RAPPORT POO - SAE 2.02-2.03 - 2025
Marescaux Alex - Deflou Aliocha - Faust Valentin

Sommaire

1. Lancement et Utilisation de l'Application

Position des ressources nécessaires

L'application repose sur une organisation bien définie des fichiers :

- Le dossier `C6` contient l’ensemble des fichiers sources et scripts nécessaires au fonctionnement.
- Les fichiers de configuration et de sauvegarde sont placés dans des sous-dossiers dédiés.

Commande de lancement et options

Pour exécuter l'application sans interface graphique :

1. Ouvrir un terminal et se placer dans le dossier `C6`

Compiler le programme via la commande :
sh
./compile.sh

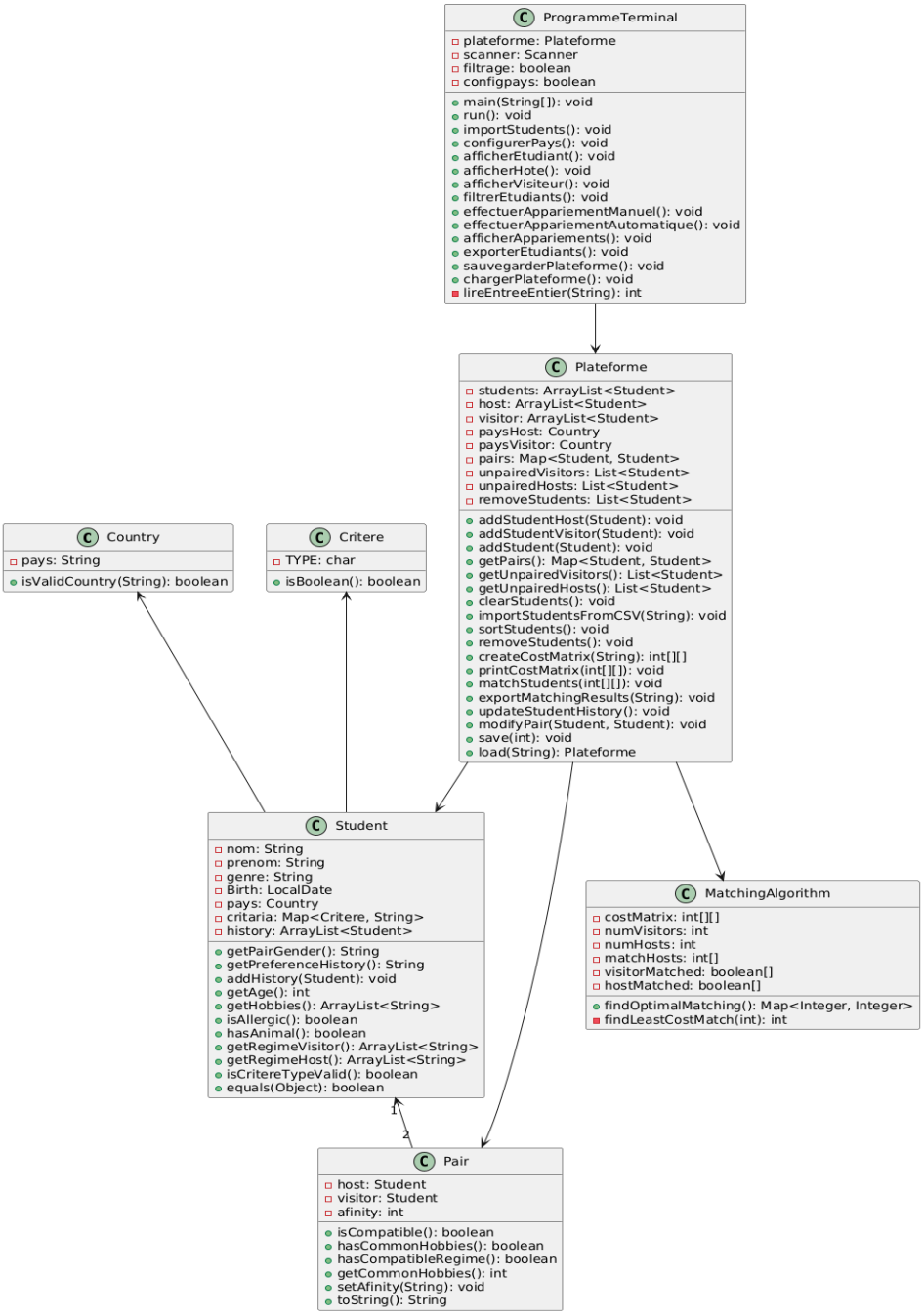
2.

Lancer l’application avec :
sh
./run.sh

3.

Des options supplémentaires peuvent être définies en modifiant les paramètres de configuration dans le fichier dédié.

Diagramme UML :



ps :Nous avons aussi développé des getters et constructeurs dans chaque classe ainsi que les méthodes toString mais ne les avons pas inclus dans l'UML car ne nous les considérons pas comme des fonctionnalités importantes.

Classe `Plateforme`

Rôle : Gestion des étudiants, de leur tri, des appariements et de l'historique.

Attributs

- `students : ArrayList<Student>` → Liste globale de tous les étudiants.
- `host : ArrayList<Student>` → Liste des étudiants hôtes.
- `visitor : ArrayList<Student>` → Liste des étudiants visiteurs.
- `paysHost : Country` → Pays des étudiants hôtes.
- `paysVisitor : Country` → Pays des étudiants visiteurs.
- `pairs : Map<Student, Student>` → Correspondances entre étudiants hôtes et visiteurs.
- `unpairedVisitors : List<Student>` → Liste des visiteurs sans correspondant.
- `unpairedHosts : List<Student>` → Liste des hôtes sans correspondant.
- `removeStudents : List<Student>` → Liste des étudiants supprimés pour non-conformité.

Méthodes

- `addStudentHost(student : Student) : void` → Ajoute un étudiant à la liste des hôtes.
- `addStudentVisitor(student : Student) : void` → Ajoute un étudiant à la liste des visiteurs.
- `setHostCountry(paysHost : Country) : void` → Définit le pays des hôtes.
- `setVisitorCountry(paysVisitor : Country) : void` → Définit le pays des visiteurs.
- `importStudentsFromCSV(filePath : String) : void` → Importe des étudiants depuis un fichier CSV.
- `removeStudents() : void` → Supprime les étudiants non conformes aux critères.
- `createCostMatrix(criterePrio : String) : int[][]` → Génère une matrice de coûts pour optimiser les appariements.
- `matchStudents(costMatrix : int[][]) : void` → Effectue l'appariement optimal entre hôtes et visiteurs.
- `exportMatchingResults(filePath : String) : void` → Enregistre les appariements dans un fichier CSV.
- `save(année : int) : void` → Sauvegarde l'historique par sérialisation binaire.

- `load(filePath : String) : static Plateforme` → Charge les données enregistrées.

Choix de conception :

- La séparation entre `host` et `visitor` facilite les opérations de tri et d'appariement.
- L'utilisation d'une `Map<Student, Student>` permet un accès rapide aux correspondances.
- La gestion de l'historique par sérialisation binaire permet de conserver les données après fermeture du programme.

Classe `MatchingAlgorithm`

Rôle : Optimisation de l'appariement des étudiants en fonction d'une matrice de coûts.

Attributs

- `costMatrix : int[][]` → Matrice de compatibilité entre hôtes et visiteurs.
- `matchHosts : int[]` → Tableau des correspondances trouvées.

Méthodes

- `findOptimalMatching() : Map<Integer, Integer>` → Retourne les meilleures correspondances basées sur le coût.
- `findLeastCostMatch(visitor : int) : int` → Recherche l'hôte le plus compatible pour un visiteur donné.

Choix de conception :

- Utilisation d'une matrice de coûts pour une approche algorithmique efficace.
- Stockage des correspondances sous forme de `Map<Integer, Integer>` pour un accès rapide aux résultats.

Classe `Student`

Rôle : Représente un étudiant avec ses informations et préférences pour l'appariement.

Attributs

- `nom : String` → Nom de l'étudiant.
- `prenom : String` → Prénom de l'étudiant.

- `genre : String` → Genre de l'étudiant.
- `Birth : LocalDate` → Date de naissance.
- `pays : Country` → Pays d'origine.
- `criteria : Map<Critere, String>` → Liste des préférences de l'étudiant.
- `history : ArrayList<Student>` → Historique des correspondances passées.

Méthodes

- `isAllergic() : boolean` → Vérifie si l'étudiant a des allergies.
- `hasAnimal() : boolean` → Vérifie si l'étudiant possède un animal.
- `addHistory(Student student) : void` → Ajoute un étudiant à l'historique des appariements.
- `isCritereTypeValid() : boolean` Vérifie si les valeurs des critères respectent les formats attendus.
- Si un critère est booléen (`yes` ou `no`), la méthode s'assure qu'il ne contient pas une valeur incorrecte.
- Cette validation est essentielle pour garantir une cohérence dans les comparaisons et éviter des erreurs lors de l'appariement.

Choix de conception :

- La `Map<Critere, String>` permet une gestion flexible et extensible des préférences des étudiants.
- L'historique des correspondances améliore la qualité des futurs appariements en évitant les erreurs passées.

Classe `Pair`

Rôle : Stocke une correspondance entre deux étudiants et leur affinité.

Attributs

- `host : Student` → Étudiant hôte.
- `visitor : Student` → Étudiant visiteur.
- `afinity : int` → Niveau d'affinité.

Méthodes

- `isCompatible() : boolean` → Vérifie si les critères de l'étudiant sont compatibles.

- `setAfinity(String criterePrioritaire) : void` → Calcule l'affinité selon un critère donné.

Choix de conception :

- Encapsule la logique de compatibilité dans une classe distincte pour une séparation claire des responsabilités.
- Permet une évaluation précise des affinités sans modifier `Plateforme`.

Classe ProgrammeTerminal
Rôle : Interface utilisateur en ligne de commande pour gérer la plateforme.
Attributs
- plateforme : Plateforme → Instance de la plateforme gérant les étudiants et les appariements.
- scanner : Scanner → Permet de lire les entrées utilisateur.
- filtrage : boolean → Indique si un filtrage des étudiants est activé.
- configpays : boolean → Indique si la configuration des pays est activée.

Méthodes
- main(String[] args) : void → Point d'entrée du programme.
- run() : void → Lance l'interface utilisateur et gère les interactions.
- importStudents() : void → Importe des étudiants depuis un fichier.
- configurerPays() : void → Configure les pays des hôtes et visiteurs.
- afficherEtudiant() : void → Affiche la liste des étudiants.
- afficherHote() : void → Affiche la liste des hôtes.
- afficherVisiteur() : void → Affiche la liste des visiteurs.
- filtrerEtudiants() : void → Applique un filtrage sur les étudiants.
- effectuerAppariementManuel() : void → Permet de créer des appariements manuellement.
- effectuerAppariementAutomatique() : void → Lance l'appariement automatique.
- afficherAppariements() : void → Affiche les appariements réalisés.
- exporterEtudiants() : void → Exporte les données des étudiants dans un fichier.
- sauvegarderPlateforme() : void → Sauvegarde l'état de la plateforme.
- chargerPlateforme() : void → Charge une plateforme sauvegardée.

Choix de conception :
- La séparation entre l'interface utilisateur et la logique métier (gérée par la classe Plateforme) garantit une meilleure modularité et maintenabilité.

L'architecture du programme repose sur une structure orientée objet bien segmentée. Voici un aperçu des principales classes et de leurs interactions :

Classes principales

- `Plateforme` : Gestion des étudiants, des appariements et de l'historique.
- `Student` : Modélisation des étudiants avec leurs caractéristiques et préférences.
- `Pair` : Association entre un hôte et un visiteur avec calcul d'affinité.
- `MatchingAlgorithm` : Gestion de la matrice de coût et du processus d'appariement optimal.

Principes Objet mis en œuvre

- **Encapsulation** : Chaque classe gère ses propres attributs et méthodes pour assurer la cohésion du programme.
- **Polymorphisme** : Permet une gestion flexible des étudiants et des critères d'appariement.
- **Héritage** : Utilisé pour structurer les objets et faciliter leur manipulation.

Ces mécanismes assurent une meilleure **modularité** et **évolutivité** de l'application.

3. Analyse Technique et Critique de l'Implémentation

Forces de l'implémentation

- La séparation entre hôtes et visiteurs **facilite le tri et l'appariement**.
- L'utilisation d'une **Map<Student, Student>** permet un accès rapide aux correspondances.
- La **sérialisation binaire** optimise la gestion de l'historique sans alourdir le programme.

Faiblesses et pistes d'amélioration

- La matrice de coût pourrait être **optimisée pour réduire la complexité algorithmique**.
- La gestion des critères pourrait être **plus dynamique** afin d'intégrer de nouveaux paramètres sans modifier le code source.
- Un module de **visualisation graphique** améliorerait l'expérience utilisateur.

4. Analyse Quantitative et Qualitative des Tests

Types de tests réalisés

- **Tests unitaires** : Vérification des méthodes clés (`matchStudents()`, `importStudentsFromCSV()`).

- **Tests d’intégration** : Validation des interactions entre les différentes classes.
- **Tests de performance** : Analyse du temps de traitement des appariements en fonction du nombre d’étudiants.

Résultats et conclusions

- Les tests montrent que l’application **gère efficacement l’importation et le tri des étudiants**.
- L’algorithme d’appariement **fonctionne de manière fiable**, mais son **optimisation pourrait réduire le temps de calcul**.
- Les analyses quantitatives confirment une **bonne scalabilité** du programme.

Conclusion

Le projet a évolué sur plusieurs versions :

- **Version 1** : Mise en place des fonctionnalités de base.
- **Version 2** : Amélioration du tri et calcul de l'affinité.
- **Version 3** : Optimisation des appariements et gestion avancée des historiques.

Cette dernière version apporte une **modularité accrue**, une **approche algorithmique efficace** et une **gestion optimisée des étudiants**, avec des perspectives d’amélioration pour une gestion encore plus performante.