BORDEAUX INP Enseirb-Matmeca

Programmation Orientée Objet

Projet Schotten-Totten en Java

Rédigé par :
Ben Amor Eya
Mati Merwan

Encadrants :
Lakhlef Hicham &
Abakarim Fadwa

Décembre 2025

Table des matières

1 Introduction

Ce rapport présente la conception, l'architecture et la réalisation du projet *Schotten-Totten en Java*. L'objectif du projet est de développer une application permettant de jouer au jeu de société *Schotten-Totten*, en respectant les règles officielles et en proposant une structure logicielle modulaire, robuste et extensible.

Le jeu original oppose deux joueurs qui tentent de contrôler des bornes en posant des cartes pour former les meilleures combinaisons. L'application développée prend en charge la variante de base ainsi que la variante tactique, intégrant des cartes spéciales modifiant les règles du jeu.

2 Architecture du projet

Le projet respecte une architecture modulaire. Les classes sont réparties en plusieurs packages : `model`, `controller`, `view` et `ai`.

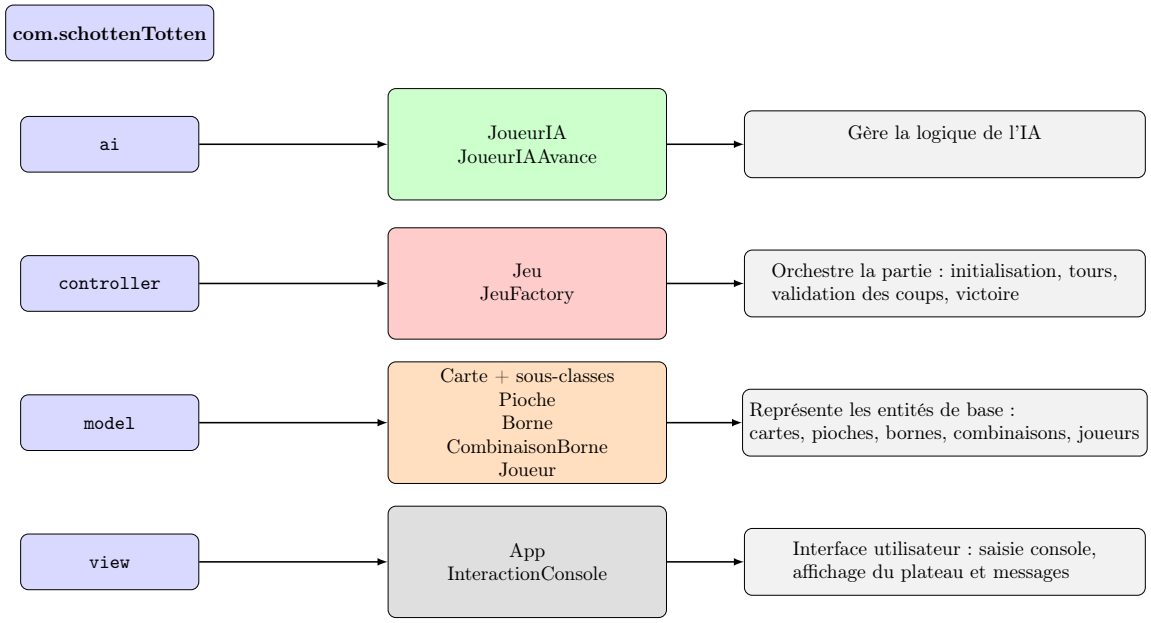

FIGURE 1 – Vue d'ensemble : packages, classes principales et responsabilités

2.1 Package `com.schottenTotten.model`

Ce package regroupe l'ensemble des classes représentant les éléments du jeu : cartes, bornes, combinaisons, joueurs, etc. Chaque relation entre classes a été pensée pour rester simple mais extensible (ajout d'autres variantes, d'autres IA, etc.).

Couleurs, cartes et pioche

Enum `Couleur`. L'énumération `Couleur` représente les six couleurs possibles des cartes Clan (`VERT`, `BLEU`, `ROUGE`, `JAUNE`, `MAUVE`, `MARRON`). Elle est utilisée comme type d'attribut dans `CarteClan`.

Classe abstraite `Carte`. `Carte` est la classe mère abstraite de toutes les cartes du jeu. Elle définit l'interface commune :

- `getNom()` : nom affiché de la carte,
- `getValeur()` : valeur numérique,
- `getCouleur()` : couleur éventuelle.

Ce choix permet d'utiliser le polymorphisme partout dans le code : pioche, main du joueur, borne, etc. manipulent des `Carte` sans connaître le type concret.

Classe `CarteClan`. `CarteClan` représente les cartes classiques, définies par une couleur et une valeur entre 1 et 9.

Relations UML :

— **Héritage** : `CarteClan` *hérite* de `Carte`.

— **Association** : `CarteClan` possède un attribut de type `Couleur`.

Classe `CarteTactique`. `CarteTactique` modélise les cartes spéciales de la variante tactique (Joker, Espion, etc.).

Relations UML :

— **Héritage** : `CarteTactique` hérite également de `Carte`.

— **Association** : un attribut de type `TypeTactique` indique l'effet de la carte.

Classe `Pioche`. `Pioche` représente une pile de cartes. Elle stocke les cartes dans une `List<Carte>` et offre les opérations classiques : `piocher()`, `estVide()`, `mettreSous()`, etc.

Relation UML :

— **Composition** entre `Pioche` et `Carte` : la pioche contient et gère la collection de cartes.

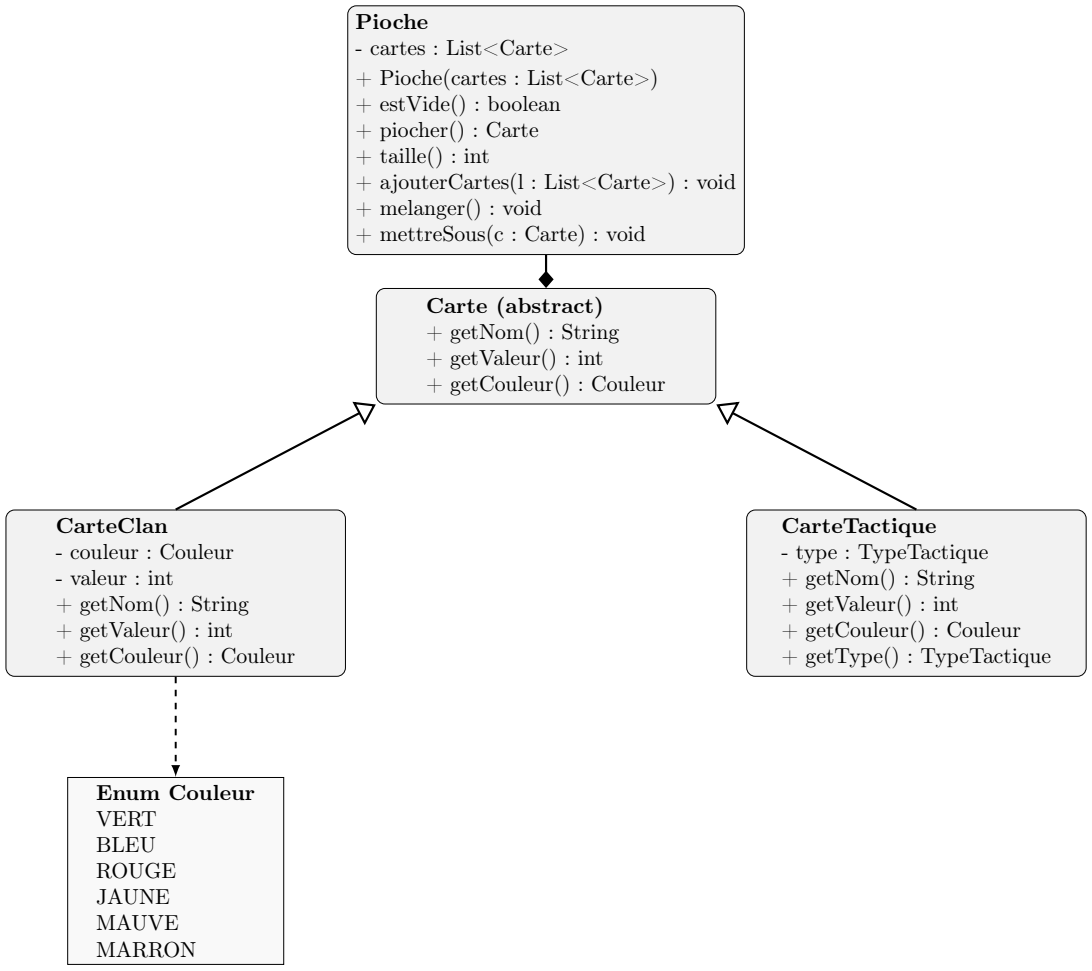

FIGURE 2 – Diagramme UML — `Pioche`, `Carte`, `CarteClan`, `CarteTactique` et `Couleur`

Bornes et combinaisons

Classe `Borne`. Une `Borne` modélise un emplacement de la frontière (il y en a 9). Elle conserve :

- son `index`,
- les cartes posées par chaque joueur (`cartesJoueur1`, `cartesJoueur2`),
- un `proprietaire` (nul tant qu'elle n'est pas revendiquée),
- un `modeCombat` optionnel (variante tactique).

La borne impose les contraintes de remplissage et encapsule le calcul du gagnant local. En cas d'égalité parfaite, la règle de départage est mémorisée par le premier joueur ayant complété sa 3^{e} carte sur cette borne.

Modes de combat tactiques. Le champ `modeCombat` (type `TypeTactique`) permet d'activer deux règles spéciales :

- `COLIN_MAILLARD` : la comparaison ignore les combinaisons (suite, brelan, etc.) et se fait *uniquement* sur la somme des valeurs des cartes posées.
- `COMBAT_DE_BOUE` : chaque joueur doit poser **4 cartes** sur la borne. Pour comparer, le programme sélectionne ensuite les **3 meilleures cartes** (plus fortes valeurs) de chaque côté puis évalue une combinaison classique sur ces 3 cartes.

Classe `CombinaisonBorne`. `CombinaisonBorne` représente le résultat d'analyse d'un groupe de cartes :

- un `type` (`TypeCombinaison`) qui porte une `force` (1 à 5),
- une `somme` des valeurs,
- une comparaison `compareTo` : d'abord la force du type, puis la somme.

Prise en compte des troupes d'élite(Joker, Espion, Porte-Bouclier) dans l'analyse :

- `JOKER` : remplace chaque Joker par toutes les cartes Clan possibles (couleur × valeur) et conserve la meilleure combinaison trouvée .
- `ESPION` : remplace l'Espion par une carte Clan de valeur 7 ; la couleur est choisie de façon à maximiser la combinaison obtenue.
- `PORTE_BOUCLIER` : remplace le Porte-Bouclier par une carte Clan de valeur 1, 2 ou 3 ; couleur et valeur sont choisies pour maximiser la combinaison finale.

En l'absence de carte tactique, l'analyse applique les règles standard : suite couleur, brelan, couleur, suite, sinon somme.

Enums `TypeCombinaison` et `TypeTactique`.

- `TypeCombinaison` encode la hiérarchie des combinaisons via un entier `force` (`SUITE_COULEUR` > `BRELAN` > `COULEUR` > `SUITE` > `SOMME`).
- `TypeTactique` regroupe les cartes tactiques et les modes de combat ; ici `Borne` n'exploite directement que `COLIN_MAILLARD` et `COMBAT_DE_BOUE`.

Relations UML :

- `Borne` **contient** deux collections de `Carte` (agrégation).
- `Borne` **utilise** `CombinaisonBorne` pour évaluer et comparer les cartes (dépendance).
- `CombinaisonBorne` **référence** `TypeCombinaison` (attribut `type`).
- `Borne` **référence** `TypeTactique` (attribut `modeCombat`).

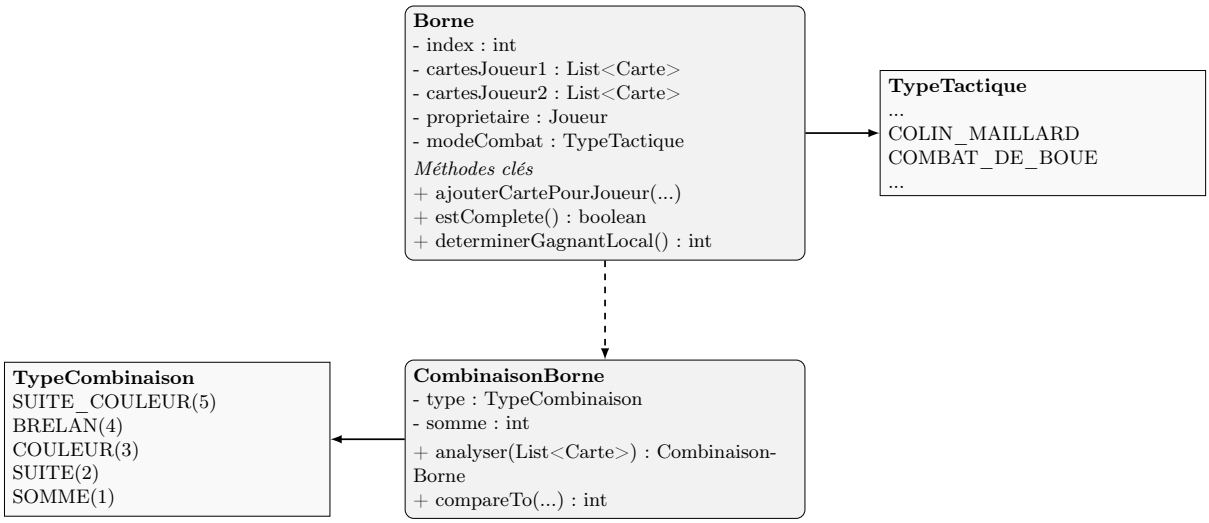

FIGURE 3 – Schéma UML simplifié — interactions entre `Borne`, `CombinaisonBorne` et les enums

Joueurs

Cette partie du modèle définit les acteurs de la partie. Pour garantir une séparation claire entre la structure du jeu et l'intelligence artificielle, les classes sont réparties sur deux packages (model et ai), bien qu'elles partagent la même hiérarchie. Les classes du package ai sont donc représentées ici pour montrer l'architecture globale mais leur fonctionnement sera expliqué en section 2.2.

Classe abstraite `Joueur` (package `model`). C'est la classe mère des différents types de joueurs. Elle gère :

- le nom du joueur,
- la main de cartes (attribut `protected` pour être accessible par les classes filles),
- les méthodes d'ajout et de retrait de cartes.

Implémentations.

- `JoueurHumain` (`model`) : hérite simplement du comportement par défaut.
- `JoueurIA` et `JoueurIAAvance` (`ai`) : ces classes étendent `Joueur` mais sont isolées dans le package `ai`. Elles implémentent leur propre logique de décision via la méthode `reflechirCoup`.

Relations UML :

- Agrégation avec Carte : un joueur possède une main de cartes, mais les cartes sont créées et détruites par d'autres composants (pioche,borne).
- Héritage : JoueurHumain et JoueurIA héritent de Joueur, et JoueurIAAvance hérite également de JoueurIA.

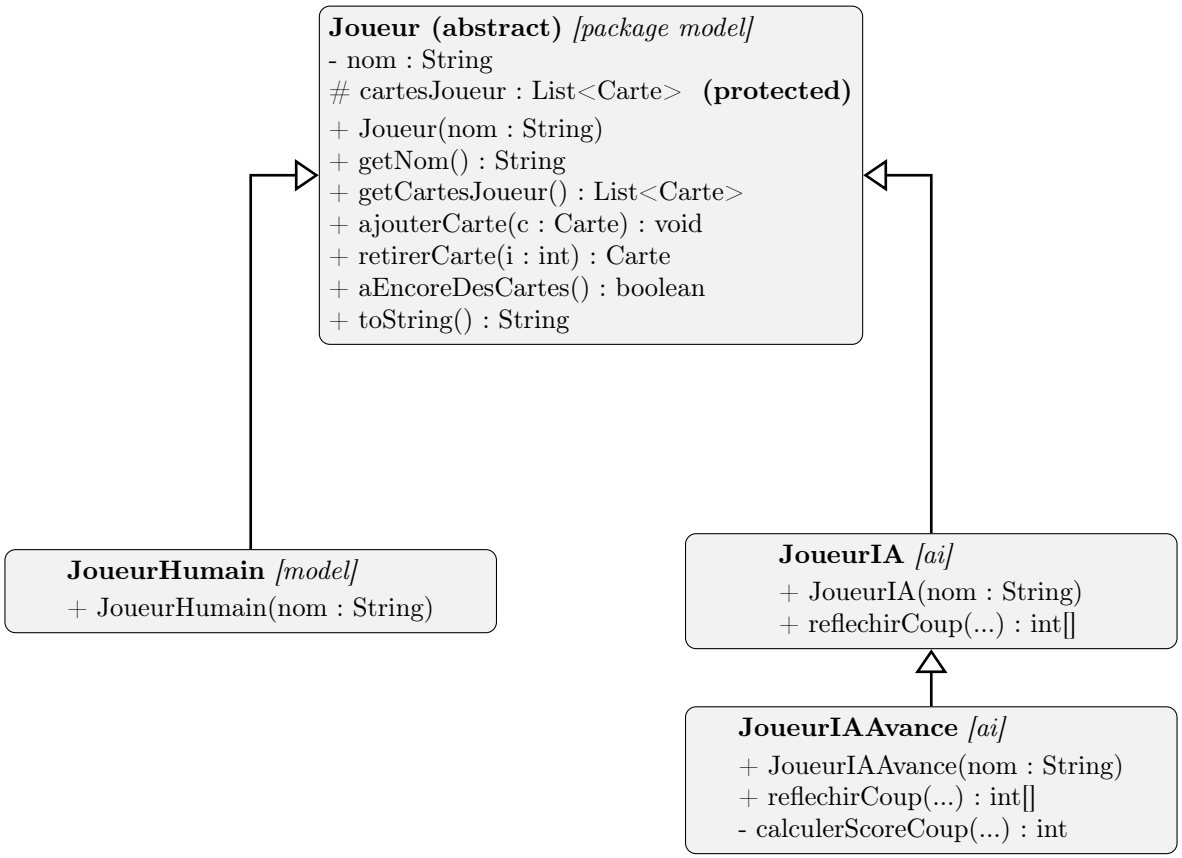

FIGURE 4 – Diagramme UML — Hiérarchie des Joueurs (Répartition sur packages model et ai)

2.2 Package `com.schottenTotten.ai`

Ce package contient la logique d'intelligence artificielle. Il exploite le polymorphisme de la classe `Joueur` dans le package model pour proposer différents niveaux de difficulté sans modifier le contrôleur.

Niveau 1 : IA Aléatoire (`JoueurIA`)

Cette classe représente le niveau de difficulté "Facile".

- **Stratégie :** elle sélectionne une carte au hasard dans sa main et une borne valide au hasard.
- **Validation :** elle n'est pas totalement aveugle : elle vérifie les contraintes du jeu (borne non pleine, respect de la limitation des cartes Tactiques via la méthode `reflechirCoup`) avant de valider son choix. Si le coup est illégal, elle retente une autre combinaison.

Niveau 2 : IA avancée (`JoueurIAAvance`)

Cette classe hérite de `JoueurIA` mais redéfinit la méthode `reflechirCoup` pour adopter un comportement réfléchi. Elle utilise un algorithme de recherche du meilleur coup local.

Fonctionnement : pour chaque carte de sa main et pour chaque borne disponible, l'IA simule la pose et calcule un score via la méthode privée `calculerScoreCoup`. Elle retient ensuite l'action ayant le score le plus élevé.

Système de coût : l'intelligence repose sur les valeurs arbitraires attribuées aux actions, que nous avons testé pour essayer de trouver des scores optimaux :

- **Construction de combinaisons :**
  - Potentiel de *Suite-Couleur* (même couleur + valeurs successives) : **+75 points**.
  - Potentiel de *Brelan* (même valeur) : **+45 points**.
  - Potentiel de *combinaison même couleur* : **+8 points**
  - Potentiel de *combinaison suite* : **+3 points**
- **Puissance brute :** Jouer une carte forte est valorisé (un 9 vaut **+28 pts**, un 8 vaut **+18 pts** et un 7 ou 6 vaut **+9 pts**).
- **Interactions :**
  - **Blocage :** si l'adversaire possède déjà des cartes sur la borne alors que l'IA n'y a rien posé, l'IA augmente le score de **+12 points** pour tenter de contester la victoire.
  - **Cartes Tactiques :** l'utilisation d'un Joker ou d'une carte spéciale reçoit un score fixe de **+20 points**, ce qui l'incite à les utiliser de manière prudente mais sans les délaisser.

2.3 Package `com.schottenTotten.controller`

Ce package contient l'intelligence centralisée de l'application. Il fait le lien entre les données du modèle, les affichages de la vue et les décisions des joueurs (humains ou artificiels).

Classe `Jeu`

La classe `Jeu` est le cœur du programme. Elle instancie les composants du modèle (Bornes, Pioches, Joueurs) et orchestre le déroulement de la partie. Ses responsabilités principales sont :

- **Initialisation :** la méthode `initialisationJeu` configure la partie selon les paramètres choisis (variante tactique activée ou non, niveau de l'IA).
- **Validation des coups :** la méthode `poserCarte` vérifie la légalité des actions. Par exemple, elle s'assure qu'un joueur ne joue pas plus de cartes Tactiques que son adversaire (règle de l'avantage tactique).
- **Boucle de jeu :** elle gère l'alternance des tours (`finirTour`), le déclenchement de l'IA (`jouerTourIA`) et la possibilité de revendiquer une borne, tout en gérant les cas d'erreurs.
- **Gestion des Ruses :** contrairement aux cartes classiques, certaines cartes Tactiques (Chasseur de Tête, Stratège...) ont un effet immédiat et ne restent pas sur le plateau. La méthode `executerEffetRuse` gère ces interactions spéciales.
- **Arbitrage :** la méthode `revendiquerBorne` appelle le modèle pour savoir qui gagne une borne, et `verifierVictoire` contrôle à chaque tour si une condition de fin de partie est remplie (5 bornes totales ou 3 adjacentes).

Classe `JeuFactory`

Cette classe implémente un design pattern. Son rôle est de décharger la classe principale de la complexité de création d'une partie. C'est elle qui interroge l'utilisateur (via les classes du package view) pour connaître le niveau de difficulté de l'IA souhaité avant d'instancier l'objet `Jeu` configuré, ainsi que le mode de jeu utilisé.

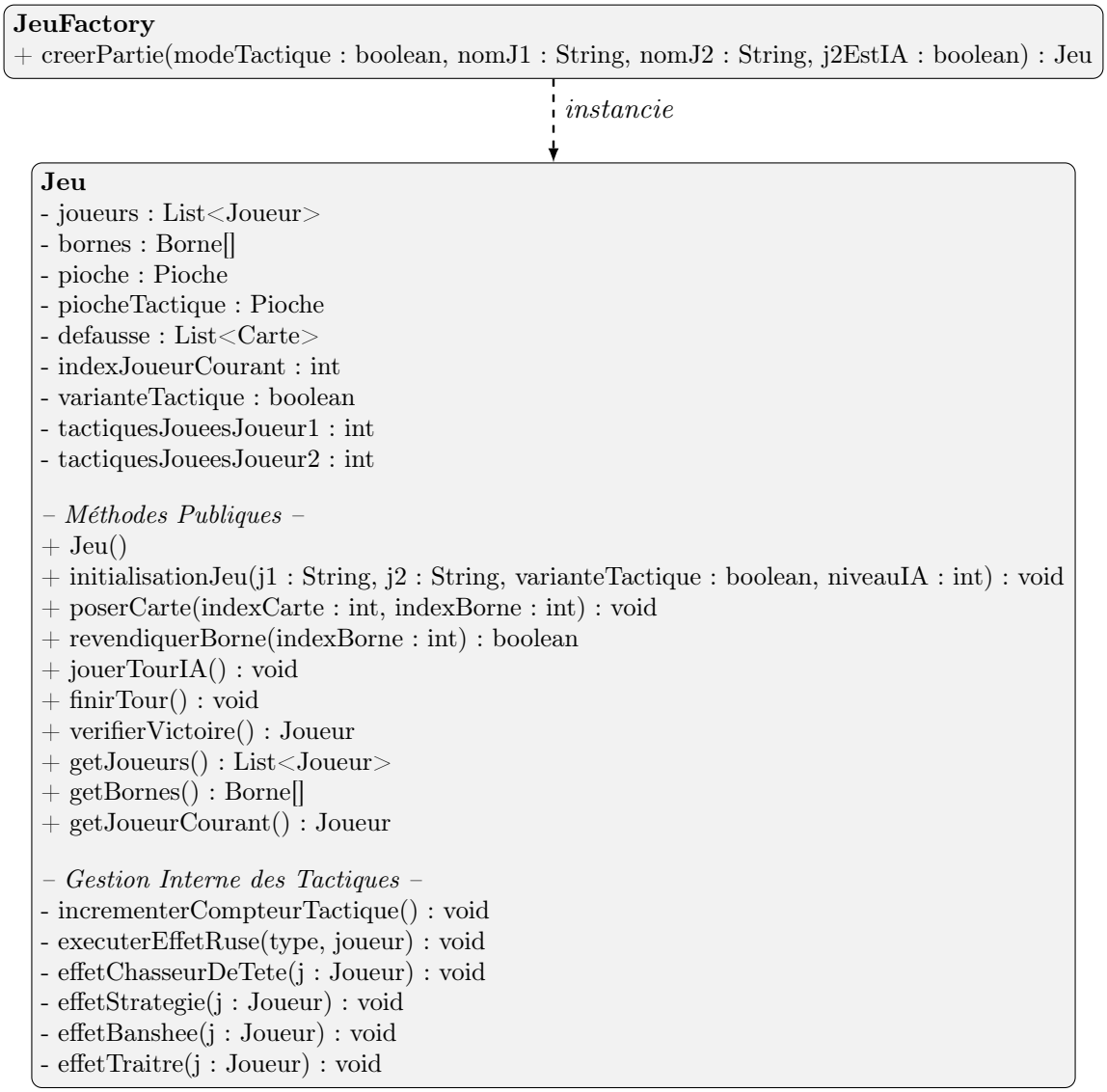

FIGURE 5 – Diagramme UML — Le Contrôleur et sa Fabrique

2.4 Package `com.schottenTotten.view`

Ce package gère l'interface utilisateur. Dans cette version du projet, il s'agit d'une interface en ligne de commande, mais la modularité de l'architecture permettrait de la remplacer par une interface graphique sans modifier le cœur du jeu.

Classe `InteractionConsole`

Cette classe utilitaire regroupe des méthodes statiques liées à l'affichage et à la saisie. Ses fonctions principales sont :

- **Affichage :** les méthodes `afficherCartes(List<Carte> list)` et `afficherMain(Joueur J)` formatent les objets du modèle pour les rendre lisibles dans la console (ex : affichage des bornes alignées, détails des cartes).
- **Saisie sécurisée :** les méthodes `demanderEntier(String message)` ou `demanderChoix(String message)` encapsulent la gestion de la classe `Scanner`, qui permet de lire les entrées utilisateur. Elles gèrent les erreurs de saisie (ex : si l'utilisateur rentre une lettre au lieu d'un chiffre) pour éviter que le programme ne plante.

Classe `App`

C'est le point d'entrée de l'application (contient le `main`). Elle orchestre la boucle de jeu principale :

1. Initialisation via `JeuFactory`.
2. Boucle `while` tant qu'il n'y a pas de vainqueur.
3. Alternance entre tour Humain (saisie via `InteractionConsole`) et tour IA, s'il y en a une.
4. Gestion des exceptions : Si le contrôleur renvoie une erreur (coup invalide), `App` affiche le message d'erreur et redemande une saisie au joueur.

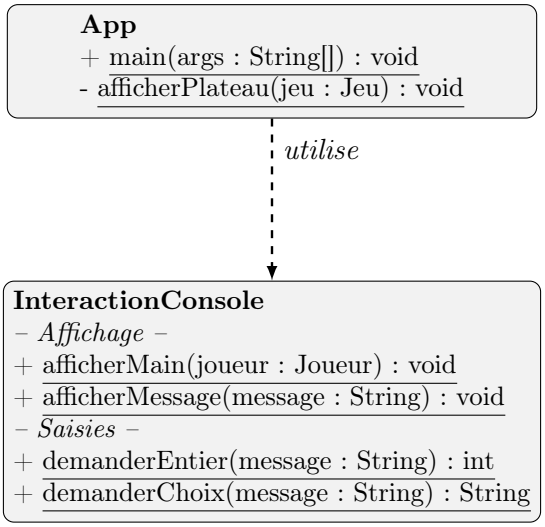

FIGURE 6 – Diagramme UML — La Vue Console

3 Conception détaillée et choix techniques

3.1 Modélisation de la force des mains (via encapsulation)

L'une des principales difficultés du jeu réside dans la comparaison des bornes. Un *Brelan* bat une *Couleur*, qui bat une *Suite*, etc. De plus, en cas d'égalité, c'est la somme qui tranche.

Pour éviter une multiplication complexe de conditions (`if/else`) dans la classe `Borne`, nous avons conçu la classe `CombinaisonBorne`, vue en section 2.1.

— **Principe :** cette classe encapsule le calcul de la force d'une main.

— **Fonctionnement :** lors de l'analyse d'une borne, nous transformons les 3 cartes en un objet `CombinaisonBorne` qui contient :

1. Un `TypeCombinaison` (enum ordonné par puissance : SUITE_COULEUR > BRELAN > ...).
2. La somme des valeurs des cartes.

— **Modularité** : si le mode tactique est activé, différentes méthodes de comparaison sont implémentées pour prendre en compte la carte Tactique présente sur la borne.

— **Avantage :** la comparaison devient simple grâce à l'interface `Comparable`. Pour savoir qui gagne, il suffit de comparer deux objets `CombinaisonBorne` entre eux, sans se soucier des cartes individuelles.

3.2 Gestion des joueurs (via polymorphisme)

Le jeu doit pouvoir faire s'affronter Humain vs Humain ou Humain vs IA. Pour ne pas surcharger le moteur de jeu avec des vérifications de type (`if joueur est IA alors...`), nous avons utilisé le polymorphisme.

- La classe abstraite `Joueur` définit le contrat commun.
- Le contrôleur `Jeu` ne manipule que des objets de type `Joueur`.
- Lors de l'appel à l'IA, le polymorphisme redirige automatiquement vers la méthode `reflechirCoup` spécifique à l'implémentation (aléatoire ou avancée).

Cela rend le code ouvert à l'extension : ajouter un "niveau 3" d'IA ne demanderait aucune modification dans la classe `Jeu`.

3.3 Création de partie (via un pattern Factory)

L'instanciation d'une partie implique de nombreux paramètres : noms des joueurs, activation du mode tactique, choix de la difficulté de l'IA. Pour respecter le principe de responsabilité unique, nous avons délégué cette tâche à la classe `JeuFactory` vue en section 2.2.

- Elle isole la logique de création (par exemple les `new`) du reste du programme.
- Elle permet de centraliser la sélection du niveau d'IA, rendant le `main` de l'application plus léger.

3.4 Gestion des erreurs et robustesse

Conformément aux exigences du cahier des charges, nous avons mis en place une gestion des erreurs à trois niveaux pour garantir que le jeu ne s'arrête jamais brutalement.

- **Validation des saisies (view) :** les classes de ce package sécurisent les entrées utilisateur via la classe `Scanner`. Si un utilisateur entre une chaîne de caractères alors qu'un entier est attendu (pour choisir une carte ou une borne), une exception est capturée localement et le programme redemande une saisie valide tant que nécessaire.
- **Protection du modèle (model) :** les classes du modèle défendent leur intégrité. Par exemple, la méthode `Borne.ajouterCartePourJoueur` lève une exception si l'on tente d'ajouter une carte sur une borne déjà pleine ou revendiquée. Cela empêche toute corruption de l'état du jeu par le contrôleur.
- **Boucle de récupération (controller/view) :** dans la classe principale `App`, la boucle de jeu est encapsulée dans un bloc `try-catch`. Si une action de jeu provoque une erreur logique (ex : tentative de jouer une carte Tactique interdite par la règle de l'avantage), l'exception est capturée, un message explicatif est affiché au joueur et le tour recommence sans planter l'application.

4 Tests et validation

Pour garantir la robustesse de l'application et le respect des règles officielles du Schotten-Totten, nous avons mis en place une suite de tests unitaires automatisés avec JUnit. Ces tests permettent de valider le fonctionnement de chaque composant indépendamment.

4.1 Organisation des tests

Les tests sont regroupés dans le package `test` et couvrent les trois couches de l'application :

4.1.1 Test du modèle (`CarteTest`)

Ce fichier valide les briques élémentaires du jeu.

— **Objectif :** vérifier que les cartes (Clan et Tactique) sont correctement instanciées.
— **Scénarios :** création d'une carte, vérification des valeurs et des couleurs.

4.1.2 Test de la logique des bornes (`BorneTest`)

Ce test valide l'algorithme de comparaison des mains (la classe `CombinaisonBorne`).
— **Objectif :** s'assurer que le jeu détermine correctement le vainqueur d'une borne.
— **Scénarios complexes :**
  — *Brelan vs Somme* : le Brelan doit gagner même si sa somme est inférieure.
  — *Suite-Couleur vs Brelan* : la Suite-Couleur doit l'emporter.
  — *Égalité parfaite* : vérification que le "premier arrivé" gagne la borne.

4.1.3 Test des conditions de victoire (`VictoireTest`)

Ce fichier simule des états de fin de partie pour vérifier que le contrôleur détecte bien le gagnant.
— **Victoire par majorité :** un joueur contrôle 5 bornes sur 9.
— **Victoire par adjacence :** un joueur contrôle 3 bornes consécutives.

4.1.4 Test de l'IA et robustesse (`JeuAvanceTest`)

Ce fichier valide le bon fonctionnement de l'intelligence artificielle et du mode Tactique.
— **Initialisation :** vérifie que le mode Tactique distribue bien 7 cartes au lieu de 6.
— **Légalité :** on demande à l'IA de calculer un coup dans une situation vide. Le test valide qu'elle renvoie un coup légal (non null) et ne provoque pas d'erreur d'exécution.

4.2 Bilan de la validation

L'exécution de la suite de tests (sur VSCode) confirme que l'ensemble des fonctionnalités sont opérationnelles.

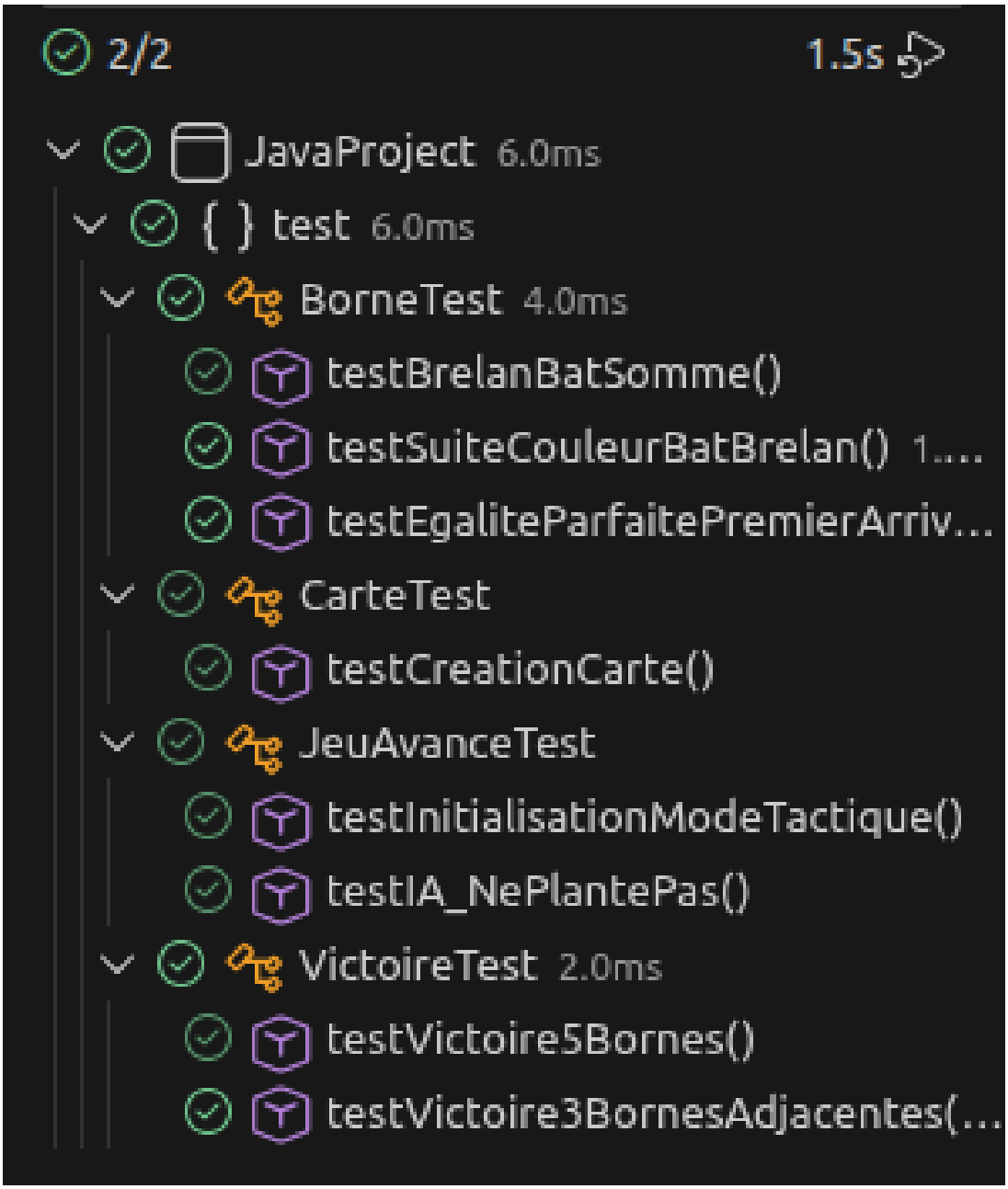

FIGURE 7 – Résultat de l'exécution de la suite de tests JUnit

4.3 Affichage console et expérience utilisateur

L'application propose une interface en ligne de commande pour permettre de jouer sans interface graphique. L'objectif est d'obtenir un affichage lisible, stable et facile à comprendre, tout en guidant le joueur à chaque étape.

Déroulement d'un tour (console). Chaque tour suit le même scénario :

1. Affichage de l'état courant du jeu : frontières (bornes) et cartes déjà posées.
2. Affichage de la main du joueur courant avec des indices `[0..n]`.
3. Demande d'action : choix de la carte à jouer (index), puis choix de la borne (index).
4. Validation par le contrôleur (`Jeu`) : si le coup est invalide, un message d'erreur est affiché et le joueur ressaisit.

Lisibilité. Les bornes sont listées au centre (`Borne 1..9`) avec les zones J1 et J2 de part et d'autre. Cela permet de visualiser rapidement l'avancement de la partie et les bornes encore contestables.

```
merwan@ADHJ-JUR:~/Cours/JavaProject$ java com.schottenTotten.view.App
========================================
   SCHOTTEN TOTTEN - IA & TACTIQUE
========================================
Nom du Joueur 1 (Vous) : merwan
Qui sera le Joueur 2 ?
1. Une IA
2. Un joueur humain
Votre choix (1 ou 2) : 1
Activer la variante tactique ? (o/n) : o
Niveau de difficulté de l'IA (1 : facile, 2 : difficile) : 2

La partie commence!

=== FRONTIÈRE ===
J1 []                                ( Borne 1 )                J2 []
J1 []                                ( Borne 2 )                J2 []
J1 []                                ( Borne 3 )                J2 []
J1 []                                ( Borne 4 )                J2 []
J1 []                                ( Borne 5 )                J2 []
J1 []                                ( Borne 6 )                J2 []
J1 []                                ( Borne 7 )                J2 []
J1 []                                ( Borne 8 )                J2 []
J1 []                                ( Borne 9 )                J2 []

--- C'est à merwan de jouer ---
Votre main :
  [0] CarteClan : MAUVE, valeur : 2
  [1] CarteClan : MARRON, valeur : 6
  [2] CarteClan : MARRON, valeur : 1
  [3] CarteClan : ROUGE, valeur : 2
  [4] CarteClan : MAUVE, valeur : 1
  [5] CarteClan : VERT, valeur : 5
  [6] CarteClan : ROUGE, valeur : 4

> Quelle carte jouer (index) ?
```

FIGURE 8 – Exemple d'affichage console : lancement d'une partie et saisie d'un coup

Avancement et fin du jeu. L'affichage en cours de jeu permet de voir l'état du plateau, les bornes déjà gagnées (et par qui), ainsi que la main à chaque tour. À la fin, le vainqueur est automatiquement affiché et le jeu s'arrête.

```
=== FRONTIÈRE ===
J1 [CarteClan : BLEU, valeur : 4]       ( Borne 1 )                J2 [CarteClan : ROUGE, valeur : 8, CarteClan : JAUNE, valeur : 8, CarteClan : ROUGE, valeur : 7]
J1 [CarteClan : ROUGE, valeur : 4, CarteClan : JAUNE, valeur : 4, CarteClan : MAUVE, valeur : 9] ( Borne 2 )                J2 [CarteClan : VERT, valeur : 9, CarteClan : BLEU, valeur : 9]
J1 [CarteClan : JAUNE, valeur : 2, CarteClan : VERT, valeur : 8, CarteClan : MAUVE, valeur : 8] [ GAGNÉE PAR m ]              J2 [CarteClan : MAUVE, valeur : 2, CarteClan : MAUVE, valeur : 1, CarteClan
: ROUGE, valeur : 1]
J1 [CarteClan : VERT, valeur : 5, CarteClan : ROUGE, valeur : 5, CarteClan : MAUVE, valeur : 7] ( Borne 4 )                J2 [CarteClan : ROUGE, valeur : 3, CarteClan : JAUNE, valeur : 5]
J1 [CarteClan : JAUNE, valeur : 3, CarteClan : BLEU, valeur : 2, CarteClan : MARRON, valeur : 2] [ GAGNÉE PAR IA Avancée ]     J2 [CarteClan : MARRON, valeur : 3, CarteClan : BLEU, valeur : 3, CarteClan
 : VERT, valeur : 3]
J1 [CarteClan : ROUGE, valeur : 9, CarteClan : MAUVE, valeur : 3, CarteClan : MAUVE, valeur : 4] [ GAGNÉE PAR IA Avancée ]     J2 [CarteClan : MAUVE, valeur : 5, CarteClan : BLEU, valeur : 6, CarteClan
: MAUVE, valeur : 6]
J1 [CarteClan : ROUGE, valeur : 6, CarteClan : ROUGE, valeur : 2, CarteClan : MARRON, valeur : 8] ( Borne 7 )                J2 [JOKER, CarteClan : BLEU, valeur : 1]
J1 [CarteClan : MARRON, valeur : 1, CarteClan : JAUNE, valeur : 9, JOKER] ( Borne 8 )                J2 [CarteClan : MARRON, valeur : 6, CarteClan : MARRON, valeur : 5, CarteClan : MARRON, valeur : 4
]
J1 [CarteClan : JAUNE, valeur : 6, CarteClan : MARRON, valeur : 7, CarteClan : VERT, valeur : 4] [ GAGNÉE PAR IA Avancée ]     J2 [CarteClan : VERT, valeur : 7, CarteClan : BLEU, valeur : 7, CarteClan :
 VERT, valeur : 6]

--- TOUR DE L'IA (IA Avancée) ---
IA joue sur borne 7
 IA gagne la borne 7
 IA gagne la borne 8

========================================
 VICTOIRE FINALE DE IA Avancée
========================================
```

FIGURE 9 – Exemple d'affichage console à la fin du jeu

5 Limites et axes d'améliorations

Bien que fonctionnelle, l'application a des possibilités d'amélioration :

— **Intelligence artificielle :** l'IA actuelle (niveau 2) utilise une méthode de coup immédiate. Elle ne prévoit pas les coups de l'adversaire à l'avance, ce qui la rend battable par un joueur entraîné. Une amélioration possible serait l'utilisation de l'algorithme Minimax pour explorer l'arbre des futurs coups possibles et prendre la meilleure décision possible. Comme mentionné, l'extensibilité de l'architecture actuelle permettrait d'ajouter cette version IA sans modification importante.

— **Complexité des ruses pour l'IA :** l'implémentation complète des effets des cartes tactiques de type 'Ruse' (Traître, Stratège...) demande une logique décisionnelle complexe (choix de cible, déplacement), et l'IA avancée actuelle joue ces cartes uniquement pour se défausser et faire tourner sa main, mais leur effet spécial est ignoré. Une version de cette IA plus robuste pourrait être mise en place pour tirer profit au maximum des Ruses.

— **Interface graphique :** l'architecture actuelle pourrait supporter une interface graphique à la place de l'interface console actuelle. Cela ne nécessiterait que peu de changements.

6 Conclusion

Ce projet nous a permis de mettre en pratique les principes de la programmation orientée objet, de concevoir une architecture modulaire et d'implémenter un jeu complet intégrant plusieurs variantes (classique et tactique). L'application est fonctionnelle et extensible : il est possible d'ajouter de nouvelles cartes tactiques, de nouvelles IA ou même d'autres variantes du jeu en s'appuyant sur le modèle déjà en place.