El ayadi Ashraf

Toche Steve-Marley

Rapport de projet Tennis

Table des matières

Introduction :

Dans le cadre du module de Programmation Orientée Objet, nous avons développé une application Java permettant de simuler un match de tennis.
L’objectif du mini-projet était d’appliquer l’ensemble des notions vues en cours tout en adoptant une démarche de développement claire, structurée et conforme aux bonnes pratiques.

Travailler en binôme nous a permis de répartir les tâches de manière cohérente et de confronter nos approches pour construire une architecture fiable. Le projet nous a donné l'occasion de mobiliser les concepts fondamentaux de la POO : encapsulation, héritage, énumérations, composition, gestion simple des exceptions, documentation Javadoc et respect des règles de nommage.

Nous avons choisi de modéliser le tennis car les règles du sport sont naturellement structurées : points, jeux, sets et match final. Cette hiérarchie se traduit très bien en classes, ce qui nous a permis de concevoir un programme lisible et bien organisé, tout en restant fidèle aux principes enseignés.

Présentation du projet :

Le but de notre mini-projet était de programmer une simulation de match de tennis en Java, en respectant la logique réelle du sport. L'idée n'était pas de refaire un jeu compliqué, mais plutôt d'avoir une structure claire qui nous permette d'utiliser correctement la programmation orientée objet.

Le fonctionnement général est assez simple : deux joueurs s'affrontent, point après point. Le programme gère automatiquement l'évolution du score, depuis les points d'un jeu jusqu'aux sets, et enfin jusqu'au gagnant du match.

Pour que ce soit lisible et bien organisé, on a séparé la logique en plusieurs classes. Chaque partie du match a sa responsabilité : une classe pour les jeux, une autre pour les sets, une pour le match complet, une pour les joueurs, etc. Le but était d'obtenir une structure propre où chaque élément sait ce qu'il doit faire, sans que tout soit mélangé.

On a choisi le tennis précisément parce que tout s'enchaîne de manière assez claire : un point fait avancer un jeu, un jeu fait avancer un set, et plusieurs sets déterminent le vainqueur. Ça nous a permis d'appliquer les notions du cours sans partir dans quelque chose d'irréaliste ou trop compliqué.

Organisation du projet et diagramme de classe :

Le projet a été conçu en suivant une organisation structurée, logique et cohérente autour des principes de la programmation orientée objet. L'objectif général était de simuler de manière réaliste la dynamique d'un tournoi de tennis, en représentant non seulement les joueurs, mais aussi les arbitres, les spectateurs, le déroulement des matchs et la progression du tournoi. Pour parvenir à une architecture claire et extensible, le code a été réparti en différents packages correspondant chacun à un domaine fonctionnel précis. Cette organisation permet une meilleure lisibilité, facilite la maintenance et soutient une séparation stricte des responsabilités.

Le noyau du projet est constitué du package responsable de la modélisation de base, dans lequel se trouve la classe Personne. Celle-ci représente l'ensemble des individus présents dans le projet et fournit les attributs communs : nom de naissance, prénom, date de naissance, nationalité, genre… Le type utilisé pour représenter la date de naissance est LocalDate, un choix technique important car il garantit la validité des dates, permet des calculs fiables (comme l'âge via Period.between) et évite les erreurs liées aux formats textuels. Toutes les classes qui incarnent des individus réels du monde du tennis héritent de cette classe mère, ce qui permet une factorisation optimale du code et limite les répétitions inutiles.

L'héritage est utilisé pour structurer sereinement le modèle : la classe abstraite Joueur étend Personne et ajoute les attributs spécifiques à un joueur professionnel, tels que sa main dominante (enum Main), son classement, son sponsor ou encore son entraîneur. À partir de cette base, deux spécialités sont définies : JoueurHomme et Joueuse, qui ajoutent un attribut simple (la couleur du short ou de la jupe) et héritent du reste du comportement. La classe Arbitre, également dérivée de Personne, représente l'arbitre de chaise qui annonce les scores, gère les fautes et tranche les litiges entre les joueurs. Une spécialisation, ArbitreCentral, permet de personnaliser encore plus le rôle, notamment en réécrivant la méthode annoncer. Une autre branche importante du modèle concerne les spectateurs. La classe abstraite Spectateur hérite de Personne et implémente l'interface Supporter, qui définit les réactions possibles dans les gradins. Cette interface joue un rôle central, car elle représente le comportement attendu d'un fan de tennis, qu'il s'agisse d'un homme ou d'une femme, et peut facilement être réutilisée pour d'autres types d'acteurs. Les spectateurs masculins et féminins héritent ensuite respectivement dans les classes SpectateurHomme et Spectatrice, chacun apportant son attribut propre comme la couleur de la chemise ou la description des lunettes.

Le déroulement d'un match a été conçu avec un découpage fidèle aux règles du tennis, basé sur une architecture hiérarchique et une utilisation judicieuse de la composition. Un Match est composé de plusieurs SetTennis, chacun contenant plusieurs instances de la classe Jeu, qui elle-même est constituée d'une suite d'objets Echange. Ce découpage permet une logique précise et progressive : un échange détermine un point, plusieurs points composent un jeu, plusieurs jeux composent un set, et plusieurs sets déterminent le vainqueur d'un match. À chaque niveau, une classe spécifique gère l'évolution du score ainsi que l'interaction avec l'arbitre et les joueurs concernés. Cette structure reflète la réalité du tennis tout en conservant une architecture simple à maintenir. Les enums CategorieMatch et NiveauMatch permettent quant à eux de définir respectivement le type de match (simple homme ou simple femme) et le tour du tableau (premier tour, huitième, quart, etc.).

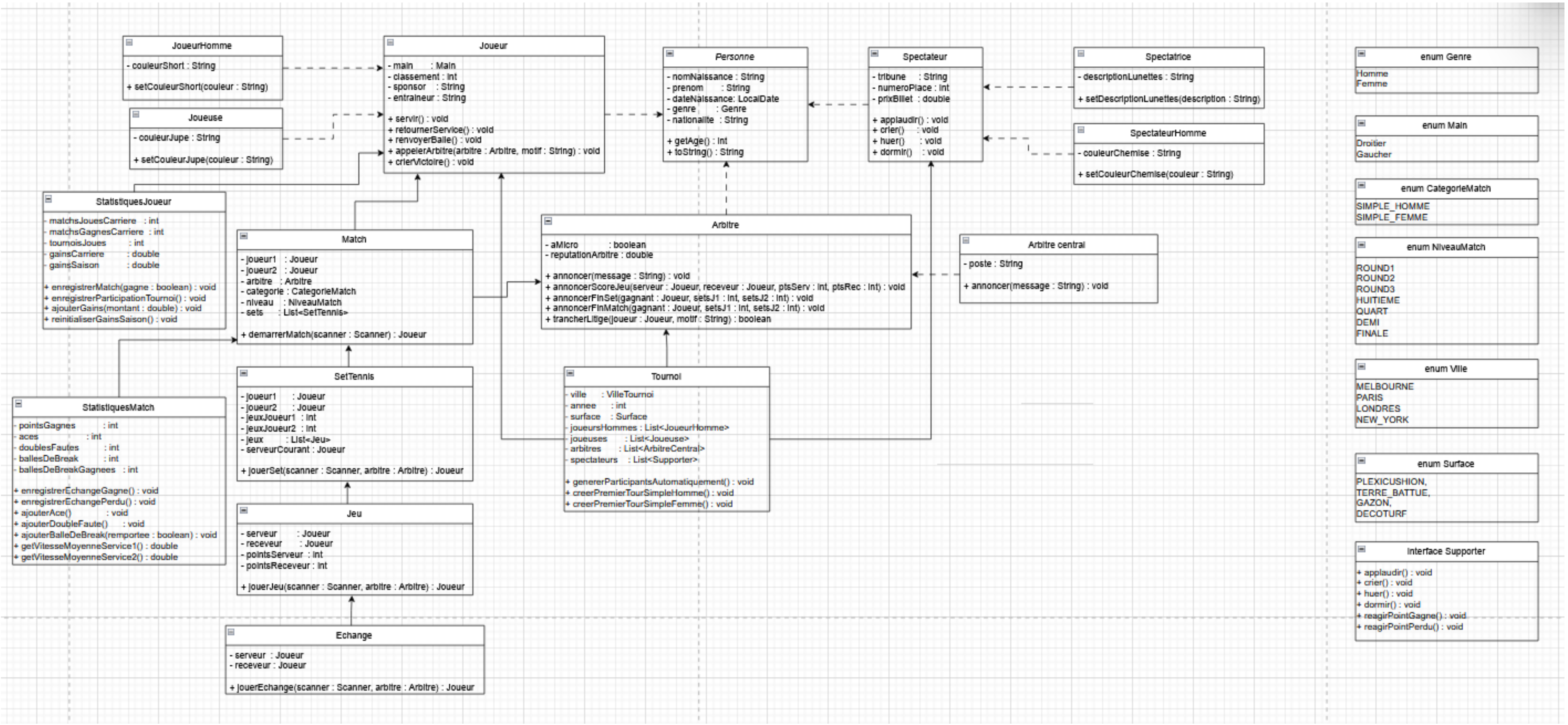
Par ailleurs, le projet inclut un volet statistique destiné à analyser la performance des joueurs. Deux classes distinctes ont été créées : StatistiquesJoueur, pour regrouper les statistiques globales d'un joueur sur sa carrière ou sa saison, et StatistiquesMatch, qui se concentre uniquement sur les statistiques relatives à un match donné. Ces classes constituent un module séparé qui pourrait facilement être réutilisé dans d'autres projets ou enrichi pour proposer un affichage graphique ou un tableau de bord. Pour éviter de surcharger l'architecture UML, seules les statistiques principales ont été retenues dans le diagramme de classes : nombre de points gagnés, aces, doubles fautes, balles de break, etc. Cela correspond pleinement aux consignes de clarté et de simplicité imposées par le professeur.

Au-dessus de l'ensemble du modèle se trouve la classe Tournoi, qui orchestre la gestion complète d'un tournoi de tennis du Grand Chelem. Un tournoi possède un lieu, une année et une surface déterminée automatiquement en fonction de la ville (terre battue à Paris, gazon à Londres…). C'est également lui qui gère la génération automatique des participants : 128 joueurs hommes, 128 joueuses, 10 arbitres et au moins 100 spectateurs. Une méthode permet de créer le premier tour du tableau, puis d'enchaîner les tours suivants en fonction des vainqueurs des matchs précédents. Le diagramme UML lie donc la classe Tournoi à ses participants, mais ne la relie pas directement aux matchs pour ne pas surcharger le schéma. Cette abstraction volontaire rend le diagramme lisible sans perdre la cohérence du fonctionnement général.

Enfin, d'un point de vue technique, plusieurs bonnes pratiques ont été appliquées. Les conditions if ont été écrites en respectant les règles de comparaison : utilisation de == pour les types primitifs (int, double) et utilisation de equals() pour les objets, afin d'éviter les erreurs fréquentes de comparaison de référence. Certaines portions de code ont été factorisées à l'aide de l'opérateur ternaire ?:, ce qui permet de réduire la taille de blocs if/else répétitifs tout en conservant une bonne lisibilité. Factoriser signifie simplifier un code trop long en le réécrivant de manière plus concise et plus élégante. Par exemple, pour initialiser un nom courant en utilisant un nom de naissance par défaut, le ternaire a permis de remplacer plusieurs lignes de code par une seule expression.

L'ensemble du diagramme de classes, présenté dans le rapport, met en évidence cette organisation claire : héritage bien structuré, utilisation d'une interface unique pour modéliser le comportement du supporter, composition précise entre les niveaux du match, enums pour limiter les valeurs possibles, classes spécialisées pour chaque rôle. Le diagramme est volontairement épuré, limité aux attributs et méthodes principaux, conformément aux recommandations du professeur. Ce travail d'organisation et de conception UML garantit non seulement une bonne compréhension du fonctionnement global, mais aussi une base solide pour de futures extensions du projet, comme l'ajout d'une interface graphique, d'une base de données ou d'une IA capable de simuler les échanges entre les joueurs.

Dans le diagramme de classes, les flèches pleines avec losange représentent les relations d'agrégation, tandis que les flèches pointées indiquent les liens d'héritage entre une classe mère et ses sous-classes.


JoueurHomme
- couleurShort : String
+ setCouleurShort(couleur : String)
Joueuse
- couleurJupe : String
+ setCouleurJupe(couleur : String)
Joueur
- main : Main
- classement : int
- sponsor : String
- entraineur : String
+ servir() : void
+ retournerService() : void
+ renvoyerBalle() : void
+ appelerArbitre(arbitre : Arbitre, motif : String) : void
+ crierVictoire() : void
Personne
- nomNaissance : String
- prenom : String
- dateNaissance: LocalDate
- genre : Genre
- nationalite : String
+ getAge() : int
+ toString() : String
Spectateur
- tribune : String
- numeroPlace : int
- prixBillet : double
+ applaudir() : void
+ crier() : void
+ huer() : void
+ dormir() : void
Spectatrice
- descriptionLunettes : String
+ setDescriptionLunettes(description : String)
SpectateurHomme
- couleurChemise : String
+ setCouleurChemise(couleur : String)
StatistiquesJoueur
- matchsJouesCarriere : int
- matchsGagnesCarriere : int
- tournoisJoues : int
- gainsCarriere : double
- gainsSaison : double
+ enregistrerMatch(gagne : boolean) : void
+ enregistrerParticipationTournoi() : void
+ ajouterGains(montant : double) : void
+ reinitialiserGainsSaison() : void
Match
- joueur1 : Joueur
- joueur2 : Joueur
- arbitre : Arbitre
- categorie : CategorieMatch
- niveau : NiveauMatch
- sets : List<SetTennis>
+ demarrerMatch(scanner : Scanner) : Joueur
Arbitre
- aMicro : boolean
- reputationArbitre : double
+ annoncer(message : String) : void
+ annoncerScoreJeu(serveur : Joueur, receveur : Joueur, ptsServ : int, ptsRec : int) : void
+ annoncerFinSet(gagnant : Joueur, setsJ1 : int, setsJ2 : int) : void
+ annoncerFinMatch(gagnant : Joueur, setsJ1 : int, setsJ2 : int) : void
+ trancherLitige(joueur : Joueur, motif : String) : boolean
Arbitre central
- poste : String
+ annoncer(message : String) : void
StatistiquesMatch
- pointsGagnes : int
- aces : int
- doublesFautes : int
- ballesDeBreak : int
- ballesDeBreakGagnees : int
+ enregistrerEchangeGagne() : void
+ enregistrerEchangePerdu() : void
+ ajouterAce() : void
+ ajouterDoubleFaute() : void
+ ajouterBalleDeBreak(remportee : boolean) : void
+ getVitesseMoyenneService1() : double
+ getVitesseMoyenneService2() : double
SetTennis
- joueur1 : Joueur
- joueur2 : Joueur
- jeuxJoueur1 : int
- jeuxJoueur2 : int
- jeux : List<Jeu>
- serveurCourant : Joueur
+ jouerSet(scanner : Scanner, arbitre : Arbitre) : Joueur
Tournoi
- ville : VilleTournoi
- annee : int
- surface : Surface
- joueursHommes : List<JoueurHomme>
- joueuses : List<Joueuse>
- arbitres : List<ArbitreCentral>
- spectateurs : List<Supporter>
+ genererParticipantsAutomatiquement() : void
+ creerPremierTourSimpleHomme() : void
+ creerPremierTourSimpleFemme() : void
Jeu
- serveur : Joueur
- receveur : Joueur
- pointsServeur : int
- pointsReceveur : int
+ jouerJeu(scanner : Scanner, arbitre : Arbitre) : Joueur
Echange
- serveur : Joueur
- receveur : Joueur
+ jouerEchange(scanner : Scanner, arbitre : Arbitre) : Joueur
enum Genre
Homme
Femme
enum Main
Droitier
Gaucher
enum CategorieMatch
SIMPLE_HOMME
SIMPLE_FEMME
enum NiveauMatch
ROUND1
ROUND2
ROUND3
HUITIEME
QUART
DEMI
FINALE
enum Ville
MELBOURNE
PARIS
LONDRES
NEW_YORK
enum Surface
PLEXICUSHION,
TERRE_BATTUE,
GAZON,
DECOTURF
Interface Supporter
+ applaudir() : void
+ crier() : void
+ huer() : void
+ dormir() : void
+ reagirPointGagne() : void
+ reagirPointPerdu() : void

Exemple de fonctionnement :

Le fonctionnement de notre programme reste volontairement simple et se fait entièrement via la console. Quand on lance l'application, elle commence par demander les noms des deux joueurs. Cela permet d'initialiser correctement les objets Joueur pour la suite du match.

Une fois le match lancé, l'utilisateur a accès à un petit menu qui permet de faire avancer la partie. Les options les plus importantes sont de donner un point au joueur 1 ou au joueur 2. À chaque point attribué, le programme met à jour le score du jeu en cours en suivant la logique du tennis.

Dès qu'un joueur gagne un jeu, on passe automatiquement au jeu suivant dans le set. Quand le set atteint un score suffisant pour être remporté, l'application bascule vers le set suivant. Tout cela se fait de manière transparente : l'utilisateur, lui, se contente d'indiquer qui marque les points.

À tout moment, le menu permet aussi d'afficher l'état actuel du score : points dans le jeu, jeux dans le set en cours, sets gagnés, etc. C'est utile pour suivre l'évolution du match, surtout lorsque les échanges deviennent serrés.

Le match se termine quand l'un des deux joueurs remporte suffisamment de sets pour être déclaré vainqueur. À ce moment-là, le programme affiche clairement le résultat final ainsi que le nom du gagnant, puis propose de quitter la simulation.

Difficultés rencontrer :

La première difficulté est venue dès le début, au moment de réfléchir à la structure générale du projet. Même en connaissant les grandes notions de POO, ce n'était pas évident d'imaginer directement comment organiser les classes entre elles. On a dû refaire plusieurs schémas avant de trouver une architecture qui nous paraissait cohérente.

La gestion de l'accessibilité des données a aussi été un vrai point de réflexion. Entre private, protected et public, on s'est rendu compte que certains choix bloquaient l'accès aux informations dans d'autres classes. Par exemple, certaines données protégées ne passaient pas dans certaines relations de composition. On a dû revoir plusieurs fois l'accès aux attributs, ajouter des getters, et faire attention à ne pas casser l'encapsulation.

L'interdépendance entre les classes a également causé pas mal de bugs au début. Comme le Match dépend du Set, qui dépend du Jeu, et que tout doit se mettre à jour correctement dans les deux sens, le moindre oubli dans une condition provoquait un comportement incorrect. On a passé du temps à corriger ces problèmes pour que tout s'enchaîne correctement : fin de jeu → mise à jour du set → mise à jour du match.

L'utilisation des énumérations a nécessité quelques essais aussi. On voulait avoir quelque chose de clair pour représenter le score d'un jeu (0, 15, 30, 40, avantage), et on a dû tester plusieurs façons de faire avant d'obtenir une enum vraiment pratique et simple à manipuler.

Pour ce qui est des **interfaces**, ça n'a pas été évident non plus. On a essayé de voir si elles pouvaient apporter quelque chose dans notre projet, mais comme une interface n'est utile que si au moins deux classes l'implémentent, ce n'était pas évident de trouver un cas où ça avait du sens. On a donc dû réfléchir à leur utilité réelle pour éviter d'en mettre juste "pour en mettre”. Au final, on a choisi de rester sur une approche plus simple, tout en gardant le principe en tête.

Une autre difficulté était de ne pas surcharger certaines classes, surtout la classe Match. Au début, on avait tendance à mettre trop de logique dedans, ce qui rendait le code difficile à suivre. On a progressivement réparti les responsabilités dans les classes Set et Jeu pour alléger la structure et garder des méthodes plus courtes, comme demandé dans le cours.

Enfin, on a eu plusieurs petits bugs liés au calcul du score, notamment sur les transitions entre “40–40” et “avantage”, ou encore lors du changement de set. Ce sont des détails qui semblent simples sur le papier, mais dans le code, la moindre erreur dans une condition pouvait bloquer tout le déroulement de la partie.

Malgré ces difficultés, ça nous a permis de mieux comprendre la logique de la POO, et surtout l’importance de bien penser l’organisation avant de coder.

Conclusion :

Ce projet nous a permis de mettre en pratique les différentes notions de programmation orientée objet vues pendant le cours. Même si l’organisation n’était pas évidente au début, travailler sur une application complète nous a aidés à mieux comprendre comment structurer plusieurs classes qui interagissent entre elles.

La simulation d’un match de tennis était un bon choix, car les règles sont simples à comprendre mais demandent tout de même une réflexion sérieuse pour les traduire correctement en code. On a dû réfléchir à la façon d’organiser l’architecture, aux relations entre les classes, aux visibilités, et à la manière de garder un code propre et cohérent.

Malgré les bugs et les ajustements nécessaires, on a réussi à obtenir un programme qui fonctionne et qui reste lisible. Ce travail nous a vraiment aidés à progresser, autant sur la technique que sur la méthode, et nous a donné une meilleure maîtrise de la POO.

Ce projet nous a aussi appris à mieux travailler en binôme, à communiquer sur nos choix et à garder une cohérence dans le développement. C’est une expérience qui nous servira clairement pour nos futurs travaux en Java ou dans d’autres langages orientés objet.