Valentin THUILLIER
Romain DEGEZ
Elise LEROY

16 juin 2023

SAé 2.01/2.02 : développement d'une application et exploration algorithmique

Rapport de développement

I/ Les diagrammes UML

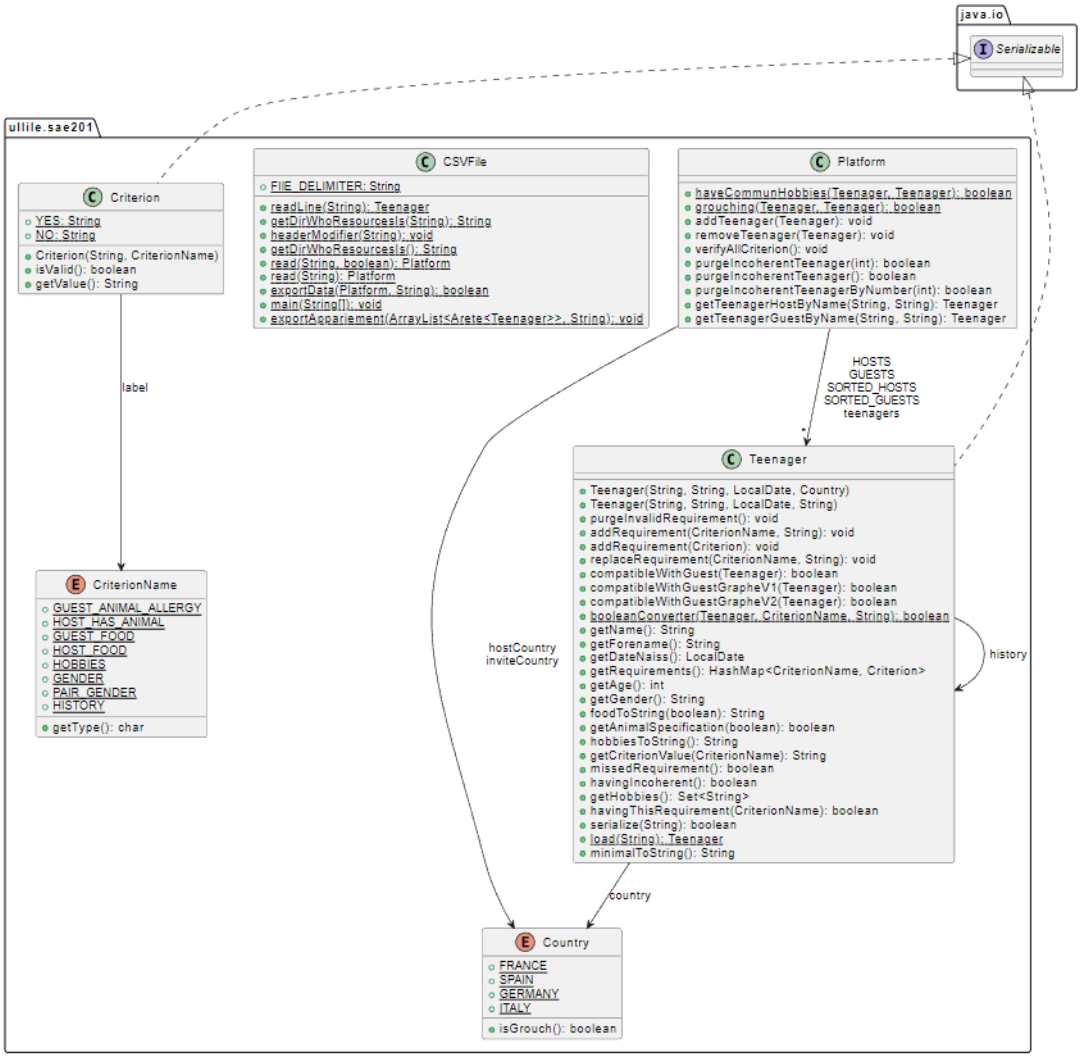

Figure 1 UML POO

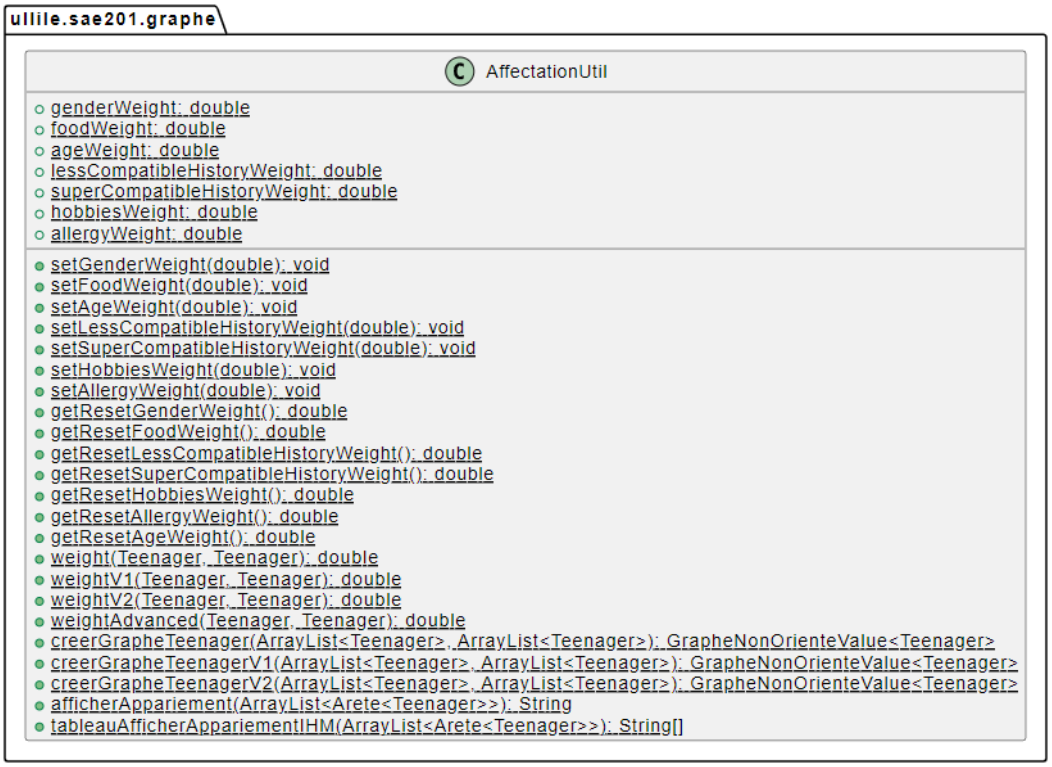

Figure 2 UML du package graphes

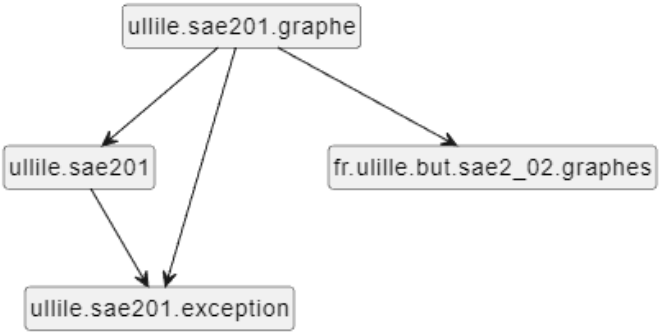

Figure 3 UML de la dépendance entre le package POO, exceptions et graphes

Seules les méthodes et attributs publics ont été générés.

Les commandes

Vous pouvez générer une javadoc à l'aide cette commande :

javadoc -sourcepath src -cp lib/sae2_02.jar -d resources/javadoc -docletpath lib/umldoclet-2.1.0.jar -doclet nl.talsmasoftware.umldoclet.UMLDoclet -encoding "UTF-8" ullile.sae201

Un .jar que nous avons ajouté dans la lib se chargera de vous générer la javadoc dans le dossier ressources.

Pour lancer l'interface de terminal, vous pouvez le lancer directement depuis votre IDE le fichier Main.java situé dans le dossier ihm-final. Si vous rencontrez cette erreur :

Valentin THUILLIER
Romain DEGEZ
Elise LEROY


```
Welcome to UniCo !
Made with love by
        - Romain DEGEZ
        - Elise LEROY
        - Valentin THUILLIER
Please enter the path to the CSV file:
Documents/adosAleatoires
Unrecognized request: { _request: evaluate }
```

Il faudra compiler et lancer le fichier « à la main » .

Dès le début du projet nous avons veillé à avoir un code décomposé au plus possible afin de nous permettre de réutiliser les méthodes. Nous avons aussi fait en sorte de laisser à l'utilisateur d'adapter le programme selon ses besoins.

II/ Commentaires sur les classes et l'UML

La classe Criterion

Nous avons pris la liberté de définir les critères POSSIBILITY_HISTORY, FOOD et GENDERS comme des HashSets pour que l'utilisateur puisse ajouter des valeurs.

```
private final static HashSet<String> GENDERS = new HashSet<>() {{
    add(e:"male");
    add(e:"female");
    add(e:"other");
}};
private final static HashSet<String> FOOD = new HashSet<>() {{
    add(e:"nonuts");
    add(e:"vegetarian");
    add(e:"");
}};
private final static HashSet<String> POSSIBILITY_HISTORY = new HashSet<>() {{
    add(e:"same");
    add(e:"other");
    add(e:"");
}};
```

La classe Teenager

Bien qu'il soit possible de faire hériter la classe pour le critère *grouching* nous avons favorisé l'usage d'un enum enrichi afin de rendre le code plus optimisé en évitant la redéfinissions des méthodes. Il était possible aussi de faire un abstract ou une interface mais nous avons estimé que cette solution était moins optimale que l'enum enrichi.

```java
public enum Country {
    FRANCE(isGrouch:true),
    SPAIN(isGrouch:false),
    GERMANY(isGrouch:false),
    ITALY(isGrouch:false);

    private final boolean ISGROUCH;

    /**
     * CriterionName constructor
     *
     * @param isGrouch (boolean) - Character state
     */
    Country(boolean isGrouch) {
        this.ISGROUCH = isGrouch;
    }

    /**
     * Get the character state
     *
     * @return (boolean) - Character state
     */
    public boolean isGrouch() {
        return ISGROUCH;
    }
```

Le constructeur de Teenager requiert un nom, prénom, une date de naissance et un pays. Ces attributs sont tous final car nous estimons qu'il y a peu de risque que ces critères changent. L'utilisateur pourra ensuite ajouter des critères ou non, le code est protégé contre d'éventuelles erreurs liées à un manque de critère.

```java
public class Teenager implements Serializable {

    private final String NAME, FORENAME;
    private final Country COUNTRY;
    private final LocalDate DATENAISS;
```

Dans la version rendue de notre code, seul le critère *gender* est obligatoire. Les critères obligatoires sont précisés dans l'attribut HashSet static et final REQUIERED. Nous avons choisi de les lister dans un Set afin qu'il soit plus facile pour l'utilisateur d'ajouter de nouveaux critères obligatoires. Par ailleurs, nous avons décidé de définir ces critères obligatoires comme static pour permettre une meilleure accessibilité étant donné que nous avions besoin d'y accéder de nombreuses fois dans notre code.

```java
private static final HashSet<CriterionName> REQUIDED = new HashSet<>() {{
    add(CriterionName.GENDER);
}};
```

Nous remarquons sur l'UML que seules les instances de Teenager et Criterion sont sérialisables car nous souhaitons de sauvegarder l'état mémoire de leurs instances dans un fichier.

La classe AffectationUtil

Les méthodes GrapheNonOrienteValue<Teenager> , Arête et d'autres classes spécifique aux graphes ont été encapsulées du .jar que nous a fourni Madame Boneva.

Il existe différentes méthodes creerGrapheTenager et wieght correspondant aux différents tags.

La gestion des exceptions

Afin d'éviter au plus possible que le programme s'arrête de façon impromptue, nous avons fait en sorte de gérer au maximum les exceptions.

Des exceptions particulières que nous avons définis dans le package ullile.sae201.exception peuvent être invocable par Citerion lorsqu'ils sont mal définis (l'utilisateur aurait donné une String au lieu d'un booléen, etc.) Et L'exception RequiermentNotFound sera soulevée par Teenagers lorsque ses critères sont mal définis. Par exemple, si l'utilisateur tente un getter sur ses hobbies alors qu'ils ne sont pas définis.

Nous avons fait remonter au plus possible les exceptions de la classe CSVFile telle qu'IllegalArgumentException, InvalidCriterion et NoSuchElementException. Pour l'import de CSV, nous avons veillé à ce que *Country* puisse être de type String (les pays sont normalement des enum) et à ce que le header n'ai pas d'organisation particulière.

Les tests

Nous avons à la fois effectué des tests unitaires et d'intégrations.

La répartition du travail

Pour cette saé nous avons décidé de principalement nous répartir les tâches par matière. Ainsi Valentin était en charge de la partie programmation orientée objet, Romain du graphe et Elise de l'IHM. Nous avons décidé de nous organiser de la sorte car nous avions remarqué que chacun d'entre nous avait une affinité avec les matières que nous nous sommes attribuées. Afin de ne pas avoir de problème avec le git, nous avons créé plusieurs branches (graphe et ihm-final) que nous avons merge au *master*, où se trouve la partie poo.

Bien sûr, nous ne nous sommes pas limités à nos parties, si l'un d'entre nous avait besoin d'aide nous pouvions aller à sa branche pour l'aider. Lorsque nous avions besoin d'une méthode dans l'une de nos parties, nous faisions appel au coéquipier en question ou nous le faisions nous même si possible.

Cette distribution des tâches nous a permis d'être plutôt efficaces étant donné que nous nous étions « spécialisés » dans notre domaine. Mais en contrepartie, si l'un de nos coéquipiers ne répondait pas à nos messages, nous perdions du temps à lire et comprendre ce qu'il avait codé.

Notre principale difficulté a été de comprendre le sujet. En commençant la partie graphe, nous nous sommes rendu compte que nous avions chacun une compréhension différente du sujet. L'un pensait qu'il s'agissait d'un échange et que le fait qu'un adolescent soit hôte ou invité n'avait pas de différence, l'autre ne s'avait pas à quel point nous étions libre de créer des méthodes, des classes, s'il était possible de modifier des éléments de l'UML et encore…