Projet - Année 2024/2025

Schotten-Totten en Java PG220

Rédigé par :

Oussama RAJI
Yassine HAMED

27 décembre 2024

Table des matières

1 Introduction

1.1 Contexte

Schotten-Totten est un jeu de cartes stratégique créé par le célèbre auteur de jeux Reiner Knizia. Publié pour la première fois en 1999, il s'est rapidement imposé comme un classique parmi les jeux de société. Ce jeu oppose deux joueurs dans une bataille tactique où l'objectif est de revendiquer des bornes en formant les meilleures combinaisons de cartes.

Les combinaisons possibles incluent des suites colorées, des brelans et des séquences. Chaque décision doit être stratégique pour contrôler les bornes tout en bloquant les opportunités adverses. Avec sa popularité croissante, une version enrichie, intitulée *Schotten-Totten 2*, a été introduite, ajoutant des éléments asymétriques pour diversifier les mécanismes de jeu.

1.2 Objectifs

Ce projet vise à reproduire l'expérience du jeu Schotten-Totten sous forme numérique en Java. L'objectif principal est de proposer une implémentation complète qui inclut plusieurs variantes de jeu (*Normal*, *Tactique*, et *Expert*), un système d'intelligence artificielle avec plusieurs niveaux de difficulté, et des interfaces utilisateur en mode console et graphique.

L'architecture du projet met un accent particulier sur l'extensibilité, permettant d'ajouter facilement de nouvelles variantes ou stratégies d'IA. Le projet vise également à offrir une interface conviviale pour permettre aux joueurs humains et virtuels de s'affronter dans un environnement immersif et stratégique.

1.3 Motivations

Ce projet présente un intérêt particulier pour plusieurs raisons :

— **Modélisation des règles complexes** : Reproduire fidèlement les mécaniques de jeu en codifiant les combinaisons et les stratégies.

— **Développement de l'IA** : Explorer des algorithmes adaptés à des jeux stratégiques pour simuler différents niveaux de difficulté.
— **Interface graphique** : Intégrer JavaFX pour créer une interface dynamique et engageante, offrant une expérience utilisateur intuitive.

En outre, ce projet permet de développer des compétences essentielles en programmation orientée objet et en gestion de projet logiciel.

2 Analyse des besoins

2.1 Fonctionnalités requises

— **Variantes de jeu** :
  — *Normal* : Règles de base avec une pioche standard et des revendications simples.
  — *Tactique* : Ajout de cartes tactiques offrant des pouvoirs spéciaux (espionnage, bouclier, ruse).
  — *Expert* : Limite augmentée de cartes par borne, et nouvelles contraintes stratégiques.
— **Gestion des joueurs** :
  — Support pour des joueurs humains et virtuels.
  — Différents niveaux d'IA (Facile, Moyen, Difficile).
— **Vérification des règles** :
  — Validation des combinaisons pour revendiquer une borne.
  — Détermination du gagnant en fonction des conditions de victoire (5 bornes ou 3 bornes adjacentes).

2.2 Public cible

Le projet s'adresse à plusieurs types d'utilisateurs :
— **Joueurs amateurs** : Les passionnés de jeux de société cherchant une version numérique de Schotten-Totten.
— **Développeurs** : Les étudiants ou professionnels souhaitant explorer la programmation orientée objet et les algorithmes d'IA.
— **Enseignants** : Utilisation comme outil pédagogique pour démontrer des concepts de POO et d'intelligence artificielle.

3 Conception du système

3.1 Architecture générale

L'architecture du projet est organisée de manière modulaire, divisée en plusieurs packages : *model*, *controller*, *view*, et *tests*. Chaque package a une responsabilité spécifique, facilitant ainsi la lisibilité, la maintenance, et l'extensibilité du code. Un diagramme d'architecture peut être utilisé pour illustrer les relations et les interactions entre ces composants.

3.2 Concepts utilisés

Programmation orientée objet (POO) : Le projet applique les principes fondamentaux de la POO :

- **Encapsulation** : Chaque classe regroupe ses données et méthodes, avec un contrôle strict des accès via des accesseurs (getters/setters) et des modificateurs de visibilité.
- **Héritage** : Les variantes du jeu (par exemple, *JeuNormal*, *JeuTactique*, *JeuExpert*) héritent d'une classe de base commune, partageant les comportements communs tout en permettant la spécialisation.
- **Polymorphisme** : Les différentes variantes exploitent des comportements spécifiques grâce à l'utilisation d'une interface ou de classes abstraites. Cela garantit une uniformité tout en facilitant l'extensibilité.

Design patterns : Un design pattern **Factory** est utilisé pour la création des variantes du jeu. Ce modèle centralise la logique d'instanciation, assurant une flexibilité dans l'ajout de nouvelles variantes.

3.3 Structure du projet

- **model** : Ce package contient les entités principales du jeu telles que :
  - *Carte* : Représente les cartes utilisées dans le jeu, avec des attributs comme la valeur et la couleur.
  - *Borne* : Représente les bornes à revendiquer, incluant la gestion des cartes posées par chaque joueur.
  - *Joueur* : Gère les informations et les actions des joueurs (humains ou IA).
  - Les différentes variantes du jeu (*JeuNormal*, *JeuTactique*, *JeuExpert*) sont également implémentées ici.
- **controller** : Ce package regroupe la logique de gestion :
  - *GestionnairePartie* : Supervise le déroulement d'une partie (tour par tour, pioche, revendication de bornes, etc.).
  - *GestionnaireVariante* : Configure les règles et les particularités de chaque variante de jeu.
  - L'interaction avec l'IA est également orchestrée ici.
- **view** : Contient les interfaces utilisateur :
  - *ConsoleUI* : Permet de jouer via une interface textuelle, adaptée pour un usage rapide ou des tests.
  - *SchottenTottenGUI* : Implémentée avec JavaFX, cette interface graphique offre une expérience utilisateur plus immersive et intuitive.
- **tests** : Ce package contient les tests unitaires, réalisés avec JUnit. Ces tests permettent de valider les fonctionnalités critiques, comme la gestion des cartes, les revendications des bornes, et les règles de victoire.

3.4 Structure des fichiers

Voici la structure des fichiers et des packages du projet :

```
projet_java/
 compile.sh                    # Script de compilation
```

```
com/schottenTotten/ model/                    # Package modèle (données et
↪  logique métier)
   IJeu.java             # Interface pour tous les types de jeu
   JeuFactory.java       # Factory pour créer les différentes variantes
   JeuNormal.java        # Implémentation du jeu normal
   JeuTactique.java      # Implémentation du jeu tactique
   JeuExpert.java        # Implémentation du jeu expert
   Carte.java            # Classe représentant une carte
   CarteTactique.java    # Classe pour les cartes tactiques
   Joueur.java           # Classe représentant un joueur
   Borne.java            # Classe représentant une borne

controller/               # Package contrôleur (gestion du jeu)
   variantes/
      VarianteJeu.java        # Énumération des variantes
      ConfigurationJeu.java    # Configuration des variantes
      GestionnaireVariante.java # Gestion des variantes
   GestionnairePartie.java      # Gestion de la partie

ai/                       # Package IA (intelligence artificielle)
   NiveauIA.java       # Énumération des niveaux de difficulté
   StrategieIA.java    # Stratégie de jeu de l'IA

view/                     # Package vue (interface graphique)
   SchottenTottenGUI.java  # Interface graphique JavaFX

tests/                    # Package tests
TestSuite.java       # Tests unitaires
```

Cette structure modulaire et claire facilite l'extension future du projet (ajout de nouvelles fonctionnalités, variantes ou niveaux d'IA) tout en maintenant une séparation stricte des responsabilités.

3.5 Instructions pour la Compilation et l'Exécution

Pour compiler et exécuter le projet **Schotten-Totten**, un script nommé `compile.sh` est fourni. Ce script automatise les étapes de compilation et de lancement du jeu. Voici les étapes à suivre :

Pré-requis

- Assurez-vous que **Java JDK** est installé sur votre machine.
- Vérifiez que le fichier `compile.sh` est présent dans le répertoire racine du projet.
- Accordez les permissions nécessaires au script en exécutant la commande suivante : `chmod +x compile.sh`

Compilation et exécution

Pour compiler et exécuter le projet, il suffit de lancer le script `compile.sh` depuis un terminal à l'aide de la commande suivante :

`./compile.sh`

Ce script réalise les actions suivantes :
— Compile tous les fichiers `.java` situés dans les sous-répertoires du projet.
— Génère les fichiers `.class` nécessaires à l'exécution.
— Lance l'application avec l'interface graphique.

Résolution des problèmes

Si des erreurs surviennent lors de la compilation ou de l'exécution, vérifiez les points suivants :
— Assurez-vous que toutes les dépendances requises pour JavaFX sont correctement configurées.
— Vérifiez que le chemin vers `compile.sh` est correct.
— Consultez les messages d'erreur affichés dans le terminal pour identifier le problème.

Avec ces étapes, vous pouvez facilement compiler et exécuter **Schotten-Totten** pour profiter de toutes ses fonctionnalités.

4 Implémentation

4.1 compile.sh

Rôle : Automatiser la compilation du projet en ligne de commande, en compilant tous les fichiers `.java` et en générant les fichiers `.class`.

Justification : Simplifie l'exécution du projet dans un environnement sans IDE et garantit une compilation uniforme.

4.2 Structure et Fonctionnalités du Package model

4.2.1 Borne.java

Description : La classe `Borne` modélise les bornes du jeu, sur lesquelles les joueurs placent leurs cartes pour tenter de les revendiquer.

Rôle : Gérer les cartes posées par chaque joueur, évaluer les combinaisons, et déterminer le propriétaire de la borne.

Fonctions principales :
— `ajouterCarte(Carte carte, Joueur joueur, Joueur joueur1)` : Ajoute une carte à la liste du joueur correspondant. Si les deux joueurs ont atteint le nombre maximum de cartes, la revendication est évaluée.
— `verifierRevendication(Joueur joueur1, Joueur joueur2)` : Compare les combinaisons des cartes des deux joueurs pour déterminer le propriétaire de la borne.
— `evaluerCombinaison(List cartes)` : Évalue les cartes posées pour calculer leur valeur selon les combinaisons (suite colorée, brelan, etc.).
— `estSuiteColoree(List cartes)`, `estBrelan(List cartes)`, etc. : Vérifient si les cartes posées correspondent à des combinaisons spécifiques.
— `peutAjouterCarte(Joueur joueur, Joueur joueur1)` : Vérifie si un joueur peut encore poser une carte sur cette borne.
— `setRevele(boolean revele)`, `setProtegee(boolean protegee)` : Configurent les attributs tactiques de la borne.

4.2.2 Carte.java

Description : La classe `Carte` représente les cartes standards du jeu, avec des attributs de valeur et de couleur.

Rôle : Modéliser les propriétés fondamentales d'une carte, nécessaires pour les règles du jeu.

Fonctions principales :

— `Carte(int valeur, String couleur)` : Constructeur pour initialiser une carte avec une valeur et une couleur.
— `getValeur()` : Retourne la valeur de la carte.
— `getCouleur()` : Retourne la couleur de la carte.
— `toString()` : Retourne une représentation textuelle de la carte.

4.2.3 CarteTactique.java

Description : `CarteTactique` étend `Carte` pour inclure des effets tactiques spécifiques aux variantes avancées.

Rôle : Ajouter des capacités tactiques aux cartes pour enrichir les mécanismes de jeu.

Fonctions principales :

— `CarteTactique(TypeTactique type)` : Constructeur qui initialise une carte tactique avec un type.
— `getType()` : Retourne le type de la carte tactique.
— `toString()` : Retourne une description textuelle du type tactique.

4.2.4 IJeu.java

Description : `IJeu` est une interface qui définit les comportements communs à toutes les variantes du jeu.

Rôle : Fournir une base uniforme pour les implémentations spécifiques des variantes.

Fonctions principales :

— `initialiser()` : Prépare le jeu (pioche, bornes, etc.).
— `jouerCarte(int indexCarte, int numeroBorne)` : Joue une carte sur une borne spécifique.
— `jouerTourIA()` : Permet à l'IA de jouer son tour.
— `estPartieTerminee()` : Vérifie si la partie est terminée.
— `determinerGagnant()` : Détermine le gagnant de la partie.

4.2.5 JeuNormal.java

Description : `JeuNormal` implémente les règles standard du jeu.

Rôle : Fournir la base fonctionnelle des règles, avec une pioche standard et des bornes classiques.

Fonctions principales :

— `initialiser()` : Prépare les bornes et distribue les cartes initiales.
— `creerPiocheStandard()` : Crée et mélange la pioche standard.
— `jouerCarte(int indexCarte, int numeroBorne)` : Joue une carte après validation.
— `verifierFinPartie()` : Vérifie les conditions de fin de partie.

4.2.6 JeuTactique.java

Description : `JeuTactique` étend `JeuNormal` pour introduire des cartes tactiques et leurs effets.

Rôle : Enrichir les règles du jeu en ajoutant des cartes tactiques aux stratégies possibles.

Fonctions principales :

— `ajouterCartesTactiques(List pioche)` : Ajoute des cartes tactiques à la pioche.

— `jouerCarteTactique(CarteTactique carte, int numeroBorne)` : Applique l'effet tactique d'une carte.

4.2.7 JeuExpert.java

Description : `JeuExpert` ajoute des contraintes avancées et étend les fonctionnalités tactiques.

Rôle : Offrir un niveau de complexité supplémentaire pour les joueurs expérimentés.

Fonctions principales :

— `estCoupValide(int indexCarte, int numeroBorne)` : Vérifie des règles spécifiques au mode expert.

4.2.8 JeuFactory.java

Description : `JeuFactory` utilise le design pattern Factory pour instancier les différentes variantes de jeu.

Rôle : Simplifier la création et garantir la validité des instances des variantes.

Fonctions principales :

— `creerJeu(VarianteJeu variante, String joueur1, String joueur2, NiveauIA niveau)` : Génère une instance de la variante demandée.

4.2.9 Joueur.java

Description : `Joueur` modélise les participants du jeu et leurs interactions avec les cartes.

Rôle : Gérer la main de cartes et différencier les joueurs humains des IA.

Fonctions principales :

— `ajouterCarte(Carte carte)` : Ajoute une carte à la main.

— `jouerCarte(int index)` : Joue une carte en la retirant de la main.

— `getMain()` : Retourne la main du joueur.

4.3 Structure et Fonctionnalités du Package Controller

4.3.1 GestionnairePartie.java

Description : La classe `GestionnairePartie` gère l'ensemble des interactions et de la logique d'une partie de Schotten-Totten. Elle orchestre les tours des joueurs, la distribution des cartes, et les vérifications des conditions de fin de partie.

Fonctions principales :

— `GestionnairePartie(VarianteJeu variante, String nomJoueur1, String nomJoueur2, NiveauIA niveau)` : Ce constructeur initialise les joueurs, les bornes, la pioche et

configure l'IA si nécessaire. L'utilisation de `GestionnaireVariante` garantit que la configuration correspond à la variante sélectionnée.

— `jouerCarte(int indexCarte, int numeroBorne)` : Permet au joueur courant de jouer une carte sur une borne. Avant d'ajouter la carte, elle vérifie si le coup est valide, met à jour la pioche, et passe au joueur suivant. Si c'est au tour de l'IA, son action est automatiquement exécutée.

— `estCoupValide(int indexCarte, int numeroBorne)` : Valide les conditions pour qu'une carte puisse être jouée sur une borne. Elle vérifie si la borne n'est pas revendiquée, si la carte est dans la main du joueur, et si la borne respecte la limite de cartes.

— `verifierFinPartie()` : Vérifie si la partie est terminée selon plusieurs critères : toutes les bornes revendiquées, 5 bornes gagnées par un joueur, ou 3 bornes adjacentes revendiquées par un même joueur. Cette méthode utilise des itérations efficaces pour évaluer ces conditions.

— `jouerTourIA()` : Détermine le meilleur coup pour l'IA à l'aide de `StrategieIA` et joue la carte choisie. Elle est directement liée au niveau de difficulté de l'IA, qui influence les décisions prises.

— `determinerGagnant()` : Compte les bornes revendiquées par chaque joueur et identifie le gagnant. En cas d'égalité, la méthode retourne `null` pour signaler un match nul.

— **Accesseurs :** Les méthodes comme `getBornes()` ou `getPioche()` permettent un accès contrôlé aux éléments du jeu, garantissant l'intégrité des données tout en facilitant l'affichage ou l'analyse.

4.3.2 ConfigurationJeu.java

Description : `ConfigurationJeu` encapsule les paramètres nécessaires pour personnaliser les variantes du jeu. Elle utilise le pattern Builder pour permettre une création modulaire des configurations.

Fonctions principales :

— **Builder** : La classe interne `Builder` permet de configurer les paramètres tels que le nombre de cartes, le nombre de bornes, ou l'utilisation de cartes tactiques.
  — `nombreCartes(int val)` : Définit le nombre total de cartes dans le jeu.
  — `nombreBornes(int val)` : Spécifie le nombre de bornes à jouer.
  — `cartesTactiques(boolean val)` : Active ou désactive les cartes tactiques.
  — `cartesParMain(int val)` : Définit le nombre de cartes distribuées par joueur au début de la partie.
  — `maxCartesParBorne(int val)` : Fixe la limite de cartes par borne.
  — `revendicationMultiple(boolean val)` : Permet ou non la revendication multiple sur une borne.

— **Accesseurs :** Les méthodes comme `getNombreCartes()` ou `hasCartesTactiques()` permettent d'accéder aux paramètres sans exposer directement les attributs.

4.3.3 GestionnaireVariante.java

Description : `GestionnaireVariante` est responsable de la configuration des variantes et de l'initialisation de la pioche en fonction des règles spécifiques.

Fonctions principales :

— `GestionnaireVariante(VarianteJeu variante)` : Initialise la configuration et prépare la pioche en fonction de la variante choisie (Normale, Tactique, ou Expert).
— `creerConfiguration(VarianteJeu variante)` : Crée une instance de `ConfigurationJeu` adaptée à la variante. Par exemple, pour la variante Tactique, elle configure 60 cartes et active les cartes tactiques.
— `initialiserPaquet()` : Prépare la pioche en combinant les cartes normales et, si nécessaire, les cartes tactiques. Le paquet est mélangé pour garantir une distribution équitable.
— `creerCartesNormales()` : Génère toutes les combinaisons possibles de cartes normales (valeurs de 1 à 9 pour chaque couleur).
— `creerCartesTactiques()` : Ajoute des cartes tactiques à la pioche, en tenant compte des effets spécifiques définis dans `CarteTactique.TypeTactique`.
— **Gestion des cartes tactiques :**
  — `peutJouerCarteTactique(CarteTactique carte, Joueur joueur, Borne borne)` : Vérifie si une carte tactique peut être jouée sur une borne en fonction de son type et de l'état du jeu.
  — `appliquerCarteTactique(CarteTactique carte, Joueur joueur, Borne borne)` : Applique les effets tactiques à une borne ou un joueur.
— **Accesseurs :**
  — `getPaquet()` : Retourne une copie sécurisée de la liste des cartes.
  — `paquetEstVide()` : Indique si la pioche est vide.
  — `piocherCarte()` : Retire et retourne la première carte du paquet.

4.3.4 VarianteJeu.java

Description : L'énumération `VarianteJeu` définit les trois variantes principales du jeu (Normal, Tactique, Expert).

Fonctions principales :
— `getNom()` : Retourne un nom descriptif pour identifier facilement la variante lors de l'affichage ou de la configuration.

Conclusion

Le package `controller` est conçu pour garantir une gestion cohérente et flexible des parties de Schotten-Totten. Chaque classe joue un rôle spécifique : `GestionnairePartie` pour orchestrer le déroulement du jeu, `ConfigurationJeu` et `GestionnaireVariante` pour personnaliser et configurer les parties, et `VarianteJeu` pour structurer les différentes options de jeu. Ensemble, elles assurent une extensibilité facile et une gestion robuste des règles de jeu.

4.4 Structure et Fonctionnalités du Package AI (Intelligence Artificielle)

4.4.1 NiveauIA.java

Description : Cette classe énumération définit trois niveaux de difficulté pour l'IA : `FACILE`, `MOYEN`, et `DIFFICILE`. Chaque niveau influence la stratégie adoptée par l'IA pour jouer une carte.

4.4.2 StrategieIA.java

Description : La classe `StrategieIA` implémente la logique permettant à l'IA de déterminer et de jouer le meilleur coup possible, en se basant sur la configuration actuelle des bornes, des cartes en main, et des cartes déjà posées.

Fonctions principales et Logique utilisée par l'IA :

- `StrategieIA(Joueur joueurIA, NiveauIA niveau, List bornes)` : Initialise l'IA avec son niveau de difficulté, les bornes du jeu, et les cartes disponibles dans sa main. Le niveau d'IA détermine la complexité de la stratégie utilisée dans les tours.
- `determinerMeilleurCoup()` : Parcourt toutes les cartes de la main de l'IA et évalue chaque carte pour chaque borne disponible. Les bornes disponibles sont celles qui ne sont pas encore revendiquées et qui ne sont pas complètes. La méthode retourne le coup ayant le meilleur score.

Logique utilisée pour déterminer le meilleur coup :

1. Limiter les bornes évaluées : Seules les bornes disponibles (non revendiquées et non complètes) sont prises en compte, réduisant ainsi le nombre de calculs nécessaires.
2. Évaluer chaque carte sur chaque borne :
   - Pour chaque carte de la main, un score est calculé pour toutes les bornes disponibles.
   - La méthode `evaluerPotentielCoup()` est utilisée pour calculer ce score.

Logique d'évaluation du potentiel d'un coup :

- `evaluerPotentielCoup(Carte carte, Borne borne, List mainComplete)` : Évalue l'impact de jouer une carte sur une borne donnée. Le score est déterminé par les critères suivants :
  - **Combinaisons formées** :
    - Une suite colorée rapporte +1000 points.
    - Un brelan rapporte +800 points.
    - Une suite rapporte +600 points.
    - Une couleur rapporte +400 points.
  - **Distribution stratégique** :
    - Favorise les bornes où l'IA n'a pas encore posé de cartes (+50 points).
    - Ajoute un bonus si la borne est proche d'être complétée (+200 points pour 2 cartes déjà posées).
    - Pénalise les bornes adjacentes déjà presque complètes (-50 points par borne adjacente).
  - **Blocage de l'adversaire** :
    - Si l'adversaire est sur le point de compléter une combinaison puissante, l'IA gagne des points supplémentaires en bloquant cette borne (+300 points pour une borne où l'adversaire a 2 cartes).

Fonctions auxiliaires pour les combinaisons :

- `peutFormerSuiteColoree()` : Vérifie si les cartes actuelles et celles en main permettent de former une suite colorée (trois cartes consécutives de la même couleur).

— `peutFormerBrelan()` : Vérifie si les cartes actuelles et celles en main permettent de former un brelan (trois cartes de la même valeur).
— `peutFormerSuite()` : Vérifie si les cartes actuelles et celles en main permettent de former une suite (trois cartes consécutives, peu importe la couleur).
— `peutFormerCouleur()` : Vérifie si les cartes actuelles et celles en main permettent de former trois cartes de la même couleur.

Gestion des niveaux de difficulté :

— **IA Facile** : Joue la première carte disponible sur la première borne non revendiquée. Ne prend pas en compte les combinaisons ou les blocages.
— **IA Moyenne** : Analyse les bornes disponibles et privilégie les coups qui favorisent des combinaisons simples ou empêchent l'adversaire de compléter une borne.
— **IA Difficile** : Évalue stratégiquement chaque coup en fonction des combinaisons possibles, des opportunités de blocage, et de la distribution optimale des cartes sur les bornes.

Conclusion

Le package `ai` combine simplicité et complexité pour fournir une IA adaptée à différents niveaux de joueurs. La logique de l'IA repose sur une évaluation systématique des coups possibles et une gestion stratégique des cartes, permettant à l'IA de s'adapter aux différents scénarios de jeu tout en conservant un comportement compétitif.

4.5 Structure et Fonctionnalités du Package `view`

4.5.1 ConsoleUI.java

Description : La classe `ConsoleUI` offre une interface utilisateur en ligne de commande pour jouer à Schotten-Totten. Elle utilise des menus interactifs pour gérer les interactions des joueurs et afficher l'état actuel de la partie.

Principe

`ConsoleUI` fonctionne sur un flux continu où le joueur interagit avec le jeu via des entrées textuelles. Elle repose sur une boucle principale qui affiche l'état actuel du jeu et permet aux joueurs de choisir une action (jouer une carte ou revendiquer une borne). La classe gère les entrées utilisateur en validant chaque choix pour éviter les erreurs.

Points clés

— **Affichage de l'état du jeu** : La méthode `afficherEtatJeu()` présente les bornes, les cartes posées, et la main du joueur courant.
— **Interaction utilisateur** : `jouerTourHumain()` permet au joueur de choisir une carte à jouer ou une borne à revendiquer.
— **Validation des entrées** : La méthode `lireEntier()` assure que les choix des utilisateurs sont valides en vérifiant qu'ils se situent dans les limites autorisées.

4.5.2 SchottenTottenGUI.java

Description : La classe `SchottenTottenGUI` fournit une interface graphique immersive utilisant JavaFX. Elle gère toutes les interactions visuelles du joueur avec le jeu, en mettant l'accent sur l'intuitivité et l'esthétique.

Principe utilisé

L'interface graphique repose sur des composants visuels organisés de manière hiérarchique. Chaque partie du jeu (main du joueur, bornes, interface de contrôle) est représentée par des conteneurs distincts pour une organisation claire et modulaire. La classe utilise des animations et des effets visuels (comme des transitions et des ombrages) pour améliorer l'expérience utilisateur. L'architecture événementielle de JavaFX est exploitée pour gérer les actions du joueur, comme cliquer sur une carte ou choisir une borne.

Caractéristiques principales

- **Écran d'accueil** : Présente un menu interactif où le joueur peut configurer la partie (choix de la variante, des joueurs, et du niveau de l'IA).
- **Gestion des bornes** :
  - Les bornes sont représentées graphiquement et mettent à jour leur état en temps réel en fonction des actions des joueurs.
  - Des animations indiquent lorsqu'une borne est gagnée.
- **Main des joueurs** : Les cartes sont affichées sous forme de composants interactifs, permettant au joueur de choisir une carte à jouer en cliquant dessus.
- **Barre de statut** : Fournit des informations sur le joueur courant et l'état du jeu, comme les bornes revendiquées ou le tour de l'IA.

Réactivité au tour de l'IA

Si le joueur courant est une IA, une transition est ajoutée pour simuler une réflexion avant de jouer. Cela améliore l'immersion en évitant que l'IA ne joue immédiatement.

Validation des entrées

Des boîtes de dialogue sont utilisées pour alerter le joueur en cas d'erreur (comme une tentative de jouer une carte invalide) ou pour confirmer des choix importants (comme abandonner la partie).

Illustration de l'interface graphique

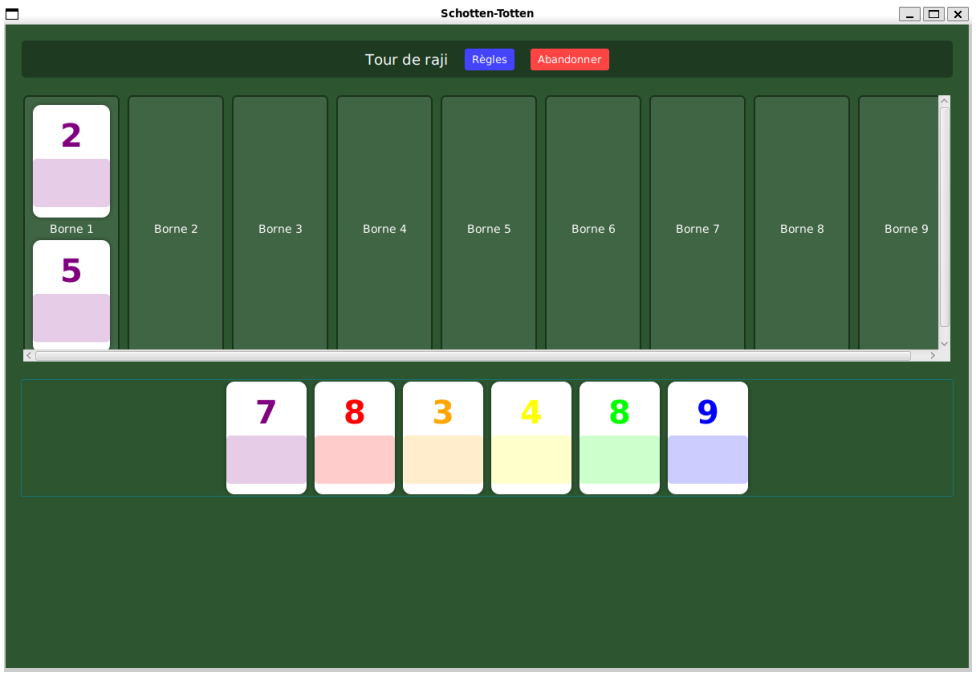

FIGURE 1 – Interface graphique de `SchottenTottenGUI` montrant une partie en cours.

Conclusion

Le package `view` propose deux interfaces distinctes pour répondre à différents besoins : une interface console simple et rapide pour les tests ou les environnements sans interface graphique, et une interface graphique immersive avec JavaFX pour une expérience utilisateur riche. Ces deux classes démontrent une utilisation efficace des principes de conception orientés utilisateur et garantissent une interaction fluide avec les composants du jeu.

5 Tests et Validation

5.1 Stratégie de tests

La validation du projet Schotten-Totten repose sur une stratégie rigoureuse combinant **tests unitaires** et **validation visuelle** à travers des scénarios de jeu. Les tests unitaires ont été réalisés avec JUnit pour vérifier la robustesse des fonctionnalités principales, tandis que des tests d'intégration ont permis de valider l'interaction entre les composants du jeu.

- **Outils utilisés :**
  - JUnit pour les tests unitaires.
  - Interface graphique et console pour les tests fonctionnels et visuels.
- **Approche adoptée :**
  - Tester les cas normaux, limites, et exceptionnels pour chaque fonctionnalité.
  - Simuler des parties complètes pour valider la logique de jeu.

5.2 Cas de test pour chaque fonctionnalité

Création de cartes :

- **Cas testé :** Création de cartes avec des valeurs et couleurs valides.
- **Résultat attendu :** Les cartes sont créées sans erreur.

Revendication des bornes :

- **Cas testé :** Revendiquer une borne complète avec une combinaison valide. Essayer de revendiquer une borne déjà revendiquée.
- **Résultat attendu :**
  - La revendication réussit si les règles sont respectées.
  - Une erreur est levée si la borne est déjà revendiquée.

Fin de partie :

- **Cas testé :** Vérification des conditions de victoire (5 bornes ou 3 bornes adjacentes). Gestion des égalités.
- **Résultat attendu :** Le gagnant est correctement identifié ou une égalité est signalée.

Interaction utilisateur :

- **Cas testé :** Jouer une carte via l'interface console ou graphique. Valider que les entrées utilisateur sont gérées correctement.
- **Résultat attendu :** Les actions sont exécutées et validées sans erreur.

Tests spécifiques :

- **IA :** Simulation de tours joués par l'IA en différents niveaux (facile, moyen, difficile). Validation de la logique de choix des cartes.
- **Cas limites :** Tentative de jouer une carte invalide. Revendiquer une borne avant qu'elle ne soit complète.

5.3 Résultats des tests

Les tests ont confirmé la robustesse du jeu, tant dans ses composants individuels que dans son intégration globale. Voici quelques résultats clés illustrés par des captures d'écran :

Écran d'accueil :

L'écran permet de configurer une partie avec choix de variantes, mode de jeu (Joueur vs Joueur ou Joueur vs IA), et niveau de l'IA.

FIGURE 2 – Configuration pour une partie normale.

FIGURE 3 – Configuration pour une partie contre l'IA avec niveau "Moyen".

Début de partie :

L'interface graphique initialise les bornes et les cartes des joueurs.

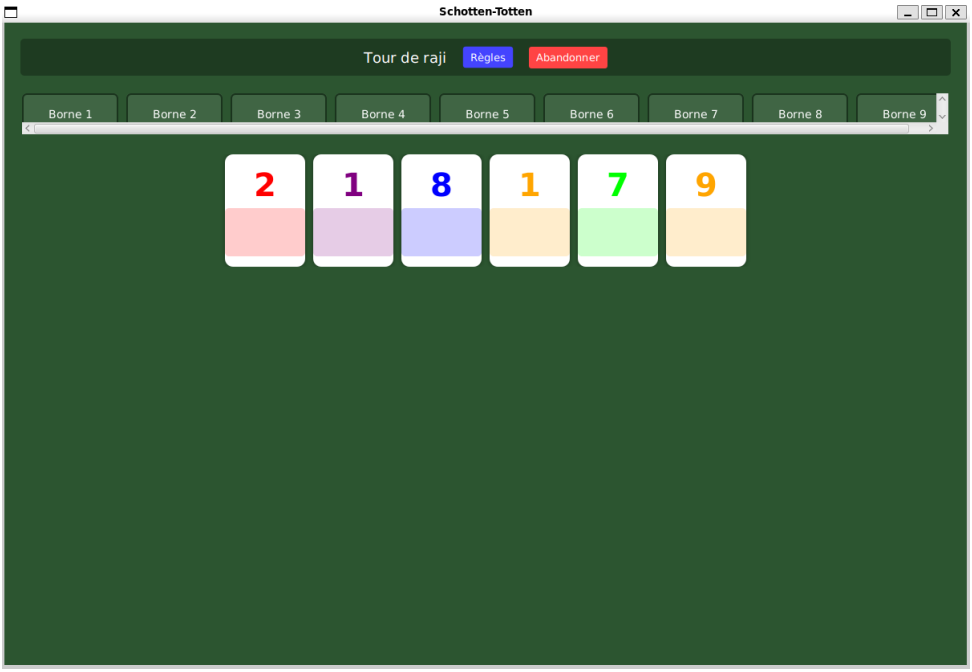

FIGURE 4 – État initial d'une partie.

Tour en cours :

Le joueur peut sélectionner une carte pour la jouer sur une borne disponible. Les bornes non complètes restent interactives.

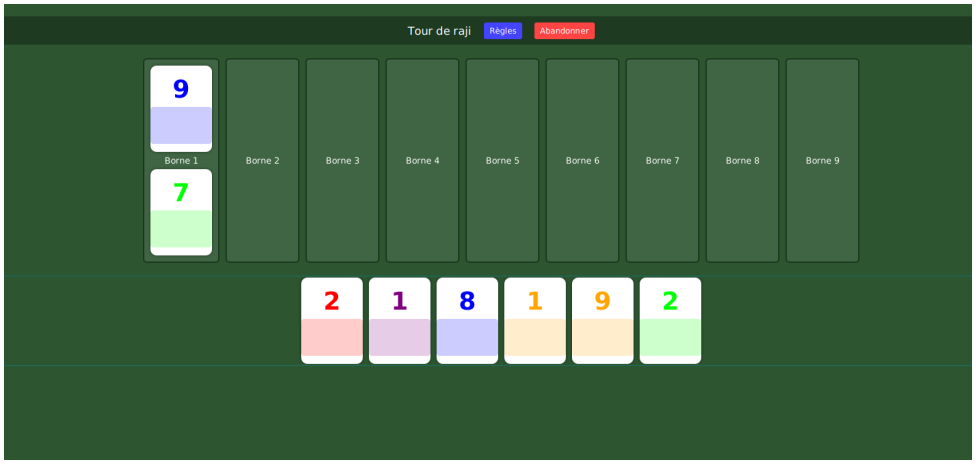

FIGURE 5 – Exemple d'un tour en cours où le joueur sélectionne une carte.

Revendication de bornes et progression :

Les bornes sont progressivement revendiquées par les joueurs ou l'IA, avec des mises à jour visuelles en temps réel.

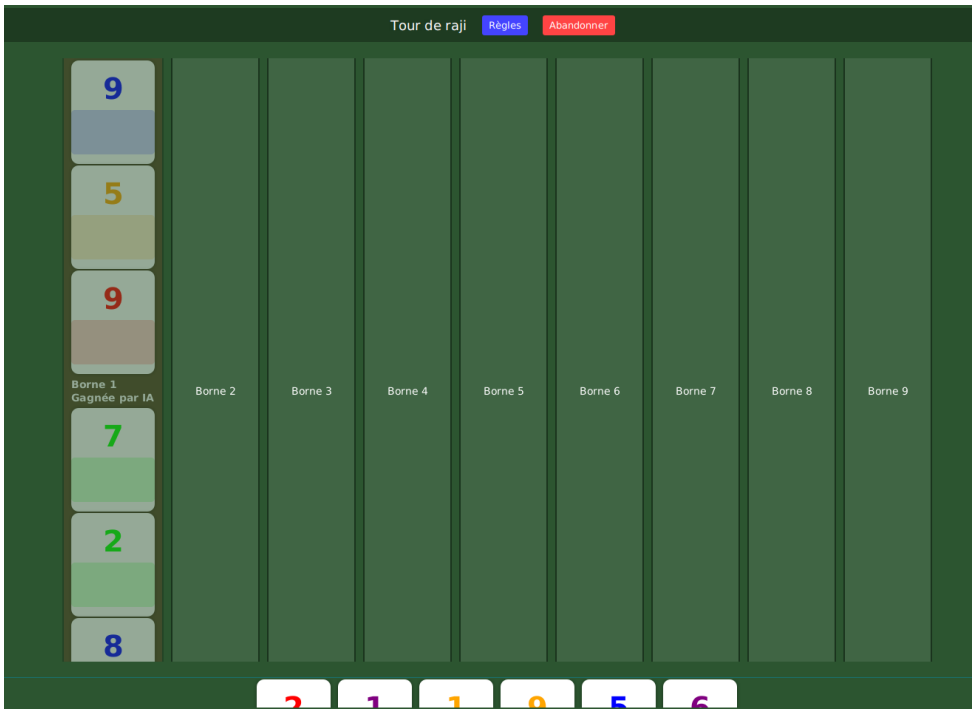

FIGURE 6 – Illustration d'une borne revendiquée.

Fin de partie :

— **Image 7 :** La partie se termine car le joueur a gagné **3 bornes successives**, remplissant une des conditions de victoire.
— **Image 8 :** La partie se termine car le joueur a revendiqué **5 bornes au total**, une autre condition de victoire.

FIGURE 7 – Fin de partie avec victoire par 3 bornes successives.

FIGURE 8 – Fin de partie avec victoire par contrôle de 5 bornes.

tabularx

5.4 Tableau récapitulatif des tests

Fonctionnalité	**Cas testé**	**Résultat attendu**	**Résultat obtenu**
Création de cartes	Carte valide	Création réussie	Succès
Interaction utilisateur	Jouer une carte sur une borne	Carte jouée	Succès
Revendication des bornes	Borne complète	Revendication réussie	Succès
Revendication invalide	Borne déjà revendiquée	Erreur levée	Succès
Fin de partie	5 bornes contrôlées	Victoire signalée	Succès
Fin de partie	3 bornes successives	Victoire signalée	Succès
Fin de partie	Égalité	Match nul signalé	Succès

TABLE 1 – Synthèse des résultats des tests.

5.5 Analyse et conclusion

Les tests ont validé toutes les fonctionnalités critiques, avec des résultats conformes aux attentes dans les cas normaux et limites. Les captures d'écran renforcent ces validations en illustrant les différentes étapes d'une partie.

- **Points forts :**
  - Interface intuitive et réactive.
  - IA fonctionnelle avec niveaux de difficulté adaptés.
  - Règles du jeu correctement implémentées.
- **Améliorations possibles :**

— Ajouter davantage de tests pour des scénarios complexes (exemple : plusieurs cartes tactiques jouées successivement).
— Optimiser la gestion des erreurs pour rendre les messages plus détaillés.

Le jeu est globalement stable et prêt pour une utilisation complète.

6 Résultats

6.1 Fonctionnalités implémentées

Le projet **Schotten-Totten** a réussi à implémenter les fonctionnalités principales du jeu en respectant les exigences initiales. Voici une liste des fonctionnalités qui ont été correctement mises en œuvre :

Variantes fonctionnelles :

— **Mode normal :** Implémentation des règles classiques du jeu.
— **Mode tactique :** Utilisation de stratégies spécifiques à l'IA pour optimiser ses décisions.
— **Mode expert :** Difficulté accrue avec des stratégies avancées.

Gestion des joueurs :

— Support pour les parties Joueur contre Joueur.
— IA fonctionnelle avec différents niveaux de difficulté (facile, moyen, difficile).

Interface utilisateur :

— Interface graphique intuitive et immersive avec JavaFX.
— Affichage des cartes, des bornes et des actions en temps réel.
— Interface console fonctionnelle pour une utilisation simplifiée ou dans des environnements non graphiques.

Gestion de la partie :

— Revendication des bornes selon les règles établies.
— Conditions de victoire respectées :
  — **3 bornes successives** gagnées.
  — **5 bornes au total** revendiquées.

Intelligence artificielle :

— Stratégies adaptées au niveau de difficulté choisi, garantissant un défi progressif pour le joueur.

6.2 Limites

Malgré les succès obtenus, certaines fonctionnalités et aspects du jeu n'ont pas été complètement implémentés ou pourraient être améliorés :

Cartes tactiques :

— Les cartes tactiques, bien que non représentées graphiquement, sont prises en compte dans la logique du jeu lorsqu'on joue via l'interface console.
— Elles influencent les stratégies et les décisions prises par les joueurs en fonction de leur type et de leurs effets spécifiques.

Optimisation des performances :

— L'IA pourrait être optimisée davantage pour des parties plus longues ou complexes.

Gestion des erreurs utilisateur :

— Les messages d'erreur dans l'interface console et graphique pourraient être plus descriptifs pour une meilleure expérience utilisateur.

Esthétique de l'interface graphique :

— Bien que fonctionnelle, l'interface graphique pourrait être améliorée pour inclure des animations et des transitions fluides.

7 Conclusion

Le projet **Schotten-Totten** illustre la capacité à transformer un jeu de société populaire en une application informatique complète, tout en respectant les principes fondamentaux de la programmation orientée objet. Ce travail ne se limite pas à une simple implémentation technique, mais démontre également une réflexion approfondie sur l'organisation, la modularité et l'expérience utilisateur.

La réalisation du projet a permis de relever plusieurs défis, notamment l'intégration d'une intelligence artificielle capable d'adapter ses stratégies selon les variantes et les niveaux de difficulté, ainsi que la gestion de différentes interfaces utilisateur (console et graphique). Ce projet a également offert une opportunité de mettre en pratique des concepts avancés de développement, tels que les design patterns, tout en renforçant les compétences en tests et validation.

Cependant, le projet se distingue également par les questions qu'il soulève pour l'avenir. Les choix techniques effectués ont permis de répondre aux besoins fondamentaux, mais laissent entrevoir un potentiel d'amélioration et d'enrichissement. Ces réflexions invitent à envisager le projet comme une première étape dans une démarche itérative, où chaque version future pourrait apporter de nouvelles fonctionnalités, une meilleure optimisation, et une expérience utilisateur toujours plus immersive.

En somme, **Schotten-Totten** dépasse son rôle de simple application ludique pour devenir un exercice de synthèse et un terrain d'expérimentation dans le domaine de la conception et du développement logiciel. Ce projet est une démonstration claire que même un jeu de société peut devenir un vecteur d'apprentissage riche et stimulant.