Dames Chinoises

Un jeu de dames chinoises, écrit en Java 8.

Louis Bal dit Sollier, Clara Bringer, Antonin Décimo, Pierre Gervais

Compiler & exécuter

Ce projet utilise Gradle comme moteur de production. Vous pouvez compiler le projet avec la commande `./gradlew` sous un UN*X ou avec `gradlew.bat` sous Windows.

Les dépendences de Gradle pouvant prendre un certain temps à télécharger, vous pouvez simplement lancer `make` pour compiler et exécuter le jeu.

Avec Gradle

```
./gradlew --daemon run # compile & exécute le jeu
./gradlew --daemon build # compile, vérifie, teste et distribue le jeu
./gradlew --daemon javadoc # génère la javadoc dans build/docs/javadoc
```

Gradle utilises le répertoire `build` pour stocker les classes compilées, la documentation et les exécutables.

Exécuter

Vous pouvez créer un exécutable avec `./gradlew --daemon installDist` qui sera disponible dans `build/install/Chinese_checkers`.

```
Chinese_checkers
├── bin
│   ├── ChineseCheckers        # Bash script
│   └── ChineseCheckers.bat    # Batch script
└── lib
    └── ChineseCheckers-0.0.1.jar
```

```
Usage: ChineseCheckers [-h] [--help] [--version]

Options:
   --version             Show ChineseCheckers version and exit.
   -h, --help            Show this help message and exit.

   -s <port>,            Launch the game in standalone server mode.
      --server <port>    Default port is 25565.
```

Fonctionnalités

Introduction

Nous avons développé un jeu complet de dames chinoises en réseau. Un seul exécutable est disponible, mais il peut être lancé en mode client avec une interface normale, dans lequel le client peut lancer un serveur pour héberger ses parties ou en mode serveur seul avec l'option `--server <port>`, afin que l'on puisse par exemple le faire tourner sur un serveur dédié.

Dans la suite du rapport, à la présentation de chaque fonctionnalité nous indiquerons quelle classe en est responsable.

Arborescence

Nous avons voulu organiser notre projet dans le style Java. Toutes nos classes se trouvent dans le package `org.copinf.cc`.
Nous avons utilisé le design pattern Modèle-Vue-Contrôleur. Les classes correspondantes sont respectivement situées dans les packages `org.copinf.cc.model`, `org.copinf.cc.view`, `org.copinf.cc.controller`.
Le dernier package `org.copinf.cc.net` contient toutes les classes relatives au réseau.

La classe `Main` est la classe d'entrée du programme. Elle va s'occuper de gérer les options de la ligne de commande. Si le jeu est lancé en mode client, elle crée un `MainController`.

Modèle

Toutes les classes du modèle du jeu se trouvent dans le package `org.copinf.cc.model`. La plus importante est certainnement la classe `Game` qui représente une partie, avec ses joueurs (classe `Player`), les équipes (`Team`) et le plateau.

La classe `AbstractBoard` est censée pouvoir définir un type abstrait de tableau auquel `DefaultBoard` se conforte. Dans les faits, nous avons abandonné l'idée de proposer d'autres types de plateaux, et notre code est devenu dépendant au plateau par défaut.

Le plateau utilise également les classes `Coordinates` pour représenter les coordonnées des cases `Square` sur lesquels se trouvent les pions `Pawn`.
Le plateau est représenté en interne en trois dimensions par une `Map<Coordinates, Square>`.

La vérification de la validité d'un mouvement s'effectue dans la méthode `checkMove` de la classe `AbstractBoard`.
Notre plateau est capable de s'adapter à un nombre variable de joueurs (de 2 à 6), d'équipes et de zones contrôlées par les joueurs. Sa taille peut également varier.

Le modèle est complété par deux autres classes, `Movement` qui représente le mouvement d'un pion à l'aide d'une liste, et `PathFinding` qui effectue la suggestion de mouvements.

Client

Le client utilise le design pattern MVC. A chacun des contrôleurs correspond une fenêtre du jeu et un sous-package de `view` associé. La classe `MainController` est grossièrement une pile de contrôleurs pour pouvoir facilement passer d'une fenêtre à une autre.

Home

La première fenêtre est gérée par le `HomeController`. Elle propose d'héberger ou de rejoindre un serveur, d'en sélectionner l'adresse et le port. La classe correspondante se trouve dans `view.homepanel`.

Lobby

Le lobby est la fenêtre d'où le joueur peut créer ou rejoindre une partie. Le contrôleur est `LobbyController` et le sous-package `view.lobbypanel`. Le panneau principal est `LobbyPanel`.

Le joueur doit débuter par envoyer un pseudonyme s'il veut pouvoir créer ou rejoindre une partie (classe `UsernamePanel`).

La liste des parties n'ayant pas encore un nombre suffisant de joueurs est affichée à gauche. Le joueur peut cliquer sur une des cases dont le rendu est effectué par `GameInfoRenderer` pour sélectionner une partie, puis cliquer sur le bouton "Join" pour la rejoindre.

A gauche se trouve un panel d'options pour pouvoir créer et configurer une nouvelle partie, géré par la classe `GameCreationPanel`.

Waiting room

Contrôleur : `WaitingRoomController`. Vue : `view.waitingroompanel`.
Tant qu'une partie n'a pas atteint un nombre suffisant de joueurs, ceux qui l'ont déjà rejointe peuvent échanger des messages (classe `ChatPanel`).
Si la partie se déroule par équipe, alors les joueurs peuvent également choisir leur coéquipier à cet instant (classe `TeamBuildingPanel`).

Game

Le jeu en lui-même est géré par la classe `GameController`. Dans le package `view.gamepanel` correspondant, la classe `GamePanel`est le panel principal.

- `InfoBar` s'occupe de la barre d'informations en haut de la fenêtre,
- `ActionZone` de la zone en bas avec la zone d'entrée du texte et les boutons,
- `DrawZone` gère la zone principale de dessin, à l'intérieur de laquelle :
  - le texte des messages du chat est affiché,
  - `BoardView` gère l'affichage et les événements du plateau de jeu,
  - `DisplayManager` offre les primitives de dessin du plateau.

A noter que le plateau est automatiquement disposé de manière à ce que le joueur trouve ses

pions en bas. Sont donc supportées la rotation, la mise à l'échelle (en fonction de la taille du plateau) et la distortion que nous aurions pu utiliser pour appliquer des effets au plateau.

Serveur

Toutes les classes du serveur se trouvent dans le package `net.server`.

Le thread principal `Server` dispose des listes des parties et des clients. Il s'occupe du lobby et gère le login des utilisateurs, la création de nouvelles parties et leur suppression.
Un second thread `ServerAcceptThread` accepte les connexions réseau et crée des nouveaux threads `ClientThread` pour chaque socket client qu'il ajoute dans la liste des threads clients du thread principal.

Lorsqu'un client crée une partie, un thread de partie `GameThread` est créé. Il est ajouté à la liste de threads de parties du serveur. Ce thread de partie contient la liste de threads des joueurs.
Lorsqu'un client demande à rejoindre une partie, on ajoute son thread à la liste de threads clients de la partie en question.

Le serveur utilise également quelques classes pour le réseau :

- `GameInfo` pour transmettre les caractériques de la partie,
- `Message` pour transmettre un message dans un chat,
- `Request` pour transmettre une n'importe quelle action dans le réseau. `Request` est à la base de notre protocole.

Vous pouvez retrouver la description de notre protocole dans le fichier `Protocole.md`.

Autres remarques

Nous avons écrit quelques tests unitaires avec JUnit. Gradle relance les tests à chaque compilation pour vérifier qu'ils n'échouent pas et tester la régression.
L'utilisation de plugins comme Checkstyle nous a permis d'adopter un style de code standard et de s'y conformer. D'autres plugins comme FindBugs et PMD facilitent le déboguage.
Nous avons également utilisé le service d'intégration continue Travis-CI.