SAE 2.01-2.02 Rapport :

Création d'un logiciel d'appariement

ROTY Clément - Lemaire Mano
Timothée Sergheraert

Semestre 2
Année Universitaire 2024-2025

BUT 1 Informatique - Groupe B

TABLE DES MATIÈRES :

Comment lancer notre application :

Préambule :

Ayez installé, les versions de java et de javafx dans leurs dernières versions soit à la date de ce rapport Java 23.0.1 et JavaFX SDK 24.0.1.
De plus télécharger l'archive dans son intégralité et dézipper là vous devez avoir à la fin un fichier nommé B5.

Commande :

Pour les deux versions de l'application, veuillez placer votre terminal dans le répertoire B5.
Pour la version sans interface graphique, exécutez cette commande :

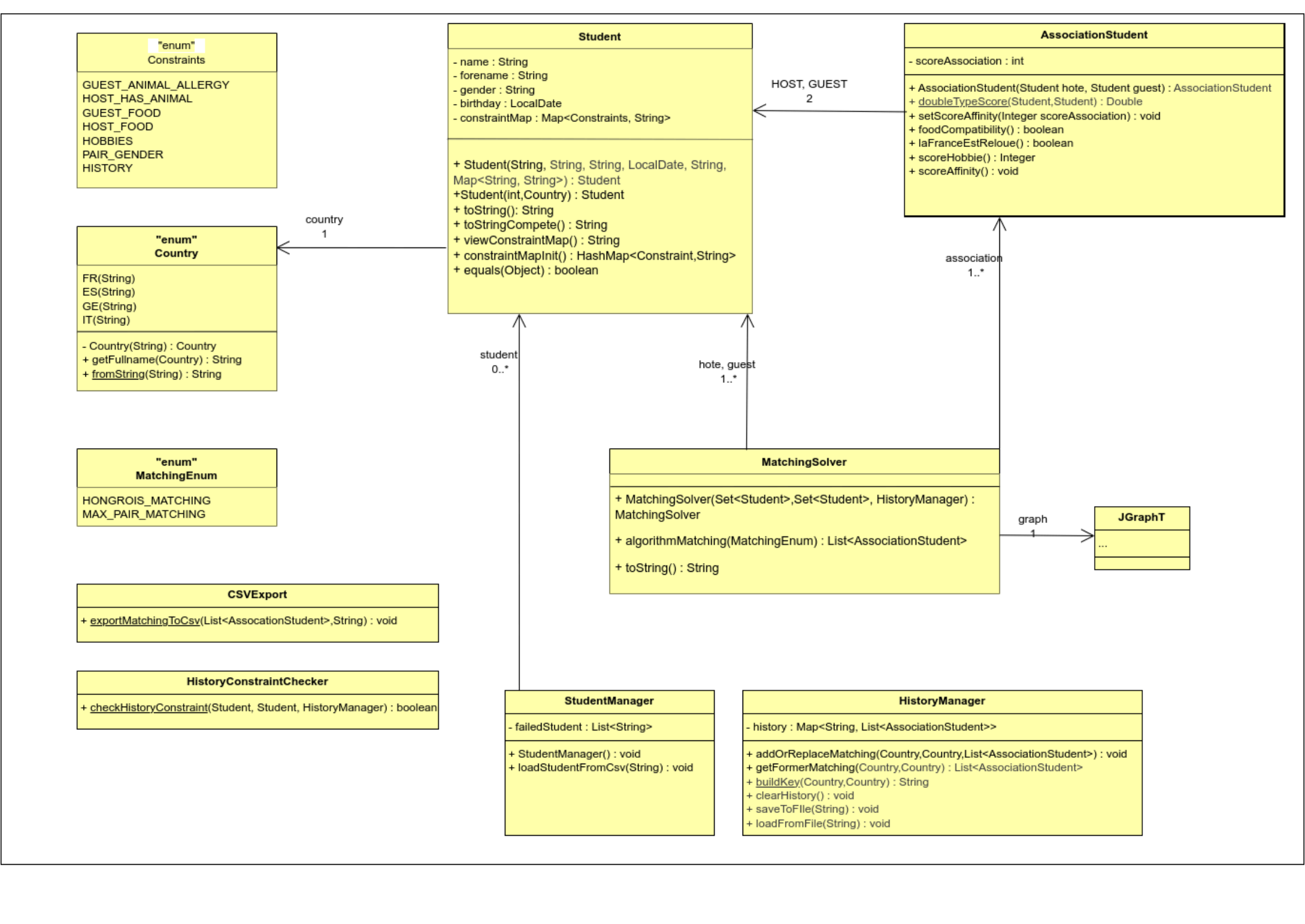
```
java -jar .\ApplicationWithoutInterface.jar
```

Pour la version avec interface graphique, exécutez cette commande :

```
java --module-path <votre_chemin>/javafx-sdk-24.0.1/lib --add-modules
javafx.controls,javafx.graphics,javafx.base,javafx.swing -jar
ApplicationFinale.jar
```

<votre_chemin> correspond à l'endroit où se situe le dossier contenant javaFx sur votre machine

Diagramme UML et réflexion sur les mécanismes objets


"enum"
Constraints
GUEST_ANIMAL_ALLERGY
HOST_HAS_ANIMAL
GUEST_FOOD
HOST_FOOD
HOBBIES
PAIR_GENDER
HISTORY
Student
- name : String
- forename : String
- gender : String
- birthday : LocalDate
- constraintMap : Map<Constraints, String>
+ Student(String, String, String, LocalDate, String, Map<String, String>) : Student
+Student(int,Country) : Student
+ toString(): String
+ toStringCompete() : String
+ viewConstraintMap() : String
+ constraintMapInit() : HashMap<Constraint,String>
+ equals(Object) : boolean
HOST, GUEST
2
AssociationStudent
- scoreAssociation : int
+ AssociationStudent(Student hote, Student guest) : AssociationStudent
+ doubleTypeScore(Student,Student) : Double
+ setScoreAffinity(Integer scoreAssociation) : void
+ foodCompatibility() : boolean
+ laFranceEstReloue() : boolean
+ scoreHobbie() : Integer
+ scoreAffinity() : void
country
1
"enum"
Country
FR(String)
ES(String)
GE(String)
IT(String)
- Country(String) : Country
+ getFullname(Country) : String
+ fromString(String) : String
association
1..*
student
0..*
hote, guest
1..*
"enum"
MatchingEnum
HONGROIS_MATCHING
MAX_PAIR_MATCHING
MatchingSolver
+ MatchingSolver(Set<Student>,Set<Student>, HistoryManager) : MatchingSolver
+ algorithmMatching(MatchingEnum) : List<AssociationStudent>
+ toString() : String
graph
1
JGraphT
...
CSVExport
+ exportMatchingToCsv(List<AssocationStudent>,String) : void
HistoryConstraintChecker
+ checkHistoryConstraint(Student, Student, HistoryManager) : boolean
StudentManager
- failedStudent : List<String>
+ StudentManager() : void
+ loadStudentFromCsv(String) : void
HistoryManager
- history : Map<String, List<AssociationStudent>>
+ addOrReplaceMatching(Country,Country,List<AssociationStudent>) : void
+ getFormerMatching(Country,Country) : List<AssociationStudent>
+ buildKey(Country,Country) : String
+ clearHistory() : void
+ saveToFIle(String) : void
+ loadFromFile(String) : void

Pour réaliser l'UML nous avons utiliser une extension VS Code nommé (drawio) qui nous a permis d'acquérir de nouvelles compétences, mais aussi pour la gestion de git car l'utiliser à plusieurs à créer quelques erreurs qui nous ont permis d'apprendre comment les gérer (git branch, git merge, prendre le réflexe de git pull avant de faire des modifications...).

Le diagramme UML ci-dessus, représente les principales classes et relations de l'application :

- **Student** : représente un étudiant, avec ses attributs (nom, prénom, genre, date de naissance, pays, contraintes).

- **AssociationStudent** : représente une association entre deux étudiants (hôte et invité), calcule le score d'affinité et fournit des méthodes d'analyse de compatibilité.

- **StudentManager** : gère le chargement des étudiants depuis un CSV, la validation des données et la gestion des étudiants rejetés.

- **HistoryManager** : gère l'historique des appariements, permet de sauvegarder/charger l'historique, d'ajouter ou de vider des appariements.

- **MatchingSolver** : construit le graphe biparti des associations possibles et applique les algorithmes de matching (Hongrois ou Hopcroft-Karp).

- **CSVExport** : utilitaire pour exporter les résultats d'appariement au format CSV.

- **HistoryConstraintChecker** : utilitaire pour vérifier les contraintes d'historique entre deux étudiants.

- **Enums** : Country, Constraints, MatchingEnum structurent les types de données et les options de l'application.

Mécanismes objets utilisés et justification

Encapsulation

Chaque classe de l'application encapsule proprement ses données. Les attributs sont déclarés privés ou protégés afin d'en interdire l'accès direct depuis l'extérieur. L'accès ou la modification de ces données se fait exclusivement via des accesseurs (getters) et mutateurs (setters), ce qui permet un contrôle total sur les valeurs autorisées.

Par exemple, la classe Student masque ses champs internes et propose des méthodes spécifiques pour accéder ou modifier les contraintes liées à un étudiant.

Intérêt :

Ce mécanisme assure l'intégrité des données, en empêchant les modifications involontaires ou incohérentes. Il évite les effets de bord et rend le système plus robuste face à d'éventuels changements futurs.

Abstraction

Les classes sont pensées pour représenter des entités métiers claires : un Student, une Association, un HistoryManager, un MatchingSolver, etc. Chaque classe est donc porteuse d'un sens métier fort, et non d'une simple structure de données.

Intérêt :

L'abstraction permet de travailler à un niveau conceptuel élevé. On raisonne sur des objets métier, ce qui améliore la lisibilité, la maintenabilité et la modularité du code. Le découplage entre logique métier et implémentation technique est ainsi facilité.

Énumérations

Le code utilise des énumérations (enum) pour représenter les pays, les contraintes et les types d'algorithmes de matching. Ce choix permet d'encadrer les valeurs possibles et d'éviter les erreurs de saisie ou les incohérences dans la logique applicative.

Intérêt :

Les enum renforcent la robustesse du système en limitant les cas à gérer et en assurant une cohérence de l'ensemble. De plus, elles simplifient le traitement des données dans l'interface utilisateur et dans les mécanismes internes.

Polymorphisme et modularité

Des méthodes comme scoreAffinity() ou describeLevelOfAffinity() sont surchargées ou spécialisées selon le contexte d'utilisation. Le choix de l'algorithme de matching est dynamique, piloté par une énumération (MatchingEnum) et l'utilisation de l'interface de la bibliothèque JGraphT.

Intérêt :

Le polymorphisme permet de modifier ou d'étendre le comportement du système sans affecter le reste du code. Il favorise une architecture modulaire, souple, et évolutive, où l'ajout d'un nouvel algorithme ou d'un nouveau type de contrainte est aisé.

Sérialisation

La classe HistoryManager implémente l'interface Serializable, ce qui permet de sauvegarder et recharger l'historique des appariements sur le disque.

Intérêt :

Ce mécanisme assure la persistance des données entre deux sessions d'utilisation, sans nécessiter un format de stockage personnalisé. L'implémentation est rapide, fonctionnelle et facilement réutilisable.

Séparation des responsabilités

Le code suit le principe de responsabilité unique. Chaque classe est dédiée à une tâche spécifique :

- StudentManager gère exclusivement le chargement et la validation des étudiants.
- MatchingSolver calcule les appariements à l'aide d'un graphe biparti.
- HistoryManager conserve la mémoire des appariements passés.
- Des classes utilitaires comme CSVExport ou HistoryConstraintChecker sont découplées des classes métier.

Intérêt :

Cette séparation améliore la lisibilité du code, facilite les tests unitaires et permet d'envisager des évolutions futures sans devoir réécrire de larges portions du programme.

Analyse technique et critique de l'implémentation

Chargement et validation des étudiants

Le module de chargement repose sur StudentManager, qui lit le fichier CSV fourni, valide la structure de chaque ligne, et sépare les étudiants valides des lignes rejetées. Les erreurs rencontrées (format incorrect, valeurs interdites, colonnes manquantes) sont automatiquement détectées et consignées dans des logs. Les lignes invalides sont listées pour que l'utilisateur puisse corriger le fichier.

Critique :

La validation est robuste et bien structurée. Toutefois, elle pourrait être enrichie par une interface graphique plus interactive, affichant les erreurs de façon détaillée avec des suggestions de correction en temps réel.

Calcul des appariements

La classe MatchingSolver modélise les étudiants sous forme de sommets dans un graphe biparti. Les arêtes du graphe sont pondérées par un score d'affinité, calculé à partir de la méthode AssociationStudent.scoreAffinity(). Deux algorithmes de matching sont disponibles : l'algorithme hongrois (pour un matching parfait de coût minimal) et Hopcroft-Karp (pour un matching maximal en cardinalité).

Avant la construction du graphe, les contraintes liées à l'historique sont intégrées via HistoryConstraintChecker. Lorsque les groupes sont déséquilibrés, des étudiants fictifs sont ajoutés afin de garantir la complétude de l'appariement.

Critique :

Le recours à la bibliothèque JGraphT est pertinent : il s'appuie sur des algorithmes éprouvés et évite les implémentations maison complexes. L'intégration des contraintes est claire, mais le traitement des étudiants fictifs mériterait d'être mieux isolé pour éviter toute confusion dans les résultats finaux.

Gestion de l'historique

HistoryManager conserve les appariements déjà réalisés, organisés par couples de pays. Chaque couple ne conserve qu'un seul appariement : toute nouvelle association écrase la précédente. L'historique est sérialisé dans un fichier, automatiquement rechargé au démarrage. Une fonction permet à l'utilisateur de réinitialiser l'historique à tout moment.

Critique :

Ce choix de simplicité évite la surcharge de gestion, mais il sacrifie la traçabilité : les anciens appariements sont perdus. Une évolution possible serait de conserver un historique complet, avec horodatage, afin de suivre l'évolution des relations entre pays au fil du temps. De plus, passer à un format de sérialisation plus portable (comme JSON ou CSV) permettrait une meilleure interopérabilité.

Export CSV

La classe CSVExport permet d'exporter les résultats d'appariement dans un fichier lisible, incluant toutes les informations utiles : identité des étudiants, score d'affinité, description textuelle, etc.

Critique :

L'export est fonctionnel et fiable. Il pourrait cependant être amélioré avec des options de personnalisation : choix des colonnes à inclure, format de date, nom du fichier, etc.

Robustesse et évolutivité

L'application présente une structure modulaire très satisfaisante. Chaque fonctionnalité est isolée dans une classe dédiée, facilitant la lecture du code et sa maintenance. Les exceptions sont gérées proprement, avec un système de log utile pour le débogage. Grâce à l'usage des énumérations et à la séparation des responsabilités, l'ajout de nouvelles fonctionnalités (nouveau pays, nouvelle contrainte, nouvel algorithme) est simple et localisé.

Limites et pistes d'amélioration :

- Les tests unitaires devraient être renforcés pour garantir une non-régression lors des évolutions du code.
- L'historique mériterait d'être enrichi pour conserver plusieurs appariements par couple de pays, avec possibilité de consulter et restaurer les anciens.
- BUG : problème d'affichage de la liste des étudiants qui n'ont pas trouvé leur bonheur. Pour faire simple si on a 6 étudiants, 3 hôtes et 3 invités, il se peut que si l'algo automatique (peu-importe le type choisi) trouve une pair réussi, lorsque qu'on clique sur le bouton pour les visualiser, 1 paires impossibles, ce qui est normal, puis 2 étudiants des 2 pays différents avec la mention aucun binôme disponible, chose réservé normalement à la personne seule si le nombre d'étudiant est impair. Nous pensons que le problème vient directement dans la manière dont parcours les algorithmes de JGraphT.

Analyse des Tests effectués

Dans le cadre du développement de notre application, nous avons mis en place une suite de tests unitaires visant à garantir la robustesse, la fiabilité et la maintenabilité de notre code métier. Ce rapport présente notre démarche, la pertinence des tests réalisés, les scénarios couverts, ainsi que les axes d'amélioration identifiés.

Notre démarche de test :

Dès le début du projet, nous avons fait le choix d'isoler la logique métier de l'interface graphique afin de pouvoir tester en profondeur toutes les fonctionnalités essentielles, sans dépendre de l'IHM. Nous avons ainsi concentré nos efforts de tests sur les packages basicclass, algorithm, manager et utils, qui constituent le cœur fonctionnel de l'application.
Pour chaque classe, nous avons identifié les méthodes publiques à tester, en veillant à couvrir :

- Les cas d'utilisation standards (parcours nominal)
- Les cas limites (valeurs extrêmes, données manquantes)
- Les cas d'erreur ou d'exception (fichiers invalides, contraintes incompatibles)
- Les scénarios métier spécifiques (règles France, historique d'appariement, etc.)
-

Analyse détaillée par classe :

Classe Student

Nous avons testé la création d'étudiants avec toutes leurs propriétés, le calcul de l'âge, la gestion des contraintes (ajout, lecture, affichage), ainsi que les méthodes d'égalité et de hachage. Nous avons également vérifié l'affichage textuel des étudiants.
Ces tests nous permettent de garantir que chaque étudiant est correctement instancié et manipulé, et que les contraintes sont bien prises en compte dans toutes les situations.

AssociationStudent

Pour cette classe centrale, nous avons vérifié la création d'associations, le calcul du score d'affinité, la description du niveau d'affinité, la compatibilité alimentaire, le calcul du score de hobbies, ainsi que la gestion des cas d'association invalide.
Nous avons pris soin de couvrir aussi bien les cas où l'association est possible que ceux où elle est bloquée par une contrainte, afin de garantir la fiabilité de la logique d'appariement.

Classe MatchingSolver

Nous avons testé l'algorithme d'appariement dans différents contextes : matching parfait, matching avec contraintes bloquantes, et récupération des associations valides ou invalides.
Ces tests nous assurent que l'algorithme produit des résultats cohérents, même en présence de contraintes fortes ou de jeux de données déséquilibrés.

Classe StudentManager

Nous avons simulé le chargement de fichiers CSV valides et invalides, afin de vérifier la robustesse du parsing et la gestion des erreurs.
Cela nous permet de garantir que l'application ne plante pas en cas de fichier mal formé et que les étudiants sont correctement importés.

Classe CSVExport

Nous avons vérifié l'export d'associations vers un fichier CSV, en testant aussi bien le cas d'une liste normale que celui d'une liste vide.
Nous nous assurons ainsi que l'utilisateur pourra toujours exporter ses résultats, quelle que soit la situation.

Classe HistoryConstraintChecker

Nous avons testé la vérification de la contrainte d'historique, en simulant des cas où deux étudiants ont déjà été appariés, ou non, ou encore des cas particuliers ("other").
Cela garantit que la logique d'historique fonctionne comme attendu et évite les doublons d'appariement.

Classe HistoryManager

Nous avons vérifié l'ajout, la récupération, la suppression et la persistance de l'historique des associations.
Ces tests sont essentiels pour garantir la traçabilité et la fiabilité des historiques d'appariement, même après redémarrage de l'application.

Scénarios couverts :

Nous avons veillé à couvrir :

- Les cas d'utilisation classiques (parcours utilisateur standard)
- Les cas limites (données vides, extrêmes, ou incohérentes)
- Les cas d'erreur (fichiers absents, contraintes incompatibles, etc.)
- Les règles métier spécifiques (règle France, historique, etc.)
- La robustesse des exports et imports de données

Points forts et axes d'amélioration :

Points forts

- Couverture large : toutes les méthodes publiques des classes métier sont testées.
- Robustesse : les tests anticipent les erreurs et vérifient la gestion des cas inattendus.
- Indépendance : chaque test est autonome et ne dépend pas de l'état global de l'application.
- Lisibilité : les tests sont clairs, bien nommés, et chaque assertion est explicite.

Axes d'amélioration

- Ajouter des tests de performance et de scalabilité (gros volumes de données).
- Tester tous les algorithmes d'appariement si plusieurs sont implémentés.
- Renforcer les tests de gestion d'exceptions pour tous les utilitaires.
- Ajouter des tests de persistance plus poussés (fichiers corrompus, droits d'accès, etc.).

Conclusion :

Grâce à cette démarche de tests, nous avons pu valider la solidité de notre code métier et anticiper de nombreux cas d'erreur.
La base de tests actuelle nous donne confiance dans la stabilité de l'application et facilite la maintenance et l'évolution du projet.
Nous restons attentifs à compléter cette couverture au fil des évolutions, notamment sur les aspects de performance et de gestion d'exceptions.