolescent hôte en possède un.
  - `HISTORY` : La contrainte de l'historique n'est pas respectée :

    - L'hôte et l'invité n'ont jamais été mis ensemble, mais aucun des deux ne souhaite être affecté à quelqu'un d'autre (`same`)
    - L'hôte et l'invité ont déjà été mis ensemble, mais au moins l'un d'entre eux souhaite être avec quelqu'un d'autre (`other`)

- La classe `Affectation` contient quelques méthodes statiques privées servant dans `compatibility()`, notamment `boolean ArrayContains(String[] array, String element)` et `boolean ArrayListContains(ArrayList<String> arrayList, String element)`, qui permettent de savoir si (respectivement) un tableau ou une arraylist contiennent un élément String spécifié. Il y a également une méthode `double getHobbiesCompatibility(String[] hobbies_a, String[] hobbies_b)`, qui permet de déporter le calcul de compatibilité des passe-temps des adolescents dans la méthode `compatibility()`.
- La classe `Teenager` contient maintenant une méthode `String getCriteriaValue(String criterion)`, qui permet d'obtenir la valeur d'un critère donné. Elle retourne une string vide (`""`) si la clé spécifiée n'existe pas.
- La classe `Teenager` implémente une méthode `double compatibility(Teenager other)`, qui crée une affectation temporaire avec un autre adolescent, et renvoie le score de compatibilité de l'affectation
- La classe `History` implémente maintenant de nouvelles méthodes pour la gestion de l'historique d'affectation :
    - La méthode `void addAffectation(Affectation affectation)`, qui permet d'ajouter une affectation dans l'historique.
    - La méthode `void addAffectations(ArrayList<Affectation> affectations)`, qui permet d'ajouter plusieurs affectations directement en passant une ArrayList d'affectations en argument.