Compte rendu SAE2-01 - POO

DAGNEAUX Nicolas - DEGRAEVE Paul - MARTEL Alexandre - Groupe D1

1 Introduction

Dans le cadre de notre BUT Informatique, nous avons comme projet la création d'une application d'aide à la prise de décision lors d'échanges scolaires dans différents pays. Ce rapport se concentrera sur la partie *Programmation Orientée Objet*.

Notre projet est disponible sur notre page Gitlab.

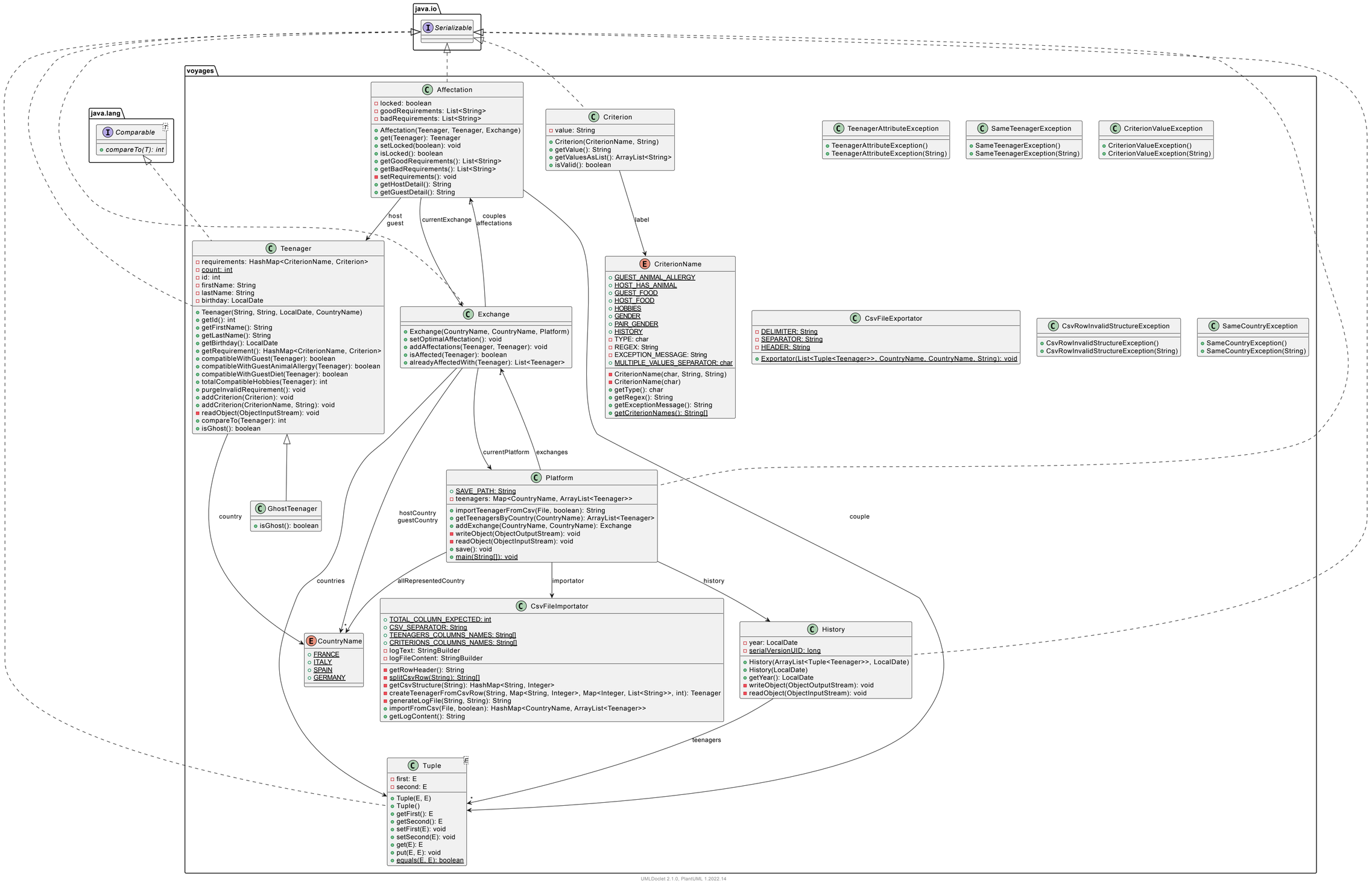
1.1 Notice d'utilisation

Pour exécuter l'application, il suffit d'une ligne de commande :

```
java -jar Affectator-SIMULATOR-3000-CLI.jar
```

2 Description fonctionnelle de l'application

2.1 Diagramme UML



Voici l'UML de notre package voyages, qui regroupe toutes les classes utilisées pour la partie POO.

2.2 Analyse Méchanisme objet

La classe principale de ce package est la classe `Platform`. Cette classe contient un dictionnaire d'adolescents associés à leur pays, un `History` pour pouvoir sauvegarder une affectation, ainsi qu'une chaîne de caractère contenant le chemin de sauvegarde de cette platforme. Cette classe implémente donc `java.io.Serializable`, pour pouvoir être sauvegardée.

Les pays d'origines des adolescents sont implémentés sous la forme d'une énumération des noms de pays, `CountryName`.

Les adolescents sont eux représentés par la classe `Teenager`. C'est la 2eme classe la plus importante car elle contient toutes les informations concernant un adolescents (nom, prénom, critère...), et elle contient aussi toutes les méthodes permettant de gérer les adolescents (la compatibilité). Cette classe implémente donc `java.io.Serializable`, pour pouvoir être sauvegardée, mais aussi `java.lang.Comparable`. En effet, il est essentiel de pouvoir comparer, et donc trié, les adolescents.

La classe `GhostTeenager` hérite de Teenager est sert uniquement à créer des adolescents fantômes pour équilibrer les listes d'adolescents lors de l'affectation à l'aide de la création d'un graphe. Elle contient une unique méthode `isGhost()` qui renvoie toujours vrai (`Teenager` possède aussi cette méthode qui elle renvoie toujours faux).

Les critères des adolescents sont stockés dans un dictionnaire de noms de critères associés à ce critère. Les noms de critère sont stockés dans une énumération `CriterionName`.

Quant aux critères en eux-même, ils sont représentés par la classe `Criterion`. Cette classe contient un seul attribut, la valeur du critère que l'adolescent rentre dans le formulaire. Cette classe implémente aussi `java.io.Serializable`.

La classe `Exchange` sert à créer un échange entre 2 pays. Elle fait appel aux méthodes de graphe pour déterminer l'affectation optimale sous la forme d'une liste de `Tuple` d'adolescents. Elle possède aussi un attribut `countries` qui est un `Tuple` de pays représentant le pays hôte et le pays visiteur.

Pour gérer les affectations, nous avons créer la classe `Affectation`. Cette classe sert à gérer plus précisément une affectations entre 2 adolescents. Elle permet notamment de bloquer une affectation entre 2 adolescents pour être sûr qu'il soit ensemble. Elle a donc comme attribut un `Tuple` d'adolescents.

La classe `Tuple<E>` est une classe générique que nous avons créer pour pouvoir stocker 2 éléments de même type. Cette classe possède donc des méthodes adaptées pour mieux répondre à nos besoins. Pour pouvoir enregistrer le résultat d'une affectation, nous avons créer la classe `History`. Cette classe contient donc une liste de `Tuple` d'adolescents.

Pour pouvoir importer et exporter des fichiers au format csv nous avons créer les classes `CsvFileImportator` et `CsvFileExportator`.

3 Analyse technique

3.1 Validité des critères

La vérification de la validité des critères se fait grâce à la méthode `purgeInvalidRequirement()` de la classe `Teenager` qui fait appel à la méthode `isValid()` de la classe `Criterion`.

La chose la plus compliqué à mettre en place est le fait de s'assurer que la valeur d'un critère soit valide.

Premièrement, nous avons fait en sorte que l'attribut `value` de la classe `Criterion` ne puisse jamais être `null`, afin de ne pas avoir à gérer ce cas de figure. Ensuite, nous avons choisi d'utiliser une expression régulière pour définir la validité de la valeur d'un critère. Bien que cela représente une difficulté certaine, cela nous a paru comme un bon choix puisque nous n'avions pas à écrire beaucoup de code :

```
public boolean isValid() throws CriterionValueException {
        boolean isValid = Pattern.matches(this.label.getRegex(),
        this.value);
        if (!isValid) {
            throw new
        CriterionValueException(this.label.getExceptionMessage()
        + (this.value.isEmpty() ? "empty" : this.value));
        }
        return true;
    }
```

La résultat de la méthode `isValid()` repose uniquement sur la valeur de l'expression régulière.

De plus, il y avait peu de valeur différentes acceptables, nous avions donc une expression régulière assez réduite. D'autre part, si nous devions par exemple intégrer le fait d'accepter d'autres régimes alimentaires, nous aurions pu écrire une méthode qui puisse générer une expression régulière à partir d'une liste de valeur correctes.

3.2 Compatibilité avec les français

Pour la compatibilité avec les français, la méthode `compatibleWithGuest` de la classe `Teenager` regarde si l'un des 2 adolescents est français et, dans le cas échéant, renvoie faux si le nombre de passe-temps en commun est inférieur à 1.

Nous avions pensé à étendre la classe `Teenager`, pour créer une classe `FrenchTeenager`, et surcharger la méthode `isValid`. Néanmoins, cela ne faisait pas de sens, puisque la méthode `compatibleWithGuest` était appelé depuis l'hôte, et que cette incompatibilité avec un Français doit être prise en compte si l'un des deux adolescents est français.

Nous avons donc choisi de vérifier dans la méthode `compatibleWithGuest` la présence d'un Français. De plus, la méthode `totalCompatibleHobbies` était utile ailleurs, et ne servait donc pas uniquement à cela.

```
if (this.country == CountryName.FRANCE ^ teenGuest.getCountry()
        == CountryName.FRANCE) {
            //check hobbies compatibility only with french people
            if (this.totalCompatibleHobbies(teenGuest) == 0) {
                return false;
            }
        }
    ...
```

3.3 Imports/exports

Premièrement, nous avons choisi de créer des classes utiles pour ces actions pour ne pas trop surcharger la classe `Platform`.

Cela permet de gérer le fait que l'ordre des colonnes ne soient pas fixes. Tout d'abord, nous allons faire correspondre chaque nom de colonnes

La gestion de l'import et de l'export de fichier au format csv se fait grâce aux classes `CsvFileImportator` et `CsvFileExportator`. `CsvFileImportator` regarde le nombres de colonnes du fichier et si tout va bien, créer un `Teenager` avec les informations données. `CsvFileExportator` possède en attribut le header du fichier et créer un fichier CSV avec la liste de `Tuple` d'adolescents donnée en paramètre ainsi que le pays hôte et le pays visiteur.

3.4 Historique

La sauvegarde de l'historique se fait grâce à la classe `History`. Cette classe possède 2 attributs, une liste de `Tuple` d'adolescents et une année. Cette classe est sérialisable grâce aux méthode `writeObject()` et `readObject()`.

3.5 Prise en compte de l'historique

La prise en compte de l'historique se fait grâce à l'attribut `history` de `Platform`. Cet historique est ensuite donnée aux méthodes de graphes qui s'occupent de gérer les affectations optimales.

4 Analyse des tests

Nous avons testé 2 grands pans de notre application, car ce sont les plus critques.

4.1 Criterion

Dans la classe de tests `CriterionTest` nous testons essentiellement la méthode `isValid` pour absolument tous les criterions. Nous avons néanmoins créer une fonction pour chaque type. Par exemple `TestFood` qui va être testé avec toutes les énumération `CriterionName` qui se réfère à la nourriture. Nous avons agis de cette manière pour tous les critères.

4.2 Teenager

Dans la classe de tests `TeenagerTest`, nous testons tout ce qui se réfère à la méthode `compatibleWithGuest`. Nous avons séparé ces tests en plusieurs parties, une par type de compatibilité : allergies, régimes, hobbies...

Là aussi, nous testons absolument toutes les combinaisons possibles.