BORDEAUX INP Enseirb-Matmeca

Rapport

Projet Schotten-Totten en Java
Année 2024-2025

Rédigé par :
Mehdi Khabouze
Thomas Meyer
Samy Maach

Encadrant :
SABRI KHAMARI

Novembre - Décembre 2024

1 Introduction

Dans le cadre de ce projet, nous avons développés sur Eclipse le jeu de société Schotten-Totten. Ce projet vise à mettre en œuvre les principes de programmation orientée objet et à proposer plusieurs variantes de jeu : standard et tactique. Ces variantes améliorent l'expérience en intégrant des mécaniques différentes comme les cartes tactiques ou la modification des règles de revendication.

2 Démarche

Pour réaliser ce projet, nous avons créé plusieurs classes dans un ordre établi afin d'assurer une organisation optimale et une architecture claire. Bien entendu, il a souvent été nécessaire de revenir sur des classes déjà implémentées au fur et à mesure de l'avancement. Voici une description de chaque classe et de leur rôle :

- **Classe `Carte`** : Première classe créée, elle représente les cartes de base du jeu en incluant des attributs comme la couleur et la valeur. Elle a été conçue pour être extensible, permettant l'ajout de fonctionnalités spécifiques via des sous-classes comme `TacticalCard`.
- **Classe `Muraille`** : Cette classe représente les bornes que les joueurs doivent revendiquer. Elle gère les combinaisons de cartes jouées par chaque joueur et vérifie les conditions pour revendiquer une muraille. Elle inclut également des fonctionnalités spécifiques pour appliquer des effets tactiques, comme `activerCombatDeBoue()` ou `activerColinMaillard()`.
- **Classe `Joueur`** : Définit les propriétés d'un joueur, notamment son nom, sa main de cartes, et s'il s'agit d'une IA. Elle gère également les interactions avec les cartes en main, comme l'ajout ou le retrait de cartes après une action ou une pioche.
- **Classe `Jeu`** : La classe centrale qui met en place les règles principales du jeu. Elle gère les tours de jeu, l'interaction entre les joueurs et les murailles, ainsi que les variantes (Standard, Tactique, Experts). Cette classe coordonne également les piocheurs, la pile de défausse et les actions des cartes tactiques via des méthodes telles que `jouerTour()`, `piocherCarte()` et `verifierVictoire()`.
- **Classe `ConsoleView`** : Fournit une interface utilisateur textuelle permettant d'afficher l'état du jeu, de gérer les interactions des joueurs humains et d'afficher les actions réalisées par l'IA. Cette classe a été conçue pour fonctionner uniquement en local, sans besoin d'interface graphique.
- **Classe `TacticalCard`** : Une extension de la classe `Carte`, elle introduit les cartes tactiques avec des effets spécifiques. Chaque type de carte tactique (comme `BANSHEE` ou `TRAITRE`) est implémenté via la méthode `applyEffect()`, qui applique son effet en fonction de la muraille cible, du joueur qui joue la carte, et de l'adversaire.
- **Classe `IA`** : Ajoute une intelligence artificielle capable de jouer automatiquement en respectant les règles et en exploitant les cartes tactiques. Elle utilise des algorithmes pour choisir les meilleures actions, comme `choisirCarteStrategique()`, `jouerCarteTactique()` ou `choisirMurailleAleatoire()`.
- **Classe `ScoreManager`** : Permet de suivre les victoires et performances des joueurs sur plusieurs parties, en enregistrant les scores et en affichant un résumé à la fin de chaque partie.

- **Choix des variantes** : Trois variantes (`STANDARD`, `TACTIQUE`, et `EXPERTS`) ont été ajoutées pour améliorer l'expérience de jeu, chaque ajout modifiant les règles et introduisant des mécanismes supplémentaires.
- La variante STANDARD : la version classique du jeu.
- La variante TACTIQUE : la version tactique du jeu où des cartes tactiques sont ajoutées. Pour cela, deux decks ont été créés (`deck` et `TacticalDeck`). Nous avons implémenté chacune des 10 cartes tactiques (2 jokers et 8 cartes tactiques différentes), qui possèdent chacune leurs spécificités, grâce à des méthodes dédiées.
  - `COMBAT-DE-BOUE` : Permet d'étendre le nombre de cartes jouées de 3 à 4 sur la muraille où elle est jouée. Cette carte est gérée dans la méthode `activerCombatDeBoue(Muraille)`.
  - `CHASSEUR-DE-TETE` : Permet au joueur de piocher 3 cartes depuis un ou deux decks (`deck` ou `TacticalDeck`), comme par exemple deux cartes du deck normal et une du deck tactique. Ensuite, le joueur choisit deux parmi les 9 cartes présentes dans sa main et les replace dans la défausse. Cette logique est implémentée dans `jouerTour(...)` et `piocherCarte()`.
  - `COLIN-MAILLARD` : Rend les cartes de la muraille cibles invisibles, empêchant leur lecture. L'effet est activé par `activerColinMaillard()`.
  - `JOKER` : Cette carte est utilisée comme carte joker et permet au joueur de choisir une couleur et une valeur. Les choix sont effectués avec les méthodes `demanderCouleurJoker()` et `demanderValeurJoker()`.
  - `TRAITRE` : Permet de déplacer une carte adverse vers une autre muraille. Cette fonctionnalité est implémentée dans `jouerTraitre(...)`.
  - `BANSHEE` : Défausse une carte adverse sur une muraille non revendiquée. L'action est réalisée via `jouerBanshee(...)`.
  - `ESPION` : Permet de jouer une carte avec une couleur choisie. Les actions liées sont gérées avec `jouerEspion(...)`.
  - `STRATEGE` : Déplace une carte d'une muraille non revendiquée à une autre muraille non revendiquée. Cette fonctionnalité est implémentée dans `jouerStratege(...)`.
  - `PORTE-BOUCLIER` : Protège une muraille cible contre les effets d'autres cartes tactiques. L'effet est géré dans `ajouterCarte(...)`.
- La variante EXPERTS : Cette variante modifie les règles pour permettre de revendiquer une muraille uniquement au début du tour, avant de jouer une carte. Cette logique a été intégrée via la méthode `jouerTourIA()` et adaptée dans `revendiquerBorne()`.

En progressant de la version normale à la version tactique, puis en ajoutant l'IA, nous avons construit un projet modulaire, évolutif et qui gère tout types d'erreurs du jeu (pas assez de carte, numéro de carte ou borne inexistante...).
Nous n'avons pas développé d'interface graphique, car cela n'était pas requis dans le cadre du sujet. De plus, une telle interface serait inutile pour un jeu se déroulant uniquement en local, sur un seul ordinateur. En effet, l'absence de connexion en ligne entre deux ordinateurs rend impossible de cacher les cartes de chaque joueur à l'autre. Dans un contexte local, chaque joueur pourrait facilement voir la main de son adversaire, ce qui annulerait l'intérêt d'une interface graphique pour ce type de jeu.

3 Diagramme UML

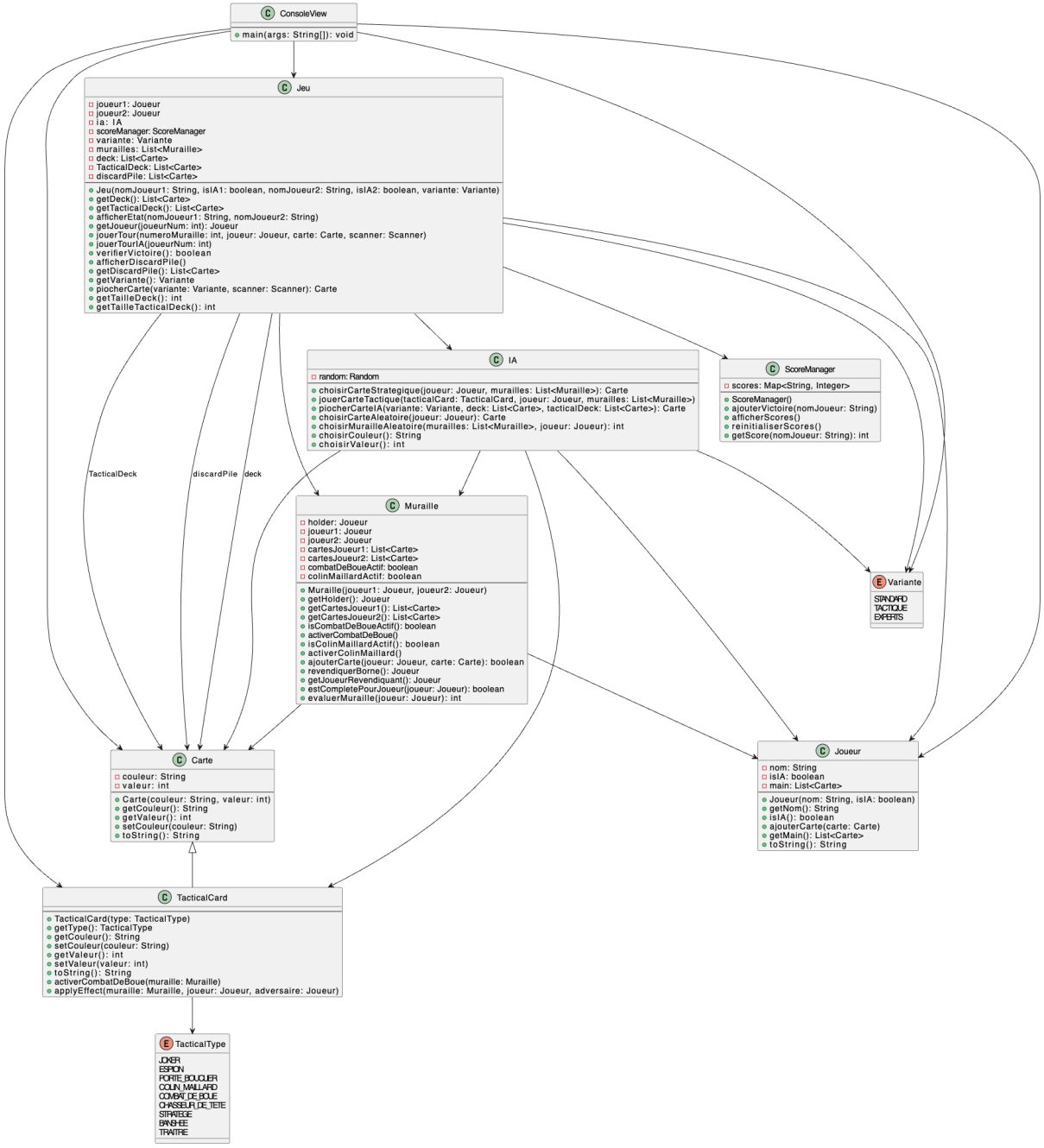

FIGURE 1 – Diagramme UML des classes du projet

4 Conclusion

Ce projet nous a permis de maîtriser les concepts de la programmation orientée objet, d'organiser efficacement un code modulaire et extensible, et de répondre aux contraintes des variantes du jeu Schotten-Totten. Nous avons désormais un jeu qui répond aux attentes et qui est tout à fait fonctionnel.