



UNIVERSITÉ DE LILLE

SAÉ S2.01-02

PROGRAMMATION ORIENTÉE OBJET

RAPPORT

---

## Conception & Développement

---

***Élèves :***

Baptiste LAVOGIEZ

Mark ZAVADSKIYI

Angèl ZHENG

***Enseignant :***

Antoine NONGAILLARD

Fabien DELECROIX

13 juin 2025

A decorative graphic in the bottom right corner of the page, consisting of a grey, folded paper-like shape.

# Table des matières

<b>1</b>	<b>Rendu 1</b>	<b>3</b>
1.1	Réflexion . . . . .	3
1.2	UML . . . . .	3
1.3	Remarques . . . . .	3
1.4	Implémentation . . . . .	4
<b>2</b>	<b>Rendu 2</b>	<b>7</b>
2.1	Attentes . . . . .	7
2.2	Réflexion . . . . .	7
2.3	UML . . . . .	7
2.4	Modifications de <b>Person</b> . . . . .	8
2.5	Modifications de <b>Criteria</b> . . . . .	9
2.6	Modifications de <b>Exchange</b> . . . . .	12
2.7	Classes de Test . . . . .	13
<b>3</b>	<b>Rendu 3</b>	<b>15</b>
3.1	Attentes . . . . .	15
3.2	Réflexion . . . . .	15
3.3	UML . . . . .	16
3.4	Nouveau dossier : res . . . . .	18
3.5	Nouvelle classe : CountryVisit . . . . .	19
3.6	Nouvelle classe : PeopleManager . . . . .	23
3.7	Modifications de Person . . . . .	35
3.8	Modifications de Criteria . . . . .	37
3.9	Modifications de Exchange . . . . .	38
3.10	Classes de Test . . . . .	46
3.11	Conclusion et critiques de ce rendu . . . . .	53
<b>4</b>	<b>Rendu 4</b>	<b>54</b>
4.1	Attentes . . . . .	54
4.2	Réflexion . . . . .	54
4.3	UML . . . . .	55
4.4	Nouvelle réflexion . . . . .	56
4.5	Modifications de PeopleManager . . . . .	57
4.6	Nouvelle classe : CriteriaConfigValidator . . . . .	61
4.7	Exemples de prétraitement . . . . .	65
4.8	Modifications de Person . . . . .	67
4.9	Modifications de CountryVisit . . . . .	67
4.10	Classes de Test . . . . .	68
4.11	Conclusion . . . . .	68

Ce rapport présentera la partie **Conception & Développement** de la SAE S2.01-02.  
Elle traitera d'une solution à la situation d'échange scolaire entre deux personnes régie  
par des critères et exigences formulées ou non.

# 1 Rendu 1

Ce premier rendu traitera de la conception des classes des bases, des contraintes et des classes de tests.

## 1.1 Réflexion

Cette semaine, nous avons à poser les bases de notre projet. Nous réfléchissons d'abord aux structures les plus adaptées pour le besoin, qui est ici d'échanger des élèves entre eux sur la base de critères.

Nous avons besoin d'une Personne, de Critères stockés avec leur types, et d'un Echange entre deux Personnes. Cela se matérialise avec un UML.

## 1.2 UML

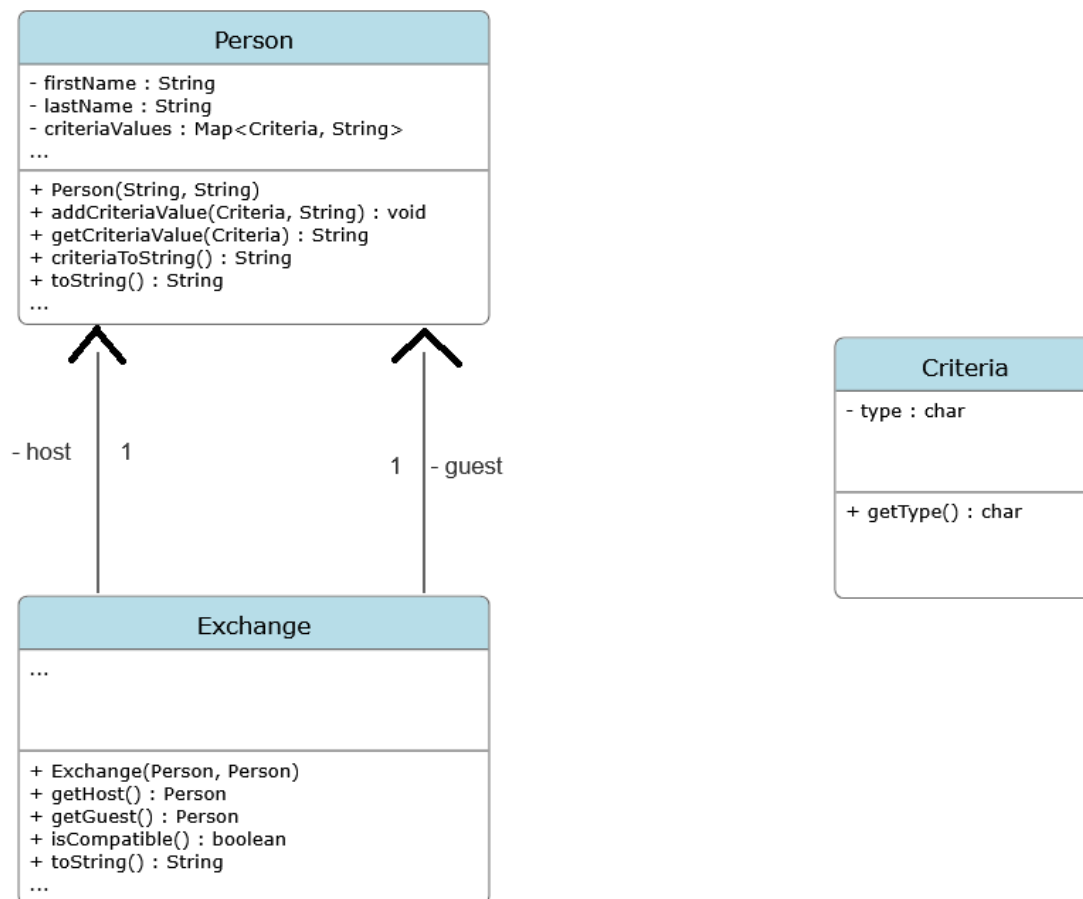


FIGURE 1 – Relations UML

## 1.3 Remarques

Nous avons décidé d'une représentation minimaliste mais avec tous les éléments nécessaires. Le code est facilement compréhensible avec des noms explicites.

### 1. Des critères...

Criteria est une énumération simple de critères associés à un type représenté sous forme de caractère.

Cette énumération est utilisée dans `criteriaValues`, la Map de la classe `Person`. Elle y sera associée à une valeur représentée en chaîne de caractères pour simplifier la modélisation. Dans `Person`, les méthodes adéquates permettent de manipuler ces critères (ajout, obtention, conversion en texte...).

## 2. Person ou Student ?

Ici, le fait que les personnes soient des étudiants n'est pas important. Choisir le modèle d'un étudiant reviendrait à devoir encapsuler une personne dans la classe, soit une classe de plus.

## 3. Le double rôle

Une personne peut aussi bien être un hôte qu'un invité. Pour cela, `Exchange` encapsule les deux afin d'avoir une combinaison claire et simple à manipuler.

## 4. Critiques

La classe `Exchange` aurait pu, de façon plus ou moins simple, se raccourcir en étant définie dans `Person`. Le problème étant que de cette façon, le rôle de `Host` et de `Guest` devrait être défini autrement. Nous préférons `Exchange` pour fonctionner de manière plus simple afin de manipuler clairement les rôles.

## 1.4 Implémentation

L'implémentation se fait de manière naturelle après avoir tout défini de façon claire. Des tests élargis sont disponibles dans le dossier sous-jacent et permettent de voir comment le modèle réagit à toutes les situations.

### 1. Aperçu de Exchange

## Java

```
1 package v1;
2
3 public class Exchange {
4     private Person host;
5     private Person guest;
6
7     public Exchange(Person host, Person guest) {
8         this.host = host;
9         this.guest = guest;
10    }
11
12    public Person getHost() {
13        return this.host;
14    }
15
16    public Person getGuest() {
17        return this.guest;
18    }
19 }
```

## 2. Aperçu de ExchangeTest

## Java

```

1 package v1 ;
2
3 public class ExchangeTest {
4
5     private Person alice;
6     private Person bob;
7     private Exchange exchange;
8
9     @BeforeEach
10    public void initTest() {
11        alice = new Person("Astana", "Nur-sultan");
12        bob = new Person("Bob", "Marley");
13        exchange = new Exchange(alice, bob);
14    }
15
16    @Test
17    public void testIncompatibleDueToAnimalAllergy() {
18        alice.addCriteriaValue(Criteria.HOST_HAS_ANIMAL, "true");
19        bob.addCriteriaValue(Criteria.GUEST_ANIMAL_ALLERGY, "true");
20
21        assertFalse(exchange.isCompatible());
22    }
23
24    @Test
25    public void testIncompatibleDueToGenderPreference() {
26        alice.addCriteriaValue(Criteria.HOST_HAS_ANIMAL, "false");
27        bob.addCriteriaValue(Criteria.GUEST_ANIMAL_ALLERGY, "false");
28        alice.addCriteriaValue(Criteria.PREFERENCE_GENDER, "Female");
29        bob.addCriteriaValue(Criteria.GENDER, "Male");
30
31        assertFalse(exchange.isCompatible());
32    }

```

## 2 Rendu 2

### 2.1 Attentes

Cette semaine, nous traiterons de :

- Gestion de la validité des critères par un mécanisme d'exception
- Développement des règles spécifiques de compatibilité pour certains pays

La règle spécifique de compatibilité étant, pour le moment, celle énonçant qu'un échange comprenant une personne française devait avoir au moins un hobby en commun sans quoi le risque de cassure serait trop important.

La validité des critères portera sur :

- La bonne entrée d'un type (car jusqu'ici ils sont tous en String) ; exemple : `HOST_HAS_ANIMAL` doit être à `true` ou `false`.
- La cohérence des valeurs entrées. Par exemple une `Person` ne peut pas être allergique aux animaux et en avoir un.

### 2.2 Réflexion

Nous avons réfléchi à comment implémenter les critères spécifiques aux pays.

Nous avons pensé à :

- Une énumération `SpecificCountryRule`
- Une `ArrayList` de "rules" au sein de `Person`

Au final, nous avons opté pour une définition dans `Person` au sein de la liste de critères.

La validité des critères, elle, a été validée en divisant la vérification en deux temps : d'abord le bon respect du type prévu (pas de `yes` dans un boolean), pour ensuite vérifier la bonne valeur (pas de `toto` dans la date de naissance).

### 2.3 UML

L'UML est, par conséquent, mis à jour et amélioré dans ses détails.



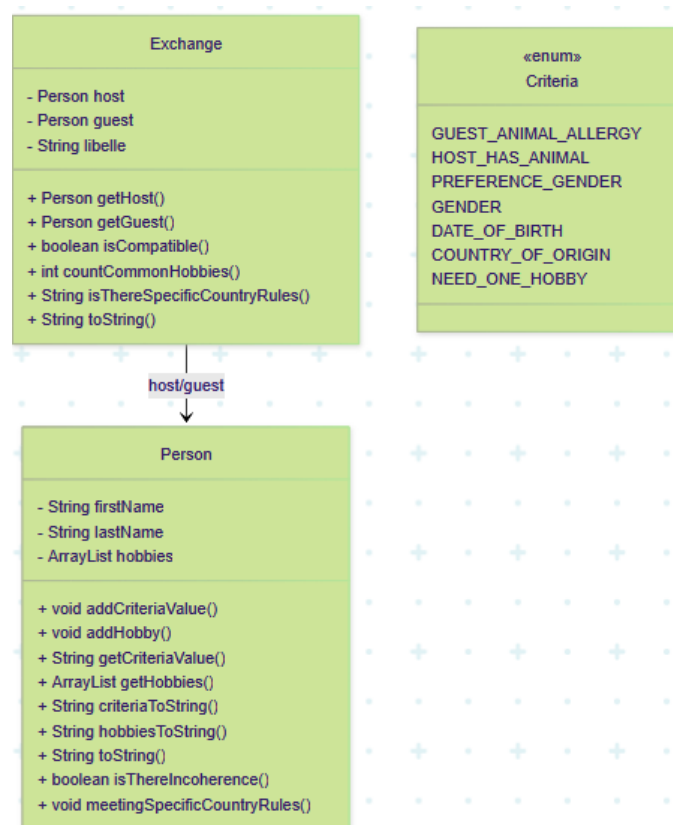


FIGURE 2 – Relations UML

## 2.4 Modifications de **Person**

Pour ce faire, nous avons procédé à l'implémentation d'une méthode `meetingSpecificCountryRules` dans la classe **Person**.

Elle est appelée dans le constructeur désormais surchargé et à l'ajout d'un nouveau critère. En somme, elle est appelée à chaque possible changement de critère pour adapter la bonne logique.

## Java

```

1 // Cette méthode change les critères en fonction du pays
2 // Elle doit être appelée à chaque fois qu'on change un critère
  (constructeur, ajout)
3 public void meetingSpecificCountryRules() {
4     try {
5         String country = this.getCriteriaValue(Criteria.
6             COUNTRY_OF_ORIGIN).toUpperCase();
7         if (country.equals("FRANCE")) {
8             this.addCriteriaValue(Criteria.NEED_ONE_HOBBY,
9                 "true");
10        }
11    } catch (NullPointerException e) {
12        // Si le pays n'est pas défini, on ne fait rien, ce n'
13        est pas vraiment une erreur
14    }
15 }

```

Dans la même idée et dans la même classe, une méthode `isThereIncoherence` a été ajoutée. Elle veille à faire respecter les contradictions de données; ici, il est sujet d'une personne allergique aux animaux mais en ayant un aussi en même temps.

## Java

```

1 public boolean isThereIncoherence() {
2     // On vérifie si la personne a un animal et est allergique
3     String hostHasAnimal = this.getCriteriaValue(Criteria.
4         HOST_HAS_ANIMAL);
5     String guestAllergy = this.getCriteriaValue(Criteria.
6         GUEST_ANIMAL_ALLERGY);
7
8     if ("true".equals(guestAllergy) && "true".equals(hostHasAnimal)
9         ) {
10        return true;
11    }
12    return false;
13 }

```

De plus, une `ArrayList` de hobbies a été ajoutée, en accord avec les critères relatifs à un nombre de hobbies en commun minimal.

## 2.5 Modifications de Criteria

Un critère a été ajouté : `NEED_ONE_HOBBY`, caractérisant le besoin dans un `Exchange` d'avoir au moins un hobby en commun en cas de règles spécifiques. Ce critère est un booléen ('B').

La vérification des critères se décompose en deux parties : type, puis valeur.

### 1. La méthode `static isCriteriaTypeValid`

## Java

```

1 public static boolean isCriteriaTypeValid(Criteria criteria, String
   value) {
2     char type = criteria.getType();
3     switch(type) {
4         case 'B':
5             value = value.toLowerCase(); // on met tout en
           minuscule
6             // On vérifie si la valeur est "true" ou "false"
7             return value.equals("true") || value.equals("false");
8         case 'T':
9             return (value.length() > 0); // TODO: Add validation for
           text criteria
10        case 'N':
11            try {
12                Integer.parseInt(value);
13                return true;
14            } catch (NumberFormatException e) {
15                System.out.println("Invalid number format: " +
16                    value);
17                System.out.println("Exception: " + e.getMessage
18                    ());
19                return false;
20            }
21        case 'D':
22    }
23    return false;
24 }

```

Cette méthode sera appelée, en complément de la suivante, à chaque nouvel ajout de critère. Si les méthodes ne sont pas satisfaites, l'ajout échoue.

## 2. La méthode static isCriteriaValueValid

## Java

```

1 public static boolean isCriteriaValueValid(Criteria criteria, String
   value) {
2     value = value.toLowerCase(); // on met tout en minuscule
3     // Mode en if car pas toutes les valeurs sont concernées
4     if(criteria==Criteria.PREFERENCE_GENDER || criteria==Criteria.
       GENDER) return value.equals("male") || value.equals("female
       ") || value.equals("other");
5     if(criteria==Criteria.DATE_OF_BIRTH) {
6         try { //Cette partie gestion d'exception est plutôt
              //à mettre lors de l'entrée des données.
7             return LocalDate.now().minusYears(18).isAfter((
              LocalDate.parse(value)));
8         } catch (DateTimeParseException e) {
9             System.out.println("Format invalide, utilisez
              le format suivant : yyyy-MM-dd.");
10            return false;
11        }
12    }
13    return true ;
14 }

```

### 3. La méthode static areCriteriaValid

## Java

```

1 // Vérifie l'ensemble des critères et leur validité. Si un n'est
  // pas valide, on renvoie false.
2 public static boolean areCriteriaValid(Map<Criteria, String> criterias
  ) {
3     for (Map.Entry<Criteria, String> entry : criterias.entrySet())
4     {
5         Criteria criteria = entry.getKey();
6         String value = entry.getValue();
7         try {
8             if (!isCriteriaTypeValid(criteria, value)) {
9                 System.out.println("Invalid type for
10                    criteria " + criteria + ": " +
11                    value);
12                 return false;
13             }
14         } catch (Exception e) {
15             System.out.println("Exception during type
16                validation for " + criteria + ": " + e.
17                getMessage());
18             return false;
19         }
20         try {
21             if (!isCriteriaValueValid(criteria, value)) {
22                 System.out.println("Invalid value for
23                    criteria " + criteria + ": " +
24                    value);
25                 return false;
26             }
27         } catch (Exception e) {
28             System.out.println("Exception during value
29                validation for " + criteria + ": " + e.
30                getMessage());
31             return false;
32         }
33     }
34     return true;
35 }

```

Cette méthode est appelée dans le constructeur prenant en entrée une la Map des critères de la Person.

## 2.6 Modifications de Exchange

### 1. La méthode countCommonHobbies

## Java

```

1 //Donne le nombre de centre d'intérêt en commun
2 //Dans le cadre du critère "NEED_ONE_HOBBY"
3 public int countCommonHobbies() {
4     ArrayList<String> hostHobbies = this.host.getHobbies();
5     ArrayList<String> guestHobbies = this.guest.getHobbies();
6     int commonHobbies = 0;
7     for (String hobby : hostHobbies) {
8         if (guestHobbies.contains(hobby)) {
9             commonHobbies++;
10        }
11    }
12    return commonHobbies;
13 }

```

Cette méthode est utilisée afin de satisfaire ou non le critère NEED\_ONE\_HOBBY.

## 2. Adaptation de isCompatible

## Java

```

1     public boolean isCompatible() {
2         ...
3         ...
4         String hostHobbies = this.host.getCriteriaValue(
5             Criteria.NEED_ONE_HOBBY);
6         String guestHobbies = this.guest.getCriteriaValue(
7             Criteria.NEED_ONE_HOBBY);
8         if (!this.host.getHobbies().isEmpty() && !this.guest.
9             getHobbies().isEmpty() && (hostHobbies.equals("true") || guestHobbies.equals("true"))) &&
10            countCommonHobbies() < 1){
11             return false;
12         }
13         return true;
14     }

```

Nous avons implémenté la gestion de la règle spécifiant que les paires avec une **Person** avec pour pays France, il faut au moins un hobby en commun. Pour ce faire, nous appelons la fonction précédemment définie `countCommonHobbies`

## 2.7 Classes de Test

Les classes `ExchangeTest` ainsi que `PersonTest` ont été mises à jour conformément aux ajouts.

De plus, la classe `CriteriaTest` fait son apparition.

## Java

```

1 public class CriteriaTest {
2
3     private Person alice;
4
5     @BeforeEach
6     void setUp() {
7         alice = new Person("Alice", "Smith");
8         alice.addCriteriaValue(Criteria.HOST_HAS_ANIMAL, "true"
9         );
10        alice.addCriteriaValue(Criteria.PREFERENCE_GENDER, "
11        Female");
12    }
13
14    @Test
15    void testCriteriaTypeValid() {
16        assertTrue(Criteria.isCriteriaTypeValid(Criteria.
17        HOST_HAS_ANIMAL, "true"));
18        assertFalse(Criteria.isCriteriaTypeValid(Criteria.
19        HOST_HAS_ANIMAL, "yes"));
20        assertTrue(Criteria.isCriteriaTypeValid(Criteria.
21        PREFERENCE_GENDER, "male"));
22        assertFalse(Criteria.isCriteriaTypeValid(Criteria.
23        COUNTRY_OF_ORIGIN, ""));
24    }
25
26    @Test
27    void testCriteriaValueValid() {
28        assertTrue(Criteria.isCriteriaValueValid(Criteria.
29        PREFERENCE_GENDER, "male"));
30        assertFalse(Criteria.isCriteriaValueValid(Criteria.
31        PREFERENCE_GENDER, "femme"));
32        assertTrue(Criteria.isCriteriaValueValid(Criteria.
33        DATE_OF_BIRTH, "2000-01-01"));
34        assertFalse(Criteria.isCriteriaValueValid(Criteria.
35        DATE_OF_BIRTH, "2020-01-01"));
36        assertFalse(Criteria.isCriteriaValueValid(Criteria.
37        DATE_OF_BIRTH, "je suis le 8 mai 1212 et je suis
38        mal écrit"));
39    }
40 }

```

## 3 Rendu 3

### 3.1 Attentes

Ces deux semaines de réalisation seront les plus denses !

Au programme, nous avons :

- Import par fichier de format csv, répondant à la structure donnée
- Export d'un résultat sous format csv
- Gestion de l'historique par sérialisation binaire
- Implémentation d'un jeu de données montrant le bon fonctionnement du nouveau système d'affectation

Bonne nouvelle, **nous avons tout réalisé !**

D'autres changements mineurs : Les commentaires dans le code ont été **réécrits en anglais** afin que le projet, une fois visible sur notre portfolio, ait une plus grande portée.

Avant d'aller tête la première dans notre IDE favori, il est important de réfléchir à comment nous allons implémenter ces attentes.

### 3.2 Réflexion

À l'issue de ce rendu, la partie POO de ce projet sera quasiment terminée. Ainsi, il est important de considérer les attentes de la partie IHM et de produire du code réutilisable dans JavaFX.

Traitons les attentes une par une et trouvons-y une procédure de traitement :

#### 1. Import de CSV

Nous devons ici lire un CSV contenant les personnes sujettes à de futurs échanges !

La forme du CSV est donnée dans le sujet à la fin. Une petite retouche mineure a été de renommer les critères de façon à ce qu'ils aient le même nom que celui de la forme. Cela ne change rien au code, mais il est largement préférable que les deux aient le même nom.

Ensuite, une fonction **readCSV()** a été imaginée, avec un mécanisme de *try with resources* pour s'habituer à une lecture propre.

L'idée sera ensuite de recréer les attributs de la personne un par un pour ensuite appeler le constructeur de Person avec ces attributs.

#### 2. Export d'un résultat en CSV

L'objectif ici est d'exporter les meilleurs échanges. Nous pensons y mettre : Le pays hôte, le pays visiteur, les dates et quelques informations sur les personnes, pour finir par le score.

#### 3. Gestion de l'historique par sérialisation binaire

Après avoir été introduit à la sérialisation en cours, nous devrions être en mesure



de pouvoir sérialiser un objet `List<Exchange>` de tous les échanges passés (soit tous ceux considérés comme les meilleurs avec une date de fin antérieure à aujourd'hui). Le reconstruire permettra de l'appliquer à un objet `List<Exchange>` `history` de type `static` accessible partout. Dans la classe `Exchange`, nous comparerons l'effectif de l'échange actuel (`this`) à ceux passés.

#### 4. Implémentation d'un jeu de données

Tout au long du développement, nous testerons des jeux de données d'une structure identique à celle donnée. Ils seront décrits et appliqués lors de tests !

### 3.3 UML

Les classes, qu'elles soient nouvelles ou modifiées, sont présentées ci-dessous.

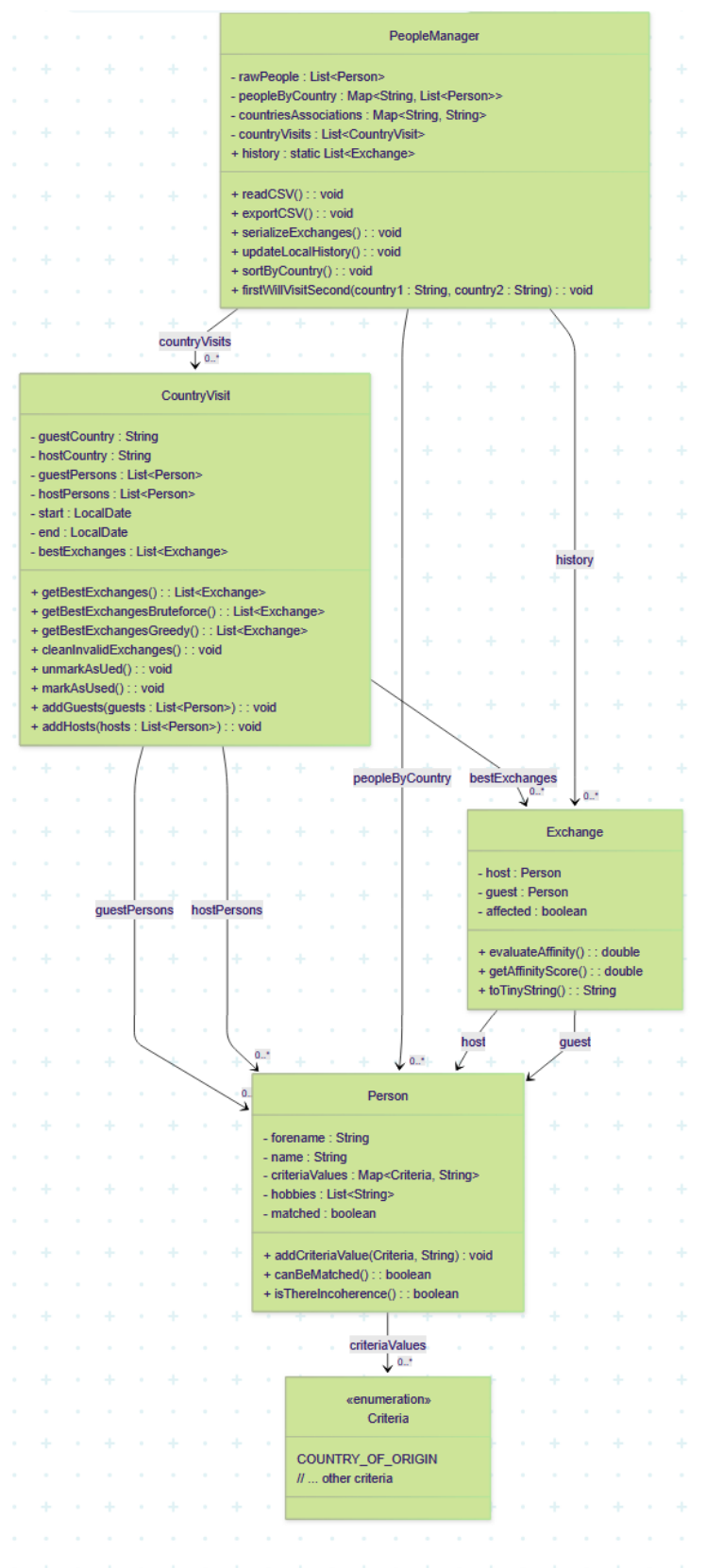


FIGURE 3 – Relations UML

Attaquons-nous maintenant à la description de tous ces changements.

**Observation:** Des commentaires ont été retirés du code affiché en extraits afin de le raccourcir. Les fichiers de code contiendront alors plus d'informations qu'ici et sont disponibles pour éclaircir tout manque de clarté.

**Observation:** L'ordre de présentation est également important et nous essayons de présenter les éléments avant qu'ils soient utilisés ailleurs.

### 3.4 Nouveau dossier : res

Ce dossier, nommé par convention, contiendra nos fichiers CSV, contenant les informations nécessaires à la construction de futurs objets Person.

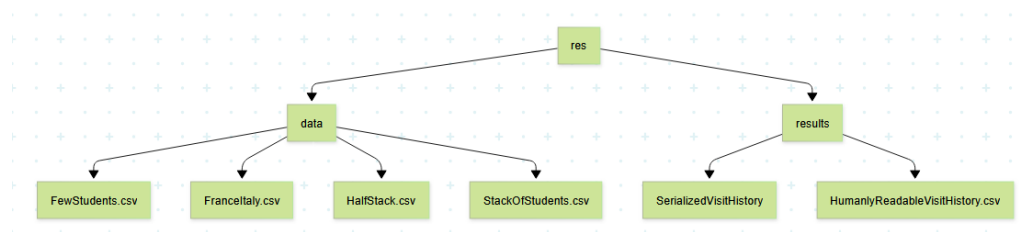


FIGURE 4 – Arborescence des éléments

- data, contenant les données prises en entrée
- results, contenant les données en sortie (exportation CSV, sérialisation)

Les fichiers CSV contiennent tous le même en-tête, tandis que le fichier sérialisé ne contient que du binaire, ne nécessitant donc aucune extension.

```

res > data > StackOfStudents.csv > data
1 FORENAME,NAME,COUNTRY_OF_ORIGIN,BIRTH_DATE,GUEST_ANIMAL_ALLERGY,HOST_HAS_ANIMAL,GUEST_FOOD_CONSTRAINT,HOST_FOOD,HOBBIES,GENDER,PAIR_GENDER,HISTORY
2 Sarah,CHANG,GERMANY,2008-03-10,no,no,nonuts,gluten-free,reading;technology;gaming;hiking,female,female,
3 Andrew,SHEPPARD,SWEDEN,2008-06-19,yes,no,,vegetarian;gluten-free,chess;sports;fashion,male,,same
4 Jennifer,BOWERS,HUNGARY,2007-09-10,no,no,nonuts,vegetarian,film;robotics;anime;painting,female,,
5 Tracy,HOWARD,DENMARK,2007-05-01,no,yes,,halal;vegetarian;halal,chess;crafts,female,female,
6 Jonathan,CAMPBELL,JAPAN,2006-10-21,no,yes,nonuts,vegetarian;halal;vegetarian,chess;reading;theatre,male,female,other
7 Lisa,KING,ITALY,2008-08-31,no,no,halal;vegan,,science;travel,female,female,other
8 Victoria,PATRICK,ITALY,2007-01-10,yes,yes,,,robotics;chess,female,female,other
9 Emily,BLATR,JAPAN,2009-04-23,no,yes,,vegetarian;halal;lactose,culture;crafts;books;anime,female,,other
10 Erin,CASTRO,EGYPT,2007-11-27,yes,no,gluten-free;lactose,music;drawing,female,male,same
11 Jeffrey,CARR,DENMARK,2007-04-09,no,yes,lactose,gaming;robotics,male,,
12 Vincent,WOODS,ITALY,2008-08-13,yes,yes,halal;nonuts,vegetarian;lactose,fashion;sports,male,,
13 David,TRUJILLO,ROMANIA,2008-09-25,yes,no,,vegetarian;halal,cooking;theatre;sports;culture,male,male,
14 Thomas,DAVIS,TURKEY,2008-02-05,no,no,,,crafts;travel;theatre;sports,male,male,other
15 Michelle,GLOVER,SWEDEN,2009-01-22,no,no,vegan;gluten-free,halal,science;dance;technology;culture,female,male,
16 April,SNYDER,ROMANIA,2007-10-26,yes,no,lactose;nonuts,nonuts;gluten-free,anime;culture;gaming,female,,other
17 David,NGUYEN,ITALY,2006-09-01,no,yes,nonuts;vegetarian,,fashion;theatre;hiking,male,female,other
18 Cheryl,BRADLEY,POLAND,2007-06-01,no,no,lactose;shellfish,vegetarian,reading;robotics;dance;science,female,,
19 Robert,THOMAS,SWITZERLAND,2008-09-16,no,no,,vegetarian;halal,animals;anime;robotics,male,,
20 Jerome,MILLER,ITALY,2007-10-06,no,yes,shellfish,vegetarian;halal,books;dance,male,same
21 Sheri,BOLTON,IRELAND,2008-06-25,yes,yes,vegan,lactose,music;reading;robotics,female,male,
22 Brian,ARNOLD,SPAIN,2006-12-06,yes,no,vegan,,films;painting;photography,male,male,
23 Kimberly,OLSEN,SPAIN,2008-09-12,no,yes,gluten-free,music;travel;reading,female,female,other
24 Christina,ILOVO,EGYPT,2008-09-23,no,yes,shellfish;gluten-free,shellfish;lactose,chess;science;reading,female,,same
25 Laurie,MALLACE,POLAND,2008-12-08,yes,no,gluten-free,vegetarian,animals;theatre;travel,female,male,
26 Rebecca,STEWART,POLAND,2007-06-07,no,no,vegan,vegetarian,drawing;art;travel,female,male,same
27 Dylan,LOPEZ,ITALY,2007-03-13,no,no,nonuts,vegetarian,chess;books,male,,same
28 Amanda,ZAVALA,SPAIN,2007-07-04,no,yes,shellfish;halal,,dance;drawing;anime,female,female,
29 John,DAVIS,IRAN,2006-12-15,yes,yes,,,art;travel;films;music,male,,
30 Sherri,MCLEAN,JAPAN,2006-09-28,no,no,halal,vegetarian,art;culture;manga,female,,
31 Stephanie,LEBLANC,MOROCCO,2008-09-12,yes,yes,nonuts,halal,books;films,female,male,same
32 Michael,MASSEY,GERMANY,2008-08-18,yes,yes,,,films;fashion;chess,male,,
33 Donald,CATIN,BRAZIL,2007-04-26,yes,no,lactose;nonuts,gluten-free,books;painting;anime,male,,
34 Steven,LAURENCE,EGYPT,2008-03-08,no,no,vegan;nonuts,gaming;films;manga;hiking,male,male,other
    
```

FIGURE 5 – Aperçu du contenu d'un fichier data

### 3.5 Nouvelle classe : CountryVisit

Il est temps que les visites entre pays aient maintenant leur propre classe ! Cette classe a pour objectif d'inclure un pays hôte et un pays visiteur, avec une liste de personnes des pays mentionnés. Ensuite, il conviendra de déterminer les meilleurs échanges en appelant la gestion du score d'affinité dans la classe Exchange.

#### 1. Les attributs

Java

```

1 private String guestCountry;
2 private String hostCountry;
3
4 private List<Person> guestPersons = new ArrayList<>();
5 private List<Person> hostPersons = new ArrayList<>();
6
7 // Start and end dates of the exchange
8 private LocalDate start ;
9 private LocalDate end ;
10
11 private List<Exchange> bestExchanges;
```

On considère que les constructeurs sont bien spécifiés, avec des getters | setters l'étant de même.

#### 2. Fonctions d'interaction

Afin de modifier les données d'un objet CountryVisit, certaines méthodes sont utiles :

Java

```

1 // Adding guests
2 public void addGuests(List<Person> guests) {
3     for (Person p : guests) {
4         if (p.getCriteriaValue(Criteria.COUNTRY_OF_ORIGIN) .
5             equalsIgnoreCase(this.guestCountry)) {
6             this.guestPersons.add(p);
7         }
8     }
9 }
10 // Adding hosts
11 public void addHosts(List<Person> hosts) {
12     for (Person p : hosts) {
13         if (p.getCriteriaValue(Criteria.COUNTRY_OF_ORIGIN) .
14             equalsIgnoreCase(this.hostCountry)) {
15             this.hostPersons.add(p);
16         }
17     }
18 }
```

### 3. Comment déterminer les meilleurs échanges ?

Il s'agit ici de déterminer la meilleure combinaison d'échanges au sein d'un effectif donné. Ici, l'effectif comprend les hôtes et les visiteurs. La finalité est d'avoir la combinaison avec la somme des scores la plus faible.

Le nombre de combinaisons est, habituellement, pour  $N$  la taille de la matrice carrée,  $N$  factorielle ( $N!$ ). De ce fait, la complexité peut très vite devenir insoutenable pour le programme.

Pour ce faire, le problème se résout couramment avec l'algorithme hongrois. Néanmoins, il s'agit d'un algorithme assez long à écrire et à mettre en place. De nombreuses bibliothèques Java comme JGraphT implémentent des algorithmes très optimisés capables de résoudre notre problème. Dans le cadre de ce sujet, une solution bruteforce sera appliquée afin de faire au plus simple.

Cette partie du code a été la plus difficile à trouver, autant dans sa réflexion que dans son écriture. Par conséquent certaines fonctions sont réutilisées de ce que nous pouvons trouver sur le Web.

Le code est assez long, et le mettre dans son entièreté ici reviendrait à inonder la page. Il se trouve alors dans la classe CountryVisit, de façon bien commentée.

Java

```

1 public List<Exchange> getBestExchangesBruteforce() {
2     [...]
3     [code permettant de trouver "best", un objet List<Exchange> qui
4         contient la somme minimale du cout des affectations]
5     %
6     this.bestExchanges = best;
7
8     if (this.bestExchanges == null) {
9         this.bestExchanges = new ArrayList<>();
10    }
11
12    // Cleaning too high affinity scores
13    this.cleanInvalidExchanges();
14
15    // Remaining exchanges persons are marked as matched and are
16    // not available for any other exchange yet.
17    this.markAsUsed();
18    return this.bestExchanges;
19 }
```

### 4. La méthode glouton

Dans les cas où l'effectif est trop grand, l'algorithme précédent, fonctionnant par bruteforce, se révèle grandement inefficace. Bien que les scénarios d'évaluation de ce programme portent difficilement sur des effectifs mettant en difficulté le programme, nous avons tout de même écrit une méthode glouton cherchant pour chaque étudiant la meilleure affinité directe avec le prochain. Cette méthode est très rapide en terme d'exécution, mais elle ne trouve pas toujours la meilleure combinaison d'échanges. Elle reste utile et elle sera appelée dans le cas de trop grands effectifs par le mécanisme suivant :

Java

```

1 // Re-searching the best exchanges every time as the persons inside may
  // have changed elsewhere
2 public List<Exchange> getBestExchanges() {
3     // Everyone is free again as we're searching another time
4     this.unmarkAsUsed();
5     int n = Math.min(this.hostPersons.size(), guestPersons.size());
6     if (n > 10) {
7         System.out.println("Warning: Bruteforce method is not
          recommended for more than 10 hosts/guests due to
          performance issues.");
8         System.out.println("Therefore, the greedy method will
          be used instead. It does not guarantee the best
          matching, but is much faster.");
9         return this.getBestExchangesGreedy(); // Fallback to
          greedy method for large groups
10    } else {
11        return this.getBestExchangesBruteforce();
12    }
13 }

```

### Exemple de conflit Bruteforce | Glouton

```

Greedy Total Affinity: 50.08
[2025_POL17->ITA7:9.35, 2025_POL35->ITA16:10.2, 2025_POL24->ITA6:14.03, 2025_POL25->ITA43:16.5]
Bruteforce Total Affinity: 48.41
[2025_POL24->ITA6:14.03, 2025_POL17->ITA7:9.35, 2025_POL25->ITA16:11.0, 2025_POL35->ITA43:14.03]

```

FIGURE 6 – Différents résultats trouvés

## 5. Enlever les mauvaises affinités

Une mauvaise affinité est caractérisée par un score d'affinité trop élevé (dans notre algorithme, environ supérieur à 100, et 200 pour une très mauvaise affinité). Néanmoins, cela n'empêche pas de se retrouver avec des échanges incompatibles | horribles dans la meilleure combinaison d'échanges, faute de mieux. Ne souhaitant pas se retrouver avec des personnes ne s'entendant pas du tout, il convient de les enlever.

Java

```

1 // Some scores may have been selected but they are way too high.
2 // It is better to remove them as their affinity is horrible. They will
   never get along with each other
3 // Better not get them together in any world
4 public void cleanInvalidExchanges() {
5     if (this.bestExchanges == null) return;
6     List<Exchange> exchangesToRemove = new ArrayList<>();
7     for (Exchange e : this.bestExchanges) {
8         if (e.evaluateAffinity() > CountryVisit.
           AFFINITY_CLEANING_THRESHOLD) {
9             exchangesToRemove.add(e);
10        }
11    }
12    this.bestExchanges.removeAll(exchangesToRemove);
13 }

```

Même situation avec et sans nettoyage

```

CountryVisit from poland to italy
Start date: 2025-06-01, End date: 2025-06-08
The best exchanges are:
2025_POL24->ITA6: 14.03
2025_POL25->ITA16: 11.0
2025_POL17->ITA19: 8516.15
2025_POL35->ITA43: 14.03

CountryVisit from germany to italy
Start date: 2025-06-01, End date: 2025-06-08
The best exchanges are:
2025_GER1->ITA37: 10020.0

CountryVisit from morocco to italy
Start date: 2025-06-01, End date: 2025-06-08
The best exchanges are:
2025_MOR42->ITA26: 15.3

```

FIGURE 7 – Résultat sans nettoyage

```

CountryVisit from poland to italy
Start date: 2025-06-01, End date: 2025-06-08
The best exchanges are:
2025_POL24->ITA6: 14.03
2025_POL25->ITA16: 11.0
2025_POL35->ITA43: 14.03

CountryVisit from germany to italy
Start date: 2025-06-01, End date: 2025-06-08
The best exchanges are:
No exchanges found (yet?). Likely because of no people left available, or the only exchanges had horrible affinity.

CountryVisit from morocco to italy
Start date: 2025-06-01, End date: 2025-06-08
The best exchanges are:
2025_MOR42->ITA26: 15.3
  
```

FIGURE 8 – Résultat avec nettoyage (seuil 200.0)

## 6. Rendre les personnes non disponibles pour de futurs échanges

Lorsque les meilleurs échanges sont trouvés, il convient de rendre les personnes y figurant comme indisponibles, dans le sens où elles ne sont disponibles que pour un échange à la fois.

Java

```

1 // Marks the people in the best exchange as used
2 // They won't be available for any other exchange.
3 public void markAsUsed() {
4     try {
5         for (Exchange e : this.bestExchanges) {
6             e.getHost().setMatched(true);
7             e.getGuest().setMatched(true);
8             e.setAffected(true);
9         }
10    } catch (NullPointerException e) {
11        System.out.println("Error while setting matched persons
12                           : " + e.getMessage());
13    }
14 }
  
```

C'en est fini pour les fonctionnalités utiles de cette classe. Cette classe a été particulièrement détaillée en raison de son importance vitale dans ce programme.

### 3.6 Nouvelle classe : PeopleManager

Cette classe sera relativement abstraite dans le sens où elle a vocation à être unique. Elle sert principalement d'interaction entre la classe Main et le reste du programme. Comme son nom l'indique, cette classe va gérer les flux de personnes, pour les attribuer à des pays, qui seront attribués à des CountryVisit, qui seront attribués à des Exchange... pour finir par les sauvegarder par sérialisation et exportation CSV.

Il faut voir un objet PeopleManager comme un individu omniscient aux capacités de gestion hors-normes et disposant de nombreux contacts.



Cette classe comprend beaucoup de fonctions. Elles peuvent paraître obscures, mais un scénario ainsi que son résultat seront donnés à la fin de cette sous-section afin de décrire leurs liens.

## 1. Les attributs

### Java

```

1 // Everyone from the CSV file
2 protected List<Person> rawPeople = new ArrayList<Person>();
3
4 // A country featuring its people
5 protected Map<String, List<Person>> peopleByCountry = new HashMap<
    String, List<Person>>();
6
7 // A country visiting another (first is the guest, second is the host)
8 protected Map<String, String> countriesAssociations = new HashMap<
    String, String>();
9
10 // Exchanges between people, with their affinity scores
11 protected List<CountryVisit> countryVisits = new ArrayList<CountryVisit>
    >();
12
13 // History of the past exchanges (Static)
14 public static List<Exchange> history = new ArrayList<Exchange>();
15
16 // Some file names / file paths
17 // Final paths that may not be modified
18
19 // Resources path
20 final static String MY_PATH = "res/";
21
22 // Two sub-directories
23 final static String MY_DATA_PATH = MY_PATH + "data/";
24 final static String MY_RESULT_PATH = MY_PATH + "results/";
25
26 // Serialized exports
27 final static String SERIALIZED_VISIT_HISTORY_FILE = "
    SerializedVisitHistory";
28 final static String SERIALIZED_VISIT_HISTORY_PATH = MY_RESULT_PATH +
    SERIALIZED_VISIT_HISTORY_FILE;
29
30 // Readable exports (won't be used in the program further)
31 final static String HUMANLY_READABLE_VISIT_HISTORY_FILE = "
    HumanlyReadableVisitHistory.csv";
32 final static String HUMANLY_READABLE_VISIT_HISTORY_PATH =
    MY_RESULT_PATH + HUMANLY_READABLE_VISIT_HISTORY_FILE;
33
34 // Changeable paths (Person database)
35 String myPersonFile ; // Called in constructor
36 String myPeoplePath ;

```

## 2. Lecture du CSV

La fonction suivante a été réécrite plusieurs fois afin de gérer la totalité des cas auquel le programme peut être confronté. L'utilisation d'un Scanner dans le try rend le programme plus clair. Si l'on détecte une incohérence dans les déclarations de la Person, on ne l'ajoutera pas au jeu effectif de données. Autrement, elle est ajoutée à une liste locale pour pouvoir ensuite être traitée selon les opérations demandées au programme !

Java

```

1 public void readCSV() {
2     try (Scanner scanner = new Scanner(new File(this.myPeoplePath))
3         ) {
4         if (!scanner.hasNextLine()) return; // fichier vide
5         String headerLine = scanner.nextLine();
6         String[] headers = headerLine.split(",");
7         while (scanner.hasNextLine()) {
8             String line = scanner.nextLine();
9             String[] values = line.split(",", -1);
10            if (values.length != headers.length) continue;
11
12            String forename = "";
13            String name = "";
14            HashMap<Criteria, String> criteriaValues = new
15                HashMap<>();
16            ArrayList<String> hobbies = new ArrayList<>();
17
18            for (int i = 0; i < headers.length; i++) {
19                String col = headers[i].trim().
20                    toUpperCase();
21                String val = values[i].trim();
22                if (val.isEmpty()) continue;
23
24                if (col.equals("FORENAME")) forename =
25                    val;
26                else if (col.equals("NAME")) name = val
27                    ;
28                else if (col.equals("HOBBIES")) {
29                    String[] hobbyArr = val.split("
30                        ");
31                    for (String h : hobbyArr) {
32                        if (!h.trim().isEmpty())
33                            hobbies.add(h.
34                                trim());
35                    }
36                }
37            }
38        }
39    }

```

Java

```

1         else {
2
3             try {
4                 Criteria crit =
5                     Criteria.
6                         valueOf(col
7                             );
8                 criteriaValues.
9                     put(crit ,
10                        val);
11             } catch (
12                 IllegalArgumentException
13                     e) {
14                 // Ignore
15                     unknown
16                     columns
17             }
18         }
19     }
20     if (!forename.isEmpty() && !name.
21         isEmpty()) {
22         // Creating the person
23         Person p = new Person(forename,
24             name, criteriaValues ,
25             hobbies);
26
27         // Checking incoherence in the
28             Person criterias (check
29             function doc)
30         // If there is one, we won't
31             add this person to the
32             practical dataset
33         if (p.isThereIncoherence()) {
34             System.out.println("
35                 Sorry, but " + p.
36                 tinyToString() + "
37                 had an animal while
38                 being allergic to
39                 them... We can't
40                 trust this person."
41             );
42         }
43         else {
44             rawPeople.add(p);
45         }
46     }
47 }
48 } catch (FileNotFoundException e) {
49     System.out.println("File not found : " + e.
50         getMessage());
51 }
52 }

```

### 3. Gestion des CountryVisit

La classe a des relations évidentes avec l'objet précédemment énoncé, CountryVisit. Des attributs de PeopleManager y font alors référence et ils sont exploités dans ces fonctions.

**Java**

```

1  // Sorting the people by their country using a map
2  public void sortByCountry() {
3      for (Person p : this.rawPeople) {
4          String country = p.getCriteriaValue(Criteria.
              COUNTRY_OF_ORIGIN);
5          if (country == null || country.isEmpty()) continue; //
              Skip if no country
6
7          List<Person> countryList = peopleByCountry.getDefault
              (country, new ArrayList<>());
8          countryList.add(p);
9          peopleByCountry.put(country, countryList);
10     }
11 }
12
13 // A country visiting another
14 public void firstWillVisitSecond(String country1, String country2) {
15     try {
16         this.countriesAssociations.put(country1, country2);
17     } catch (NullPointerException e) {
18         System.out.println("One of the countries is associated
              to a null value : " + e.getMessage());
19     } catch (Exception e) {
20         System.out.println("Error while associating the
              countries : " + e.getMessage());
21     }
22 }
23
24 // Creating country visits based on the associations
25 // All the Person from the CSV File will be added to the country visit
    following the guest / host nature of their country in the visit
26 public void createVisits() {
27     for (Map.Entry<String, String> entry : this.
        countriesAssociations.entrySet()) {
28         String guestCountry = entry.getKey();
29         String hostCountry = entry.getValue();
30
31         List<Person> guestPersons = this.peopleByCountry.get(
            guestCountry);
32         List<Person> hostPersons = this.peopleByCountry.get(
            hostCountry);
33
34         if (guestPersons != null && hostPersons != null) {
35             CountryVisit visit = new CountryVisit(
                guestCountry, hostCountry);
36             visit.addGuests(guestPersons);
37             visit.addHosts(hostPersons);
38             this.countryVisits.add(visit);
39         }
40     }
41 }

```

#### 4. La sortie

### Exportation CSV (lisible par l'humain)

Ici, nous exportons les meilleurs échanges au format CSV pour le rendre lisible par l'humain, contrairement au format binaire. Ce fichier n'a pas vocation à être utilisé par le programme. Il doit juste permettre un partage des données en étant lisible par l'humain.

**Java**

```

1 // CSV Export
2 // This method exports the best exchanges for each country visit to a
  human-readable CSV file.
3 // The CSV includes host/guest countries, dates, IDs, and affinity
  scores for each exchange.
4 // It is useful for sharing results or analyzing them outside the
  program.
5 public void exportCSV() {
6     try (FileWriter fw = new FileWriter(
7         HUMANLY_READABLE_VISIT_HISTORY_PATH)) {
8         // Write the CSV header
9         fw.write("HOST_COUNTRY,GUEST_COUNTRY,START_DATE,
10             END_DATE,HOST_ID,GUEST_ID,AFFINITY_SCORE\r\n");
11         // Iterate over all country visits
12         for (CountryVisit visit : this.countryVisits) {
13             String hostCountry = visit.getHostCountry();
14             String guestCountry = visit.getGuestCountry();
15             String startDate = "";
16             if (visit.getStart() != null) startDate = visit
17                 .getStart().toString(); // Format start
18                 date if present
19             String endDate = "";
20             if (visit.getEnd() != null) endDate = visit.
21                 getEnd().toString(); // Format end date if
22                 present
23             // For each best exchange in the visit, write a
24             line in the CSV
25             for (Exchange e : visit.getBestExchanges()) {
26                 int hostId = e.getHost().getID();
27                 int guestId = e.getGuest().getID();
28                 double score = e.evaluateAffinity(); //
29                 Calculate the affinity score for
30                 this exchange
31                 fw.write(hostCountry + "," +
32                     guestCountry + "," + startDate + ",
33                     " + endDate + "," +
34                     hostId + "," + guestId + "," + String.
35                     format("%.2f", score) + "\r\n");
36             }
37         }
38     }
39     // Handle file not found error (should not happen if path is
40     correct)
41     catch (FileNotFoundException e) {
42         System.out.println("File not found: " + e.getMessage());
43         ;
44         e.printStackTrace();
45     }
46     // Handle IO errors (disk full, permission denied, etc.)
47     catch (IOException e) {
48         System.out.println("Access error: " + e.getMessage());
49         e.printStackTrace();
50     }
51     // Handle any other unexpected exception
52     catch (Exception e) {
53         System.out.println("Writing error: " + e.getMessage());
54         e.printStackTrace();
55     }
56 }

```

## Sérialisation binaire

La sérialisation est un processus d'exportation de données très rapides car nous sauvegardons un objet au format binaire.

En l'occurrence, il s'agit de constituer en historique, revenant alors à sauvegarder la liste des échanges passés, revenant alors à sauvegarder la liste des meilleurs échanges (les considérant comme acquis).

Nous allons donc créer un objet `List<Exchange>` de tous les meilleurs échanges créés par un `PeopleManager`. `PeopleManager` dispose également d'un objet "local" `List<Exchange>` `history`, destiné à contenir l'objet désérialisé après. L'objet `history` est en mode public statique afin d'être accessible partout. Ce n'est pas un secret et tout le monde mérite de pouvoir connaître l'historique.



## Java

```

1 // SERIALIZATION
2
3 // We will save exchanges labelled as the best exchanges possible
4 // As these best exchanges are done, they are part of history
5 // Adding everything on one only stream to do things properly
6 /**
7  * Serializes all best exchanges from all country visits into a single
8  * file.
9  * This allows for easy loading of exchange history in future runs.
10 */
11 public void serializeExchanges() {
12     List<Exchange> allExchanges = new ArrayList<>();
13     for (CountryVisit visit : this.countryVisits) {
14         for (Exchange e : visit.getBestExchanges()) {
15             System.out.println(e);
16             allExchanges.add(e);
17         }
18     }
19     try (ObjectOutputStream oos = new ObjectOutputStream(new
20         FileOutputStream(PeopleManager.
21             SERIALIZED_VISIT_HISTORY_PATH))) {
22         oos.writeObject(allExchanges);
23     } catch (Exception e) {
24         System.out.println(e.getStackTrace());
25     }
26 }
27
28 // Associating history
29 /**
30  * Loads the exchange history from the serialized file and updates the
31  * static history list.
32  * This is useful for affinity calculations that depend on past
33  * exchanges.
34 */
35 public void updateLocalHistory() {
36     try (ObjectInputStream ois = new ObjectInputStream(new
37         FileInputStream(new File(PeopleManager.
38             SERIALIZED_VISIT_HISTORY_PATH)))) {
39         // Read the list of exchanges
40         // History is now the list of exchanges read from the
41         // file
42         PeopleManager.history = (List<Exchange>) ois.readObject();
43     } catch (Exception e) {
44         e.printStackTrace();
45     }
46 }

```

## Désérialisation

Afin de remplir l'objet history de PeopleManager, le fichier binaire contenant l'objet précédemment sérialisé sera lu.

## Java

```

1 // Associating history
2 /**
3  * Loads the exchange history from the serialized file and updates the
4  * static history list.
5  * This is useful for affinity calculations that depend on past
6  * exchanges.
7  */
8 public void updateLocalHistory() {
9     try (ObjectInputStream ois = new ObjectInputStream(new
10         FileInputStream(new File(PeopleManager.
11             SERIALIZED_VISIT_HISTORY_PATH)))) {
12         // Read the list of exchanges
13         // History is now the list of exchanges read from the
14         // file
15         PeopleManager.history = (List<Exchange>) ois.readObject
16             ();
17     } catch (Exception e) {
18         e.printStackTrace();
19     }
20 }

```

## 5. Scénario d'utilisation typique

Afin que les fonctions précédemment énoncées soient plus claires, il vaut mieux donner le scénario dans lequel elles seront amenées à être utilisées !

Java

```

1 // Main example of usage
2 public class Main {
3     public static void main(String[] args) {
4         // Classe anonyme concrète pour instancier
           PersonHandler
5         PeopleManager handler = new PeopleManager("HalfStack")
           {};
6         handler.readCSV();
7         //PeopleManager.displayPeople(handler.rawPeople);
8
9         handler.sortByCountry();
10        //System.out.println(handler.peopleByCountry);
11
12        handler.firstWillVisitSecond("poland", "italy");
13        handler.firstWillVisitSecond("germany", "italy");
14        handler.firstWillVisitSecond("morocco", "italy");
15        handler.createVisits();
16        for (CountryVisit visit : handler.countryVisits) {
17            System.out.println(visit);
18            // (La meilleure combinaison d'échanges est
               appelée dans le toString)
19        }
20        handler.serializeExchanges();
21        handler.exportCSV();
22
23        handler.updateLocalHistory();
24        System.out.println(PeopleManager.history);
25    }
26 }

```

```

Sorry, but Victoria PATRICK (ID: 7) had an animal while being allergic to them... We can't trust this person.
Sorry, but Vincent WOODS (ID: 11) had an animal while being allergic to them... We can't trust this person.
Sorry, but Sheri BOLTON (ID: 20) had an animal while being allergic to them... We can't trust this person.
Sorry, but John DAVIS (ID: 28) had an animal while being allergic to them... We can't trust this person.
Sorry, but Stephanie LEBLANC (ID: 30) had an animal while being allergic to them... We can't trust this person.
Sorry, but Michael MASSEY (ID: 31) had an animal while being allergic to them... We can't trust this person.
Sorry, but Lauren HAYES (ID: 34) had an animal while being allergic to them... We can't trust this person.
Sorry, but David DENNIS (ID: 38) had an animal while being allergic to them... We can't trust this person.
Sorry, but Barbara DAVIS (ID: 45) had an animal while being allergic to them... We can't trust this person.
Sorry, but Andrew SMITH (ID: 46) had an animal while being allergic to them... We can't trust this person.
CountryVisit from poland to italy
Start date: 2025-06-01, End date: 2025-06-08
The best exchanges are:
2025_POL24->ITA6:14.03: 14.03
2025_POL25->ITA16:11.0: 11.0
2025_POL35->ITA43:14.03: 14.03

CountryVisit from germany to italy
Start date: 2025-06-01, End date: 2025-06-08
The best exchanges are:
No exchanges found (yet?). Likely because of no people left available, or the only exchanges had horrible affinity.

CountryVisit from morocco to italy
Start date: 2025-06-01, End date: 2025-06-08
The best exchanges are:
2025_MOR42->ITA26:15.3: 15.3

2025_POL24->ITA6:14.03
2025_POL25->ITA16:11.0
2025_POL35->ITA43:14.03
2025_MOR42->ITA26:15.3
[2025_POL24->ITA6:14.03, 2025_POL25->ITA16:11.0, 2025_POL35->ITA43:14.03, 2025_MOR42->ITA26:15.3]

```

FIGURE 9 – Résultat du scénario d'utilisation

## 3.7 Modifications de Person

Person a été ici adapté aux nouveaux besoins !

### 1. Les attributs ajoutés

Il s'agit ici d'un ID unique, et de deux attributs booléens utiles pour la suite.

isAvailable indique si quelqu'un est disponible pour un échange. S'il est blessé par exemple, il sera mis à false. isMatched indique si quelqu'un figure dans une combinaison des meilleurs échanges. S'il y figure, il sera mis à true.

Java

```

1 // ATTRIBUTES
2 private static int cpt = 1; // Unique ID counter for each person
3
4 private final int ID = cpt++; // Unique and final ID for each person
5
6 private boolean isAvailable = true ;
7
8 // Matched in the lowest weight graph
9 // A matched person is still available , but is not available for new
  exchanges
10 private boolean isMatched = false ;

```

### 2. Des constructeurs plus précis

Par soucis du détail, il convient d'écrire plusieurs constructeurs gérant plusieurs cas de figure. La vérification des critères est désormais plus adaptée en cas d'appel futur dans la lecture du CSV (PeopleManager).

## Java

```

1 // Chained constructors to allow flexibility in instantiation
2
3 // Case where no parameters are given
4 public Person() {
5     this("Unknown", "Unknown", new HashMap<Criteria, String>(), new
        ArrayList<String>());
6 }
7
8 public Person(String firstName, String lastName) {
9     this(firstName, lastName, new HashMap<Criteria, String>(), new
        ArrayList<String>());
10 }
11
12 public Person(String firstName, String lastName, HashMap<Criteria,
    String> criteriaValues) {
13     this(firstName, lastName, criteriaValues, new ArrayList<String>
        >());
14 }
15
16 public Person(String firstName, String lastName, ArrayList<String>
    hobbies) {
17     this(firstName, lastName, new HashMap<Criteria, String>(),
        hobbies);
18 }
19
20 public Person(String firstName, String lastName, HashMap<Criteria,
    String> criteriaValues, ArrayList<String> hobbies) {
21     this.firstName = firstName;
22     this.lastName = lastName;
23     this.criteriaValues = new HashMap<Criteria, String>();
24     for (Criteria crit : criteriaValues.keySet()) {
25         String value = criteriaValues.get(crit);
26         this.addCriteriaValue(crit, value);
27         // La méthode addCriteriaValue gère déjà la validation
28         // Il vaut mieux un ajout individuel pour chaque critère
            plutôt qu'un qui bloque tout.
29     }
30     this.hobbies = hobbies;
31     this.meetingSpecificCountryRules();
32 }

```

### 3. Adaptation accrue en vue du CSV

Dans le CSV donné en modèle, des types booléens ont comme donnée yes | no. Il s'agit en réalité de true | false, alors il convient de les convertir afin de s'adapter au mieux à la structure donnée dans la méthode addCriteriaValue.

**Java**

```

1 // Associating a criteria with a value in the criteriaValues map.
2 public void addCriterionValue(Criteria criteria, String value) {
3     value = value.toLowerCase();
4     // Transforming "yes" and "no" to "true" and "false"
5     // for boolean criterias
6     if (value.equals("yes")) value = "true" ;
7     else if (value.equals("no")) value = "false" ;
8
9     if (Criteria.isCriteriaTypeValid(criteria, value) && Criteria.
        isCriteriaValueValid(criteria, value)) {
10         this.criteriaValues.put(criteria, value);
11     }
12     if (criteria == Criteria.COUNTRY_OF_ORIGIN) {
13         this.meetingSpecificCountryRules(); // Verifying the
            specific country rules
14     }
15 }

```

#### 4. Le reste

D'autres méthodes ont été ajoutées, mais elles sont déjà assez implicites dans leur nom pour ne pas devoir les mentionner, s'agissant principalement de getters | setters, ou de toString améliorés. Ces méthodes seront utilisées par la suite et leur fonctionnement est celui attendu de par leur nom.

### 3.8 Modifications de Criteria

L'objectif ici est de pouvoir spécifier si un critère est rédhibitoire ou non.

Un "ratio" est également ajouté, afin de déterminer l'importance attribuée au critère. Ce ratio est appelé lors de la modification du score en conséquence du traitement du critère. Il sera utilisé dans la partie IHM principalement (avec une jauge), donc il ne devrait pas être utilisé ici.

#### 1. Les attributs ajoutés

**Java**

```

1 // Ratio used to measure the impact given to the criteria when changing
    the score
2 private double ratio ;
3
4 // Mandatoriness of the criteria (insane score added)
5 public boolean isMandatory = false;

```

#### Application de ces attributs

## Java

```

1 GUEST_ANIMAL_ALLERGY('B', true),
2 HOST_HAS_ANIMAL('B'),
3 HOST_FOOD('T'),
4 GUEST_FOOD_CONSTRAINT('T', true),
5 PAIR_GENDER('T'),
6 GENDER('T'),
7 BIRTH_DATE('D'),
8 COUNTRY_OF_ORIGIN('T'),
9 HISTORY('T'),
10 NEED_ONE_HOBBY('B', true);

```

## 2. Constructeurs ajustés en conséquence

## Java

```

1 // CONSTRUCTORS
2 Criteria(char type) {
3     this(type, 1.0);
4 }
5
6 Criteria(char type, double ratio) {
7     this(type, ratio, false);
8 }
9
10 Criteria(char type, boolean isMandatory) {
11     this(type, 1.0, isMandatory);
12 }
13
14 Criteria(char type, double ratio, boolean isMandatory) {
15     this.type = type;
16     this.ratio = ratio;
17     this.isMandatory = isMandatory;
18 }

```

## 3.9 Modifications de Exchange

La classe Exchange a autant été améliorée qu'adaptée aux besoins de ce rendu. L'objectif de cette classe sera de donner un score d'affinité entre un hôte et un visiteur selon leurs caractéristiques. L'algorithme jugeant de l'importance de ces critères a déjà été discuté et appliqué dans la partie Graphes de cette SAÉ. Une version Python avait été écrite et de ce fait, il sera facile de l'adapter en Java.

### 1. Les attributs ajoutés

## Java

```
1 private double affinityScore = DEFAULT_AFFINITY_SCORE;
2 private static final double DEFAULT_AFFINITY_SCORE = 10.0;
3
4 public boolean affected = false ;
```

## 2. Suppression de isCompatible

La méthode isCompatible n'est aujourd'hui plus utile car elle ne répond plus à nos besoins. Elle est supprimée. Sa logique est remplacée dans la fonction qui suit.

## 3. L'attribution d'un score d'affinité

Cette méthode va, selon le retour de différents Handler (des gestionnaires) traitant de Criteria, modifier le score afin d'en donner un jugement de la compatibilité de l'hôte et du visiteur.



## Java

```

1  public double evaluateAffinity() {
2      // Resetting affinity score
3      this.affinityScore = DEFAULT_AFFINITY_SCORE;
4      // Following handlers treat "boolean" preferences
5      // If it is true, a criteria is not fulfilled, following in a
        score penalty
6
7      // Food differences
8      if (foodHandler()) changeScore(10, Criteria.
        GUEST_FOOD_CONSTRAINT);
9
10     // Animal differences
11     if (animalHandler()) changeScore(10, Criteria.
        GUEST_ANIMAL_ALLERGY);
12
13     // Do they need at least one common hobby?
14     if (oneHobbyHandler()) changeScore(10, Criteria.NEED_ONE_HOBBY)
        ;
15
16     // History check, true means they have been together before
17
18     String history = historyHandler();
19     // Lowering the score by a lot if the two wanted the same
20     if (history.equals("sameX2")) changeScore(0.1, Criteria.HISTORY
        );
21
22     // Lowering the score by a bit if one of the two wanted the
        same and the other doesn't mind
23     if (history.equals("sameX1")) changeScore(0.7, Criteria.HISTORY
        );
24
25     // Raising it by a ton if one of the two wanted another person
26     // Criteria has two mandatory modes so we have to do the
        mandatoriness in the parameters
27     if (history.equals("other")) changeScore(5010, Criteria.HISTORY
        );
28
29     // Age difference
30     this.affinityScore += (int) Math.round(this.birthDateHandler()
        * 10);
31
32     // Gender compatibility
33     if (hostGenderHandler()) changeScore(1.5, Criteria.PAIR_GENDER)
        ;
34     if (guestGenderHandler()) changeScore(1.5, Criteria.GENDER);
35
36     // Lowering the score based on the number of common hobbies on
        a progressive scale
37     // The more common hobbies, the lower the score
38     this.affinityScore *= Math.pow(0.85, this.countCommonHobbies())
        ;
39
40     this.affinityScore = Math.round(this.affinityScore * 100.0) /
        100.0;
41     return this.affinityScore;
42 }

```

## 4. Les gestionnaires de critères (Handler)

### Les booléens

Java

```

1  public boolean oneHobbyHandler() {
2      String hostRule = this.host.getCriteriaValue(Criteria.
        NEED_ONE_HOBBY);
3      String guestRule = this.guest.getCriteriaValue(Criteria.
        NEED_ONE_HOBBY);
4
5      if ("true".equals(hostRule) || "true".equals(guestRule)) {
6          return this.countCommonHobbies() < 1;
7      }
8      return false;
9  }
10
11 public boolean animalHandler() {
12     String hostHasAnimal = this.host.getCriteriaValue(Criteria.
        HOST_HAS_ANIMAL);
13     String guestAllergy = this.guest.getCriteriaValue(Criteria.
        GUEST_ANIMAL_ALLERGY);
14
15     if ("true".equals(guestAllergy) && "true".equals(hostHasAnimal)
        ) {
16         return true;
17     }
18     return false;
19 }
20
21 public boolean hostGenderHandler() {
22     String hostPrefGender = this.host.getCriteriaValue(Criteria.
        PAIR_GENDER);
23     String guestGender = this.guest.getCriteriaValue(Criteria.
        GENDER);
24     try {
25         if (!hostPrefGender.equals(guestGender)) {
26             return true;
27         }
28     } catch (NullPointerException e) {
29         // If one of the criteria is not defined, we consider
        it compatible
30         return false;
31     }
32
33     return false;
34 }

```

## Java

```

1  public boolean guestGenderHandler() {
2      String guestPrefGender = this.guest.getCriteriaValue(Criteria.
        PAIR_GENDER);
3      String hostGender = this.host.getCriteriaValue(Criteria.GENDER)
        ;
4      try {
5          if (!guestPrefGender.equals(hostGender)) {
6              return true;
7          }
8      } catch (NullPointerException e) {
9          // If one of the criteria is not defined, we consider
            it compatible
10         return false;
11     }
12     return false;
13 }
14
15 public boolean foodHandler() {
16     String hostFood = this.host.getCriteriaValue(Criteria.HOST_FOOD
        );
17     String guestFood = this.guest.getCriteriaValue(Criteria.
        GUEST_FOOD_CONSTRAINT);
18     try {
19         if (!hostFood.equals(guestFood)) {
20             return true;
21         }
22     } catch (NullPointerException e) {
23         // If one of the criteria is not defined, we consider
            it compatible
24         return false;
25     }
26     return false;
27 }

```

## L'écart d'âge

Cette méthode n'est qu'une traduction du code réalisé en Python lors de la partie graphes.

**Java**

```

1  /**
2   * Calculates a score based on the age difference between host and guest
3   * Returns a value to be added to the affinity score.
4   */
5  public double birthDateHandler() {
6      String hostBirth = this.host.getCriteriaValue(Criteria.
          BIRTH_DATE);
7      String guestBirth = this.guest.getCriteriaValue(Criteria.
          BIRTH_DATE);
8      if (hostBirth != null && guestBirth != null && !hostBirth.
          isEmpty() && !guestBirth.isEmpty()) {
9          DateTimeFormatter formatter = DateTimeFormatter.
              ofPattern("yyyy-MM-dd");
10         LocalDate today = LocalDate.now();
11         LocalDate hostDate = LocalDate.parse(hostBirth,
              formatter);
12         LocalDate guestDate = LocalDate.parse(guestBirth,
              formatter);
13
14         int hostAge = Period.between(hostDate, today).getYears
              ();
15         int guestAge = Period.between(guestDate, today).
              getYears();
16         double ageMoyen = (hostAge + guestAge) / 2.0;
17
18         int diffMois = Math.abs((hostDate.getYear() - guestDate
              .getYear()) * 12 + (hostDate.getMonthValue() -
              guestDate.getMonthValue()));
19
20         double scoreAge = 0;
21         if (diffMois <= 18) {
22             scoreAge += (diffMois / 12.0) * Math.pow(0.9,
                ageMoyen);
23             scoreAge *= 0.9;
24         } else {
25             scoreAge += (diffMois / 12.0) * Math.pow(0.9,
                ageMoyen);
26             scoreAge *= 1.5;
27         }
28         return scoreAge;
29         // arbitrary weighting
30     }
31     return 0.0;
32 }

```

## Le nombre de hobbies

La méthode countCommonHobbies reste inchangée.

## 5. Le gestionnaire d'historique

Parcourant une liste désérialisée d'échanges passés, nous pouvons déterminer si les personnes figurant dans l'échange courant ont déjà eu un échange par le passé afin de traiter leurs préférences vis-à-vis de l'historique.

## Java

```

1 public String pastPreferences() {
2     String guestPref = this.getGuest().getCriteriaValue(Criteria.
        HISTORY);
3     String hostPref = this.getHost().getCriteriaValue(Criteria.
        HISTORY);
4
5     // If null, set to empty string
6     if (guestPref==null) guestPref="";
7     if (hostPref==null) hostPref="";
8
9     // Both want the same
10    if (guestPref.equals("same") && hostPref.equals("same")) return
        "sameX2";
11    // If one prefers other, it will be other for the two
12    if (guestPref.equals("other") || hostPref.equals("other"))
        return "other";
13    // One of the two would like the same, the other doesn't mind
14    if (guestPref.equals("same") || hostPref.equals("same")) return
        "sameX1";
15
16    return "";
17 }
18
19 // Returning a string qualifying the history handling (empty, same,
    other)
20 public String historyHandler() {
21     for (Exchange e : PeopleManager.history) {
22         if ((this.hasSamePersons(e)) && (this.pastPreferences()
            .equals("sameX1"))) {
23             return "sameX1";
24         }
25         if ((this.hasSamePersons(e)) && (this.pastPreferences()
            .equals("sameX2"))) {
26             return "sameX2";
27         }
28         if ((this.hasSamePersons(e)) && (this.pastPreferences()
            .equals("other"))) {
29             return "other";
30         }
31     } // They have never been together before, then it doesn't
        matter, it's skipped anyway
32     return "";
33 }
34
35 // Checking if the same people are featured in the other exchange, in
    the same roles or reverted
36 public boolean hasSamePersons(Exchange other) {
37     if (other == null) return false;
38     // Same host/guest or inverted host/guest
39     boolean direct = this.getHost().getID() == other.getHost().
        getID() &&
40     this.getGuest().getID() == other.getGuest().getID();
41     boolean inverse = this.getHost().getID() == other.getGuest().
        getID() &&
42     this.getGuest().getID() == other.getHost().getID();
43     return direct || inverse;
44 }

```

## 6. Changer le score selon le critère

Une méthode changeant le score est utile après avoir vérifié le retour des Handler, comme le montre le code.

En cas de critère qualifié de contrainte rédhibitoire non respecté, le score est augmenté de façon à ce qu'il soit toujours strictement supérieur à n'importe quel échange n'ayant pas de contrainte rédhibitoire non respectée, avec une affinité quelconque.

Lorsqu'un int est en paramètre, la méthode augmentera le score de cette valeur.

Lorsqu'un double est en paramètre, la méthode multipliera le score par cette valeur.

Cette différenciation est utile car lorsque l'on baisse le score, nous préférons le baisser par une multiplication afin d'être plus précis selon différentes situations. Par exemple, beaucoup de baisses utilisent un facteur de 0.5.

Java

```

1 // Change the affinity score of the exchange by adding a value (int)
  following the ratio of the criteria.
2 public void changeScore(int value, Criteria criteria) {
3     // No Integer.MAX_VALUE to avoid problems, not needed
4     if (criteria.isMandatory) value+=9999;
5     this.affinityScore += value * criteria.getRatio() ; // Default
      ratio is 1.0
6 }
7
8 // Multiply the affinity score by a factor (double)
9 public void changeScore(double value, Criteria criteria) {
10     if (criteria.isMandatory) value+=9999;
11     this.affinityScore *= (value * criteria.getRatio());
12 }

```

C'en est tout pour Exchange. Les modifications ont été assez importantes sur cette classe, mais la logique reste la même qu'en Graphes.

## 3.10 Classes de Test

Les tests seront maintenant plus fournis afin que peu importe les futures modifications, les situations présentées soient toujours résolues.

Pour chaque partie, un test en particulier, jugé comme le plus intéressant, sera donné en extrait. Un commentaire l'accompagnera. Néanmoins, le code en contient encore plus !

### 1. Person

Ce test est assez simple. Person n'a pas vraiment à être testé car elle ne dépend pas d'autres classes, mais plutôt l'inverse. Person est principalement composé de fonctions simples de type getter | setter. Les tests restent couplés aux anciens, en enlevant les toString car l'ordre est aléatoire et n'a pas d'importance.

## Java

```

1      public class PersonTest {
2
3          private Person alice;
4
5          @BeforeEach
6          public void initTest () {
7              alice = new Person("Alice", "Smith");
8          }
9
10         @Test
11         public void testAvailabilityAndMatching () {
12             // By default, a person is available and not
13             matched
14             assertTrue(alice.isAvailable());
15             assertFalse(alice.isMatched());
16             assertTrue(alice.canBeMatched());
17
18             // Set as matched, should not be available for
19             new matches
20             alice.setMatched(true);
21             assertTrue(alice.isMatched());
22             assertFalse(alice.canBeMatched());
23
24             // Set as unavailable, should not be available
25             for new matches
26             alice.setMatched(false);
27             alice.setAvailability(false);
28             assertFalse(alice.isAvailable());
29             assertFalse(alice.canBeMatched());
30
31             // Set as available again, should be available
32             for matching
33             alice.setAvailability(true);
34             assertTrue(alice.canBeMatched());
35         }
36     }

```

## 2. Exchange

Ici, nous testons le rapport des gens avec un historique simulé et l'influence sur leurs critères. L'ancien ExchangeTest est désormais obsolète, car nous avons supprimé la méthode isCompatible, sur laquelle ces tests s'appuyaient. Son adaptation est néanmoins implémentée dans ces tests.



## Java

```

1  public class ExchangeTest {
2
3      private Person alice;
4      private Person bob;
5      private Exchange e1;
6
7      @BeforeEach
8      public void initTest () {
9          alice = new Person("Astana", "Nur-sultan");
10         bob = new Person("Bob", "Marley");
11         e1 = new Exchange(alice, bob);
12     }
13
14     ...
15     @Test
16     public void testHistoryHandlerWithPastExchanges () {
17         // Preparing
18         PeopleManager.history.clear();
19
20         // Case "other"
21         alice.addCriteriaValue(Criteria.HISTORY, "other");
22         bob.addCriteriaValue(Criteria.HISTORY, "");
23         Exchange past = new Exchange(alice, bob);
24         PeopleManager.history.add(past);
25         assertTrue(e1.historyHandler().equals("other"));
26
27         // Case "sameX2"
28         PeopleManager.history.clear();
29         alice.addCriteriaValue(Criteria.HISTORY, "same");
30         bob.addCriteriaValue(Criteria.HISTORY, "same");
31         Exchange past2 = new Exchange(alice, bob);
32         PeopleManager.history.add(past2);
33         assertTrue(e1.historyHandler().equals("sameX2"));
34
35         // Case "sameX1"
36         PeopleManager.history.clear();
37         alice.addCriteriaValue(Criteria.HISTORY, "same");
38         bob.addCriteriaValue(Criteria.HISTORY, "");
39         Exchange past3 = new Exchange(alice, bob);
40         PeopleManager.history.add(past3);
41         assertTrue(e1.historyHandler().equals("sameX1"));
42
43         // Case ""
44         PeopleManager.history.clear();
45         alice.addCriteriaValue(Criteria.HISTORY, "");
46         bob.addCriteriaValue(Criteria.HISTORY, "");
47         Exchange past4 = new Exchange(alice, bob);
48         PeopleManager.history.add(past4);
49         assertTrue(e1.historyHandler().equals(""));
50     }

```

### 3. Criteria

Un seul test a été ajouté, car il n'y a eu qu'un seul nouveau critère.

Java

```

1  @Test
2  void testCriteriaValueValid() {
3      assertTrue(Criteria.isCriteriaValueValid(Criteria.
4          PREFERENCE_GENDER, "male"));
5      assertFalse(Criteria.isCriteriaValueValid(Criteria.
6          PREFERENCE_GENDER, "femme"));
7      assertTrue(Criteria.isCriteriaValueValid(Criteria.DATE_OF_BIRTH
8          , "2000-01-01"));
9      assertFalse(Criteria.isCriteriaValueValid(Criteria.
10         DATE_OF_BIRTH, "2020-01-01"));
11     assertFalse(Criteria.isCriteriaValueValid(Criteria.
12         DATE_OF_BIRTH, "je suis le 8 mai 1212 et je suis mal écrit "
13     ));
14 }

```

#### 4. CountryVisit

L'objectif sera ici de vérifier que les tailles obtenues à l'issue de situation d'échanges sont les bonnes. Si la logique vient à changer, les tailles doivent rester les mêmes.

## Java

```

1  public class CountryVisitTest {
2
3      private PeopleManager handler;
4
5      @BeforeEach
6      public void initTest() {
7          handler = new PeopleManager("StackOfStudents");
8          handler.readCSV();
9          handler.sortByCountry();
10     }
11
12     /* These tests are hard to write because :
13
14     Not all the persons from the CSV are added to the effective
15     database because many have incoherences
16     10 guests with 10 hosts doesn't always mean 10 exchanges
17     because some are not compatible
18
19     So these tests are more "predicted" and the main objective here
20     is to observe if the expected numbers is still the same,
21     which means the logic didn't change for whatever reason.
22     */
23
24     @Test
25     public void testRoomForThreePoorItalians() {
26         handler.firstWillVisitSecond("poland", "italy");
27         handler.firstWillVisitSecond("spain", "italy");
28
29         handler.createVisits();
30         CountryVisit visit1 = handler.getCountryVisit("poland",
31             "italy");
32         CountryVisit visit2 = handler.getCountryVisit("spain",
33             "italy");
34
35         assertEquals("poland", visit1.getGuestCountry());
36         assertEquals("italy", visit1.getHostCountry());
37
38         assertEquals("spain", visit2.getGuestCountry());
39         assertEquals("italy", visit2.getHostCountry());
40
41         // Checking same size as before
42         assertEquals(visit1.getBestExchanges().size(), 10);
43
44         // There are 3 italians left and 14 spaniards
45
46         // But think about it : these 3 italians left were not
47         compatible with the other poles
48         // It is very likely that their criterias are too
49         strong to allow any compatibility with others.
50
51         // This way, there is only one italian that is
52         compatible with the spaniards, or at least does not
53         have a horrible affinity.
54         assertEquals(visit2.getBestExchanges().size(), 1);
55     }
56 }

```

## 5. PeopleManager

Pour finir, ce test portera sur :

- Le succès de la sérialisation d'un échange type
- Le succès de sa désérialisation

Java

```

1 // Mostly about serialization tests
2 public class PeopleManagerTest {
3
4     private PeopleManager handler;
5
6     @BeforeEach
7     public void setup() {
8         // Use the provided CSV file for a realistic test
9         handler = new PeopleManager("StackOfStudents");
10        handler.readCSV();
11        handler.sortByCountry();
12    }
13
14    @Test
15    public void testSerializationAndHistoryUpdate() {
16        // Setup a visit and create exchanges
17        handler.firstWillVisitSecond("poland", "italy");
18        handler.createVisits();
19
20        // Serialize the best exchanges
21        handler.serializeExchanges();
22
23        // Check that the file was created
24        File serFile = new File(PeopleManager.
25            SERIALIZED_VISIT_HISTORY_PATH);
26        assertTrue(serFile.exists(), "Serialized file should
27            exist after serialization");
28
29        // Clear the static history and update it from the file
30        PeopleManager.history.clear();
31        assertEquals(0, PeopleManager.history.size());
32
33        handler.updateLocalHistory();
34
35        // After update, history should be filled with Exchange
36        // objects
37        assertFalse(PeopleManager.history.isEmpty(), "History
38            should be loaded from the serialized file");
39        for (Exchange e : PeopleManager.history) {
40            assertNotNull(e.getHost());
41            assertNotNull(e.getGuest());
42            assertTrue(e.getAffinityScore() > 0 || e.
43                getAffinityScore() == 0);
44        }
45    }
46 }

```

## 6. Bonus : AllTests

Ce petit morceau de code permet de tout lancer à la fois afin de détecter un changement pouvant être critique.

## Java

```

1 package current;
2
3 import org.junit.platform.suite.api.SelectClasses;
4 import org.junit.platform.suite.api.Suite;
5
6 // This class runs all test classes in the 'current' package
7 @Suite
8 @SelectClasses({
9     PersonTest.class,
10    CriteriaTest.class,
11    ExchangeTest.class,
12    CountryVisitTest.class,
13    PeopleManagerTest.class
14 })
15 public class AllTests {
16     // No code needed, the annotations handle everything
17 }

```

### 3.11 Conclusion et critiques de ce rendu

Notre programme répond aux demandes de ce rendu, mais il se peut qu'il ait des vulnérabilités. Les demandes à satisfaire sont faisables, mais mêlées entre elles, elles peuvent mener à des conflits entre les demandes. Une critique à faire porterait sûrement sur notre façon de trouver les échanges qui peut être assez bancal (notamment sur markAsUsed et unmarkAsUsed). Il est bon de critiquer son code et nous avons tâché de le faire en le remettant très souvent en question. Le développement pur n'est pas forcément notre qualité première, donc nous essayons de compenser cela par une conception théorique réfléchie au maximum.

## 4 Rendu 4

Ces deux semaines de réalisation seront assez maigres mais importantes : il s'agira ici de peaufiner les résultats et d'éliminer certaines incohérences remarquées.

### 4.1 Attentes

Au programme, nous avons :

- Prétraitement des critères des adolescents sur demande
- Prétraitement automatique des critères en cas d'existence d'un fichier de configuration à un emplacement prédéfini

Ces demandes paraissent assez floues, et pour y répondre au mieux, nous nous sommes renseignés.

Il s'agit alors de :

- *Prétraitement des critères des adolescents sur demande* : Lorsqu'un ajout d'une personne est refusé pour cause de critères d'entrée insatisfaits (incohérences), l'utilisateur doit avoir le choix de l'ajouter ou non à la liste effective.
- *Prétraitement automatique des critères en cas d'existence d'un fichier de configuration à un emplacement prédéfini* : Un fichier de configuration doit pouvoir donner les instructions à suivre (par exemple minAge=15, maxAge=20) et le programme doit valider automatiquement les entrées des personnes selon cette configuration.

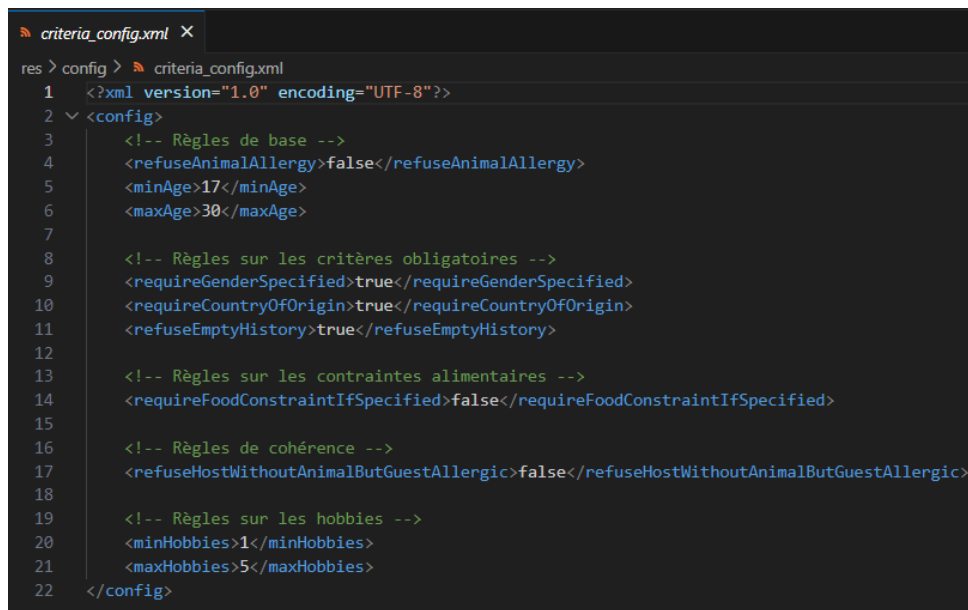
### 4.2 Réflexion

Le prétraitement sur demande n'est pas quelque chose de complexe à réaliser, puisqu'il s'agira sûrement de l'ensemble suivant :

- Si une personne est refusée, donner le texte : *Voulez-vous entrer cette personne ? Oui/Non*
- Lire cette entrée avec un Scanner positionné sur System.in
- Agir selon la réponse

Le prétraitement automatique, lui, nécessitera un peu plus d'efforts ! Il nécessite d'abord un fichier de configuration. Le format XML sera choisi car il est une des solutions les plus efficaces dans ce cas.

La forme de ce fichier peut d'ailleurs être réalisée dès maintenant :



```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <config>
3   <!-- Règles de base -->
4   <refuseAnimalAllergy>false</refuseAnimalAllergy>
5   <minAge>17</minAge>
6   <maxAge>30</maxAge>
7
8   <!-- Règles sur les critères obligatoires -->
9   <requireGenderSpecified>true</requireGenderSpecified>
10  <requireCountryOfOrigin>true</requireCountryOfOrigin>
11  <refuseEmptyHistory>true</refuseEmptyHistory>
12
13  <!-- Règles sur les contraintes alimentaires -->
14  <requireFoodConstraintIfSpecified>false</requireFoodConstraintIfSpecified>
15
16  <!-- Règles de cohérence -->
17  <refuseHostWithoutAnimalButGuestAllergic>false</refuseHostWithoutAnimalButGuestAllergic>
18
19  <!-- Règles sur les hobbies -->
20  <minHobbies>1</minHobbies>
21  <maxHobbies>5</maxHobbies>
22 </config>

```

FIGURE 10 – Forme du fichier de configuration XML

Un exemple vaut souvent mille mots mais en nécessite tout de même un peu :

- Ici, nous donnons une valeur booléenne ou un nombre au champ donné.
- La forme est très simple et est facilement modifiable, le XML étant un format plus ou moins universel et répandu.
- Toutes les règles des critères sont couvertes.
- Ce fichier aura donc vocation à être lu et interprété par une fonction spécifique.

Pour ce faire, nous utiliserons une nouvelle classe, utilisée dans notre gestionnaire PeopleManager, nommée CriteriaConfigValidator.

### 4.3 UML

Voici donc le nouvel UML, qui sera donc l'UML final !

Cet UML cible tout ce qui est important en excluant getters | setters | constructeurs ou fonctions implicites de par leur nommage.



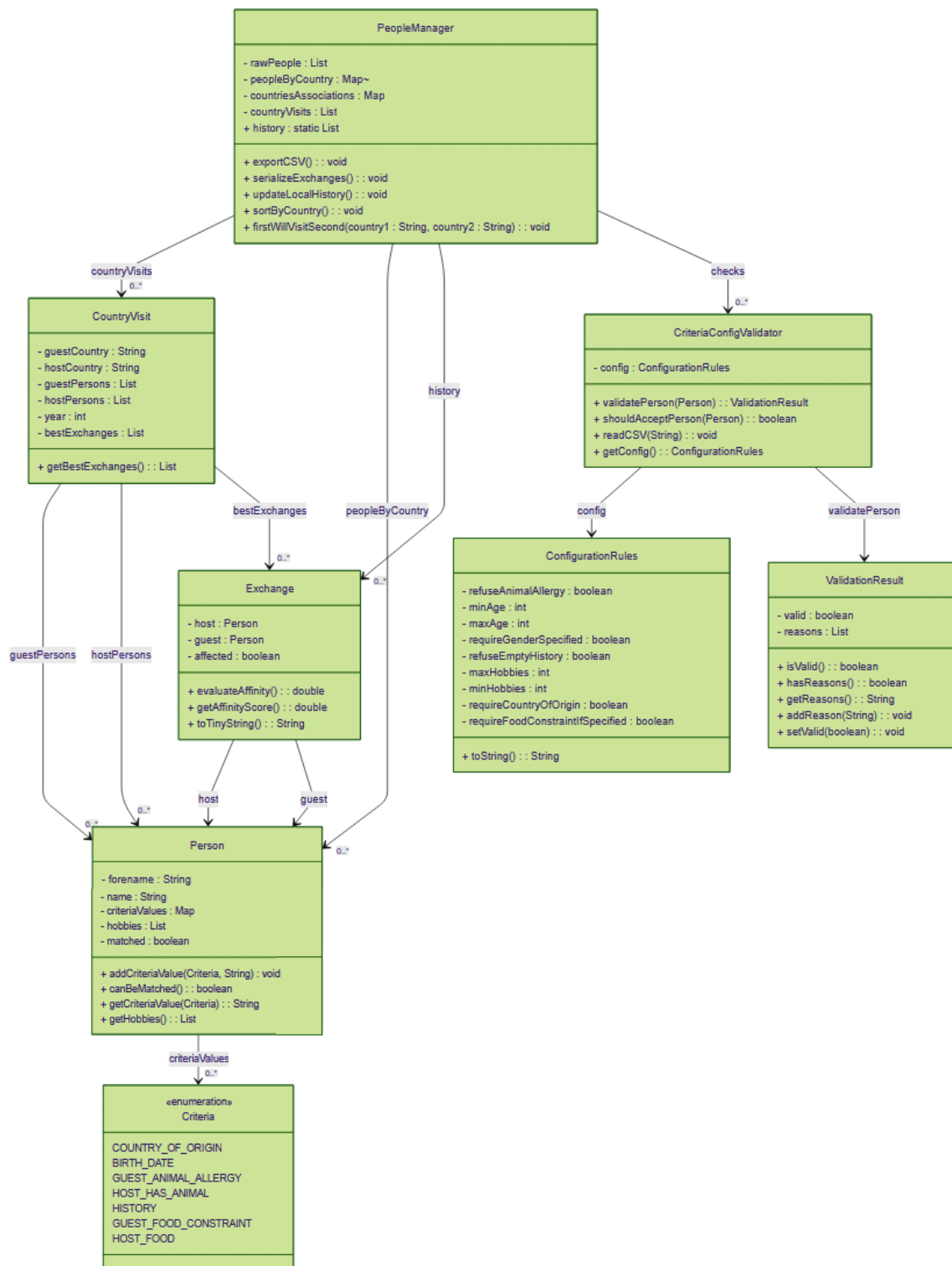


FIGURE 11 – Relations UML

## 4.4 Nouvelle réflexion

Il a été réfléchi que pour qu'un échange soit valide, il doit :

- Ne pas s'être produit auparavant lors de la même année (historique)
- Ne pas contenir des membres contenus dans des échanges produits lors de la même année au sein même de cette visite (unicité locale)
- Ne pas contenir des membres affectés dans d'autres échanges la même année (unicité universelle)
- Ne pas contenir des membres n'étant plus disponibles

## 4.5 Modifications de PeopleManager

Le principal angle d'attaque a été l'historique. Après avoir réfléchi, il n'est que logique qu'un échange déjà présent dans l'historique n'ait tout simplement pas lieu. Par conséquent, l'historique a été révisé, ainsi que la classe `Exchange`, pour que ce soit le cas.

Si bien que lorsqu'on lance le programme un nombre infini de fois, l'historique atteindra à un moment une taille maximale, tous les échanges possibles s'étant déroulés.

Les visites entre pays sont également uniques par année désormais.

Le code suivant permet alors de tester cette nouvelle fonctionnalité :

## Java

```

1 public static void main(String[] args) {
2     PeopleManager handler = new PeopleManager("HalfStack") ;
3
4     PeopleManager.alwaysCheckCSVInputs = false ;
5
6     handler.readCSV() ;
7     handler.sortByCountry() ;
8
9     handler.firstWillVisitSecond(2023, "germany", "egypt") ;
10    handler.firstWillVisitSecond(2023, "germany", "egypt") ;
11
12    handler.firstWillVisitSecond(2023, "germany", "iran") ;
13
14    handler.firstWillVisitSecond(2024, "germany", "egypt") ;
15
16    handler.firstWillVisitSecond(2025, "germany", "iran") ;
17
18    handler.createVisits() ;
19
20    int previousSize = -1 ;
21    int currentSize = PeopleManager.history.size() ;
22    while (currentSize != previousSize) {
23        previousSize = currentSize ;
24        for (CountryVisit c : handler.countryVisits) {
25            c.getBestExchanges() ;
26            handler.serializeExchanges() ;
27            //System.out.println("historique : " +
28                //    PeopleManager.history) ;
29        }
30        currentSize = PeopleManager.history.size() ;
31    }
32
33    System.out.println("historique : " + PeopleManager.history);
34    // historique : [2023:GER1->IRA92:11.9, 2023:GER82->IRA94:10.0,
35        //    2024:GER89->EGY9:14.03, 2024:GER1->EGY41:7.95, 2024:GER87
36        //    ->EGY83:10.0, 2025:GER1->IRA92:11.9, 2025:GER82->IRA94:1.0]

```

## 1. Modifications relatives au traitement de l'historique

Afin que le code ci-dessus fonctionne, il a été nécessaire de modifier le traitement de l'historique. Le but est de constamment réalimenter l'historique à chaque nouvelle modification. Ce code a vocation à être utilisé comme dans l'exemple ci-dessus.

## Java

```

1 // SERIALIZATION || HISTORY MANAGEMENT
2
3 // We will save exchanges labeled as the best exchanges possible
4 // Once these best exchanges are made, they are added to history
5 // All in a single stream to ensure correct processing
6 /**
7  * Serializes all best exchanges from all country visits into one file.
8  * This allows easy loading of exchange history in future runs.
9  */
10 public void serializeExchanges() {
11     //List<Exchange> allExchanges = new ArrayList<>();
12     List<Exchange> allExchanges = history ;
13     for (CountryVisit visit : this.countryVisits) {
14         for (Exchange e : visit.getExchanges()) {
15             //System.out.println(e);
16
17             // Do not add duplicates
18             if (PeopleManager.isInHistory(e)) continue ;
19
20             // Otherwise, add
21             allExchanges.add(e);
22         }
23     }
24     try (ObjectOutputStream oos = new ObjectOutputStream(new
25         FileOutputStream(PeopleManager.
26             SERIALIZED_VISIT_HISTORY_PATH))) {
27         //System.out.println(allExchanges);
28         oos.writeObject(allExchanges);
29     } catch (Exception e) {
30         System.out.println(e.getStackTrace());
31     }
32     this.updateLocalHistory();
33 }

```

De plus, la fonction static suivante vérifie au sein de l'historique la disponibilité de la personne par l'année.

## Java

```

1 // Check if a person was matched in history this year
2 // (history contains only exchanges, so only matched persons)
3 // Static so it can be used in CountryVisit
4 public static boolean isMatchedThisYearInHistory(Person p, int year) {
5     for (Exchange e : PeopleManager.history) {
6         if (e.getGuest().equals(p) || e.getHost().equals(p)) {
7             if (e.getYear() == year) {
8                 return true ;
9             }
10        }
11    }
12    return false ;
13 }
14
15 // Supplémentaire :
16
17 // Check if the exchange exists in history
18 // Static so it can be used in CountryVisit
19 public static boolean isInHistory(Exchange e1) {
20     for (Exchange e2 : PeopleManager.history) {
21         if (e1.equals(e2)) return true ;
22     }
23     return false ;
24 }

```

## 2. Modifications relatives à la configuration automatique

La fonction readCSV() a été adaptée d'une façon liée à la nouvelle classe Criteria-ConfigValidator. Elle appelle une vérification au sein de cette classe et va, si échec de la vérification, donner le choix à l'utilisateur d'accepter ou non cette personne malgré l'échec de la vérification (en lui en donnant les raisons).

## Java

```

1  ...
2  if (!forename.isEmpty() && !name.isEmpty()) {
3      // Creating the person
4      Person p = new Person(forename, name, criteriaValues, hobbies);
5      // === NEW STEP: AUTOMATIC XML VALIDATION ===
6      boolean passedAutoValidation = configValidator.
7          shouldAcceptPerson(p);
8      // configValidator prints reasons of the rejection if so
9
10     if (!passedAutoValidation) {
11         // The person was rejected by automatic validation
12         continue; // Skip to next person
13     } else {
14         System.out.println("\n===== \n");
15         System.out.println("Person: " + p.tinyToString() + "
16             automatically added.");
17         rawPeople.add(p);
18     }
19 }

```

Cette fonction s'appuie sur une nouvelle classe qui vaut donc la peine d'être introduite !

## 4.6 Nouvelle classe : CriteriaConfigValidator

Des extraits de codes seront donnés ici afin de ne pas surcharger, mais la classe est pleinement commentée. Cette classe implémente aussi la classe interne ConfigurationRules afin de décomposer le code.

### 1. Les attributs

## Java

```

1  // Default path for configuration file
2  private static final String DEFAULT_CONFIG_PATH = "res/config/
3      criteria_config.xml";
4
5  // Configuration loaded from XML
6  private ConfigurationRules config;

```

### 2. La classe interne ConfigurationRules

## Java

```

1  /**
2   * Inner class to store configuration rules
3   */
4   public static class ConfigurationRules {
5       public boolean refuseAnimalAllergy = false;
6       public int minAge = 0;
7       public int maxAge = 150;
8       public boolean requireGenderSpecified = false;
9       public boolean refuseEmptyHistory = false;
10      public boolean requireFoodConstraintIfSpecified = false;
11      public boolean refuseGuestAllergicIfHostHasAnimal = false;
12      public int maxHobbies = -1; // -1 = no limit
13      public int minHobbies = 0;
14      public boolean requireCountryOfOrigin = false;
15  }

```

### 3. Lecture de la configuration XML

La configuration est un fichier XML placé dans le dossier res / config / \*.xml. Elle sera lue ici et son contenu sera retranscrit au sein des attributs locaux (ConfigurationRules).

## Java

```

1  /**
2   * Loads configuration from XML file
3   */
4  private void loadConfiguration(String configPath) {
5      config = new ConfigurationRules();
6
7      File configFile = new File(configPath);
8      if (!configFile.exists()) {
9          System.out.println("Configuration file not found: " +
10             configPath);
11          System.out.println("Using default configuration.");
12          return;
13      }
14
15      try {
16          DocumentBuilderFactory factory = DocumentBuilderFactory
17              .newInstance();
18          DocumentBuilder builder = factory.newDocumentBuilder();
19          Document document = builder.parse(configFile);
20          document.getDocumentElement().normalize();
21
22          Element root = document.getDocumentElement();
23
24          // Reading different configuration criteria
25          config.refuseAnimalAllergy = getBooleanValue(root, "
26             refuseAnimalAllergy", false);
27          config.minAge = getIntValue(root, "minAge", 0);
28          config.maxAge = getIntValue(root, "maxAge", 150);
29          config.requireGenderSpecified = getBooleanValue(root, "
30             requireGenderSpecified", false);
31          config.refuseEmptyHistory = getBooleanValue(root, "
32             refuseEmptyHistory", false);
33          config.requireFoodConstraintIfSpecified =
34             getBooleanValue(root, "
35             requireFoodConstraintIfSpecified", false);
36          config.refuseGuestAllergicIfHostHasAnimal =
37             getBooleanValue(root, "
38             refuseGuestAllergicIfHostHasAnimal", false);
39          config.maxHobbies = getIntValue(root, "maxHobbies", -1)
40             ;
41          config.minHobbies = getIntValue(root, "minHobbies", 0);
42          config.requireCountryOfOrigin = getBooleanValue(root, "
43             requireCountryOfOrigin", false);
44
45          System.out.println("Configuration loaded successfully
46             from: " + configPath);
47          System.out.println("Applied rules: " + config.toString
48             ());
49
50      } catch (Exception e) {
51          System.err.println("Error loading XML configuration: "
52             + e.getMessage());
53          System.out.println("Using default configuration.");
54      }
55  }

```



## Java

```

1  /**
2   * Utility methods for reading values from XML
3   */
4  private boolean getBooleanValue(Element parent, String tagName, boolean
    defaultValue) {
5      NodeList nodeList = parent.getElementsByTagName(tagName);
6      if (nodeList.getLength() > 0) {
7          String value = nodeList.item(0).getTextContent().trim()
            .toLowerCase();
8          return value.equals("true") || value.equals("1") ||
            value.equals("yes");
9      }
10     return defaultValue;
11 }
12
13 private int getIntValue(Element parent, String tagName, int
    defaultValue) {
14     NodeList nodeList = parent.getElementsByTagName(tagName);
15     if (nodeList.getLength() > 0) {
16         try {
17             return Integer.parseInt(nodeList.item(0).
                getTextContent().trim());
18         } catch (NumberFormatException e) {
19             System.err.println("Invalid value for " +
                tagName + ": " + nodeList.item(0).
                getTextContent());
20         }
21     }
22     return defaultValue;
23 }

```

#### 4. L'acceptation d'une personne ou non

Cette méthode déterminera si la personne en paramètre passe la vérification selon les critères chargés lors du fichier de configuration. Si elle ne les passe pas, la liste des raisons sera donnée.

## Java

```

1  /**
2   * Integrated method to use in readCSV() for automatic validation
3   */
4   public boolean shouldAcceptPerson(Person person) {
5       ValidationResult result = validatePerson(person);
6
7       if (!result.isValid()) {
8           System.out.println("== AUTOMATIC VALIDATION ==");
9           System.out.println("Person: " + person.tinyToString());
10          System.out.println("Result: REJECTED");
11          System.out.println("Reasons:");
12          System.out.println(result.getReasons());
13
14          // If manual verification is enabled, ask for
            confirmation
15          if (PeopleManager.alwaysCheckCSVInputs) {
16              System.out.print("Do you still want to add this
                person? (Yes/No): ");
17              String response = PeopleManager.
                SCANNER_SYSTEM_IN.nextLine().trim().
                toLowerCase();
18              boolean accepted = response.equals("oui") ||
                response.equals("o") || response.equals("
                yes") || response.equals("y");
19              System.out.println(accepted ? "Person added
                despite the rules." : "Person definitively
                rejected.");
20              System.out.println("
                =====\n");
21              return accepted;
22          } else {
23              System.out.println("Person automatically
                rejected.");
24              System.out.println("
                =====\n");
25              return false;
26          }
27      }
28
29      return true; // Accepted
30  }

```

## 5. Le reste

Le reste de la classe est assez implicite (nommage des fonctions) et ne vaut pas la peine d'être décrit.

## 4.7 Exemples de prétraitement

Les exemples suivants démontrent l'implémentation efficace de ce nouveau système. Une vérification manuelle est donc possible par dessus la vérification automatique, tout

comme il est possible de la désactiver (PeopleManager.alwaysCheckCSVInputs : boolean).

```
Configuration loaded successfully from: res/config/criteria_config.xml
Applied rules: ConfigurationRules{refuseAnimalAllergy=false, minAge=14, maxAge=20, requireGenderSpecified=true,
n=true}
=== START OF CSV PROCESSING WITH AUTO VALIDATION ===

=====

Person: Sarah CHANG (ID: 1) automatically added.

=====

Person: Andrew SHEPPARD (ID: 2) automatically added.

=====
```

FIGURE 12 – Début des vérifications de l'effectif

```
Person: Tracy HOWARD (ID: 4) automatically added.

=====

Person: Jonathan CAMPBELL (ID: 5) automatically added.

=====

Person: Lisa KING (ID: 6) automatically added.
=== AUTOMATIC VALIDATION ===
Person: Victoria PATRICK (ID: 7)
Result: REJECTED
Reasons:
- Person rejected: allergic to animals and host has an animal
Do you still want to add this person? (Yes/No): Yes
```

FIGURE 13 – Une personne qui ne passe pas automatiquement

```
Person: Victoria PATRICK (ID: 7)
Result: REJECTED
Reasons:
- Person rejected: allergic to animals and host has an animal
Do you still want to add this person? (Yes/No): Yes
Person added despite the rules.

=====

=====

Person: Victoria PATRICK (ID: 7) automatically added.

=====
```

FIGURE 14 – Acceptation manuelle de la personne

```

Person: Cassandra BENNETT (ID: 98) automatically added.
=====
Person: Alexa HERNANDEZ (ID: 99) automatically added.
=====
Person: George RICHARDSON (ID: 100) automatically added.
=== END OF CSV PROCESSING ===
Number of persons added: 61

```

FIGURE 15 – Fin des vérifications + effectif ajouté

## 4.8 Modifications de Person

La classe Person a également été modifiée afin d'inclure la notion de disponibilité par année.

Java

```

1 // Matched in the lowest weight graph
2 // A matched person is still available , but is not available for new
  exchanges
3 // Being matched depends of the year , because it only happens once a
  year
4 private Map<Integer , Boolean> isMatched = new HashMap<Integer , Boolean
  >() ;
5
6 public void setMatched(int year , boolean isMatched) {
7     this.isMatched.put(year , isMatched);
8 }
9
10 // Is this person matched?
11 public boolean isMatched(int year) {
12     Boolean matched = this.isMatched.get(year);
13     return matched != null && matched;
14 }
15
16 public boolean canBeMatched(int year) {
17     // A person can be matched if it is available and not already
      matched
18     return this.isAvailable && !this.isMatched(year);
19 }

```

## 4.9 Modifications de CountryVisit

La classe CountryVisit doit également implémenter ces nouvelles considérations, et cela passe par de plus amples vérifications au sein de l'algorithme de recherche d'appariement :

## Java

```

1  if (hostCombo.get(i).canBeMatched(this.getYear()) && permutedGuests.get
    (i).canBeMatched(this.getYear())) {
2      Exchange e = new Exchange(this.getYear(), hostCombo.get(i),
        permutedGuests.get(i));
3      // The current exchange being in history has no real sense :
4      // It just means the user has loaded it twice at least
5      // This way we'll ignore because there's only one country visit
        between two countries possible per year.
6      // It is hard to conceive this fact
7      //System.out.println(e);
8
9      if (
10         PeopleManager.isInHistory(e) ||
11         PeopleManager.isBanned(e) ||
12         // Checking if the person is already matched in the current
            year in the history
13         PeopleManager.isMatchedThisYearInHistory(e.getHost(), e.getYear
            ()) ||
14         PeopleManager.isMatchedThisYearInHistory(e.getGuest(), e.
            getYear())) {
15             // Globally, if the exchange or its containing persons
                are not valid, the pair is skipped
16             sum = Double.MAX_VALUE;
17             break;
18         } else {
19             current.add(e);
20         }
21         sum += e.getAffinityScore();
22
23     } else {
24         // If one of the persons cannot be matched, skip this pairing
25         sum = Double.MAX_VALUE;
26         break;
27     }

```

## 4.10 Classes de Test

Les classes de Test n'ont pas eu à être mises à jour car ce rendu s'appuyait principalement sur la gestion améliorée de l'historique (vérification manuelle possible) et sur la vérification automatico-manuelle des entrées de personnes.

Néanmoins, la classe de test CriteriaConfigValidatorTest existe et est consultable dans le code. Elle n'a rien de très complexe, donc il vaut mieux ne pas surcharger en la mettant ici.

## 4.11 Conclusion

C'était donc aujourd'hui le dernier rendu et le projet est donc terminé ! Il a été assez long et fastidieux, le plus long de l'année, mais nous avons tâché de séquencer au maximum la production dans le temps. Si bien que finalement, on ne se rend compte de certaines choses qu'en dehors du code. Une réflexion sur ce projet venait souvent dans les pensées

quotidiennes et pas forcément devant un IDE. En somme, le séquençage du projet a permis de façonner le programme selon les règles les plus affinées possibles.

Certains concepts nous ont été introduits sur le tas (sérialisation, scanner) et nous considérons les avoir assez bien maîtrisés. Des cas de test sont toujours présents et décrivent la manière d'utiliser le programme. De plus, ce programme est essentiellement une base à l'autre partie : Interaction Humain Machine, dans laquelle la partie *Poo* réalise les calculs en arrière plan de l'interface.

**Notre projet n'est donc pas parfait mais nous considérons qu'il est bien documenté, bien écrit, et qu'il répond pleinement aux demandes logiques formulées.**

Merci d'avoir consulté cette présentation