Projet Java : Basic Card Game

TP1 - Projet Java

Développement d'un framework de Jeu

Jérôme Buisine
Florian Leprêtre

jerome.buisine@univ-littoral.fr
florian.leprêtre@univ-littoral.fr

29 mars 2021

Durée : 9 heures

La finalité de la suite des TPs proposés, est de concevoir un système permettant à des utiliseurs de jouer à des jeux cartes en réseau. Les premiers jeux de cartes développés seront le jeu de Bataille classique ainsi qu'une version simplifiée du Poker.

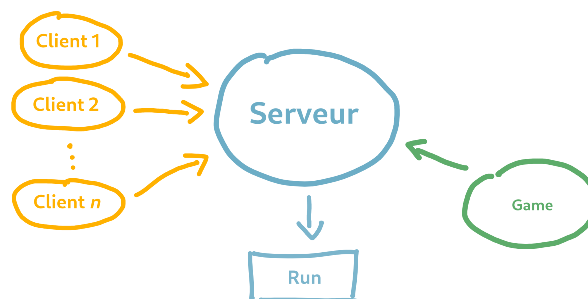
1 Travail

L'objectif de ce TP est d'initialiser les travaux relatifs à la création du framework de jeu de cartes envisagé. La mise en place d'une telle architecture de projet doit se faire dans un ordre bien défini et être bien réfléchie. Il vous sera demandé de bien suivre les différentes étapes du TP **dans le même ordre que présentées**.

1.1 Objectifs

L'objectif global est de mettre en place un système de jeu où chaque joueur peut se connecter à un serveur de jeu. Un serveur de jeu sera donc lié à un jeu qu'il héberge. Une fois le nombre de joueurs connectés attendu pour le jeu, le jeu peut démarrer.

Voici un petit schéma récapitulatif de l'attendu final :



L'objectif de ce TP est de développer un premier jeu de cartes tout en proposant une généricité du framework. Il s'agit du jeu de Bataille, où plusieurs joueurs s'affrontent avec le même nombre de cartes (si cela est possible) d'un paquet de 52 cartes. L'ensemble des règles du jeu seront détaillées et adaptées à notre contexte.

1.2 Environnement de travail

Voici l'ensemble des outils / langages qui seront utilisés pour pour l'ensemble des TPs :

— 1. IntelliJ/Idea (version Ultimate) comme environnement de développement (ou Eclipse en fonction de votre préférence) ;

— 2. Le langage Java avec la version 8 de Java SE ;

— 3. Git comme système de versionning du projet ;

— 4. Gitlab comme serveur d'hébergement de votre projet **Git**.

2 Initialisation du projet

Étant donné que nous allons utiliser la plateforme d'hébergement Gitlab, nous allons procéder à l'initialisation de celui-ci.

2.1 Récupération de projet

Une structure de projet a été initialisée et vous est proposée sur Gitlab. Elle sera votre base de développement pour l'ensemble des TPs. Il vous faudra créer un fork du projet et cloner votre propre projet pour le charger avec votre éditeur Intellij/Idea (`XXXXX` étant votre nom d'utilisateur Gitlab) :

```
git clone https://gitlab.com/XXXXX/l2_projet_java.git
```

Ce projet comprend plusieurs éléments :

— Un fichier d'aide pour l'utilisation de **Git** ;

— Un fichier README.md contenant un ensemble d'indications utiles aux TPs, qui sera mis à jour à chaque correction intermédiaire ;

— Un dossier `resources/games` qui comprendra au fur et à mesure les fichiers nécessaires aux jeux développés ;

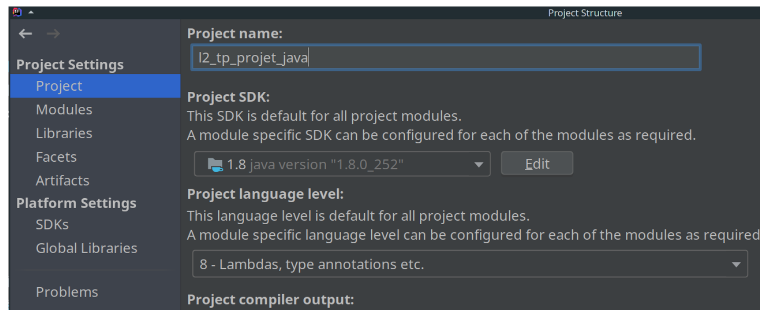
— La base du code source donné disponible dans le package Java `src/ulco/cardGame` ;

— Les diagrammes UML de chaque début de TP tout comme celui de l'attendu seront disponibles dans le dossier `resources/uml`.

2.2 Configuration du SDK

Il vous sera peut-être nécessaire de télécharger un SDK 1.8 directement ou via l'interface de configuration de structure du projet **`File > Settings > SDKs`**.

Afin de configurer votre projet pour Java 8, il vous faut ensuite spécifier votre SDK de la manière suivante via **`File > Settings > Project`** :



3 Développement du jeu

Nous allons maintenant développer le premier jeu de notre framework de jeu spécialisé dans un premier temps, jeu de cartes.

3.1 Régles du jeu

Voici les différentes étapes du déroulement du jeu de Bataille :

— L'ensemble des 52 cartes sont mélangées aléatoirement ;

— Chaque carte du jeu est distribuée une à une à chacun des joueurs ;

— Les joueurs vont s'affronter lors d'un ***round*** en jouant la première carte de leur main ;

— Le joueur avec la carte la plus haute remporte le ***round*** et récupère sa carte ainsi que toutes celles de ses adversaires. Ces nouvelles cartes sont stockées dans le bas de sa pile ;

— Le jeu consiste à réaliser un certain nombre de ***rounds*** jusqu'à ce qu'il y ait un joueur vainqueur. Le **gagnant** est celui ayant récolté l'ensemble des 52 cartes. Le jeu s'arrête donc puisque les autres joueurs ne sont plus en capacité de jouer ;

— Si un joueur ne possède plus de cartes en main, alors il ne peut plus jouer ;

— En cas d'**égalité** lors d'un round, une bataille peut-être entreprise. Dans le cadre de nos développements, nous désignerons simplement **aléatoirement** le gagnant ;

— On définira dans le cadre de notre jeu, que **tous les dix rounds** chaque joueur mélange sa main ;

— On considère ici que le **score** associé au joueur est son nombre de cartes en main. Il faudra bien faire attention à mettre à jour ce score lorsque le joueur joue ou gagne des cartes ;

3.2 Initialisation du jeu

Nous allons initialiser le jeu de cartes de la Bataille. Pour cela, il nous faut créer une classe représentative d'un tel jeu.

3.2.1 L'abstraction des jeux de plateaux

Afin de déjà voir les choses de manière générique, nous allons premièrement créer une classe **abstraite** dans le package `ulco.cardGame.common.games`, que l'on nommera `BoardGame` qui implémente l'interface `ulco.cardGame.common.interfaces.Game` proposée.

Cette classe `BoardGame` représentera de manière abstraite l'ensemble des jeux possibles basés sur une notion de *plateau*. On considère donc qu'un jeu de cartes fait également référence à un jeu de plateau.

La notion d'interface est ici très importante, puisqu'elle défini un **contrat** que chaque classe de jeu devra respecter. L'interface `Game` proposée, définit donc un ensemble de méthodes qui seront utiles à tous les types de jeux envisagées.

Dans cette nouvelle classe `BoardGame`, nous allons définir quelques méthodes proposées par l'interface. Étant donné que c'est une classe abstraite, toutes ne sont pas nécessairement obligées d'être implémentées mais uniquement celles que l'on considère comme communes à ce type de jeu. Nous définirons également quelques attributs liés à un état et à la gestion d'un jeude plateau.

Les attributs `protected` de la classe `BoardGame` :

— `String` **name** : permet de stocker un nom associé au jeu ;

— `Integer` **maxPlayers** : permet de savoir le nombre maximum attendu de joueurs pour le jeu ;

— `List<Player>` **players** : stocke la liste des joueurs. Le type `Player`, fait référence à l'interface `ulco.cardGame.common.interfaces.Player` disponible. Cette interface tout comme `Game`, définit un contrat de comportement, mais ici spécifique à un joueur ;

— `boolean` **endGame** : garde un état du jeu afin de savoir s'il est fini ou non (si au moins deux joueurs peuvent encore jouer) ;

— `boolean` **started** : permet de stocker un état du jeu afin de savoir s'il peut commencer ou non (nombre de joueurs attendu atteint).

Les méthodes de la classe `BoardGame` :

— `public` **BoardGame(**`String` **name,** `Integer` **maxPlayers,** `String` **filename)** : Constructeur de la classe initialisant l'état du jeu. Ce constructeur va également faire appel à la méthode `initialize` permettant de charger un jeu à partir d'un fichier. Cette méthode est encore non implémentée mais le sera spécifiquement à chaque jeu héritant de la classe `BoardGame` ;

— `public boolean` **addPlayer(**`Player` **player)** : méthode qui permet d'ajouter un joueur à la liste des joueurs. Cette méthode va vérifier la capacité d'accueil des joueurs. Si le nombre maximum est atteint, alors la partie peut commencer. De plus, il n'est pas possible d'ajouter un joueur, si un autre joueur du même nom est présent ;

— `public void` **removePlayer(**`Player` **player)** : méthode qui va supprimer un joueur de la liste des joueurs ;

— `public void` **removePlayers()** : méthode qui va supprimer l'ensemble des joueurs présents dans le jeu ;

— `public void` **displayState()** : méthode permettant d'afficher l'état d'un jeu. Ici, nous afficherons l'ensemble des joueurs présents ;

— `public boolean` **isStarted()** : méthode qui va spécifier si le jeu est commencé ou non (ou peut commencer) ;

— `public Integer` **maxNumberOfPlayers()** : retourne le nombre maximum de joueurs attendu ;

— `public List<Player>` **getPlayers()** : permet de récupérer la liste des joueurs du jeu en cours ;

Note : vous pouvez également vous réferer au diagramme de classes disponible dans votre projet afin de visualiser l'attendu du TP1.

Travail : À partir des indications fournies ci-dessus sur les attributs et méthodes attendues, développer la classe `BoardGame`.

3.2.2 Vers un jeu specifique de cartes

La classe `BoardGame` maintenant développée, nous pouvons définir une nouvelle classe, héritant des comportements de `BoardGame`. Nous appelerons cette classe `CardGame`, faisant référence au jeu de Bataille. Cette classe sera stockée au même endroit que la classe `BoardGame`.

Étant donné que nous allons développer un jeu de cartes, il nous faut créer une classe représentant une carte dans un jeu. Pour cela, une classe abstraite `Component`, a été initialement développée et est disponible dans le package `ulco.cardGame.common.games.components`. Elle permet la représentation de composants associés à un jeu. Un composant à un nom, une valeur de jeu (entière) et peut être lié ou non à un joueur (main du joueur par exemple). À noter qu'elle intègre un identifiant unique à chaque nouveau composant créé.

Travail : On définira donc pour notre jeu, le composant `Card`, qui héritera de la classe `Component` et qui sera stocké dans le même package. Ce composant prendra en compte un attribut supplémentaire, `hidden`, qui permettra de dire si la carte est retournée ou non (face visible ou non). Ce champs sera notamment utile lors de l'affichage de cartes ou non via une interface graphique.

Grâce à la définition de composants de type `Card`, nous pouvons définir les fonctionnalités de notre jeu. La classe `CardGame` sera composée des attributs suivants (propres à la classe, soit privés) :

— **`List<Card>` cards** : une liste de cartes, de type `Card`. Cette liste sera liée à l'ensemble du jeu de cartes ;

— **`Integer` numberOfRounds** : un compteur permettant de savoir le nombre de rounds réalisés. Il sera utile pour savoir à quels moments les joueurs doivent à nouveau mélanger leurs mains ou non.

Mais également les méthodes suivantes :

— **`public` CardGame(`String` name, `Integer` maxPlayers, `String` filename)** : Constructeur de la classe faisant appel au constructeur de la classe mère `BoardGame` ;

— **`public void` initialize(`String` filename)** : méthode qui charge le contenu du fichier de jeu et initialise le jeu de cartes en conséquence (la liste des cartes) ;

— **`public Player` run()** : méthode associée à la boucle de jeu ;

— **`public boolean` end()** : méthode permettant de vérifier si l'on est dans un état de fin de jeu ou non. Cette méthode va vérifier s'il y a un gagnant ou non ;

— **`public String` toString()** : affiche des informations relatives au jeu. On se limitera ici, au nom du jeu ;

Note : Le jeu de cartes proposé est disponible dans le fichier `resources/games/cardGame.txt`. Il précise pour chacune des 52 lignes, le nom associé à une carte ainsi que sa valeur. La représentation des noms de cartes est réalisée de la manière suivante avec un nommage anglais :

Il vous faudra donc lire ce fichier, y extraire les informations et créer une nouvelle carte en conséquence pour chaque ligne. Par défaut, on considère que toutes les cartes sont dans un état caché (cette fonctionnalité n'est pas importante pour le moment).

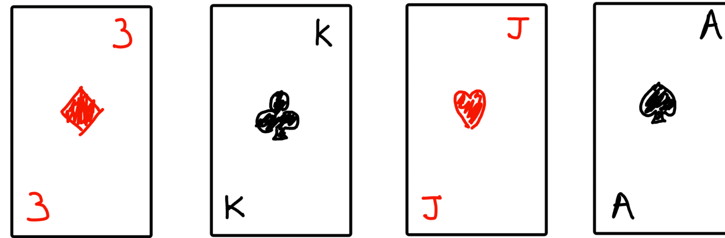
Voici un exemple de lecture de fichier en Java :

```
try {
  File cardFile = new File(filename);
  Scanner myReader = new Scanner(cardFile);

  while (myReader.hasNextLine()) {

        String data = myReader.nextLine();

        // get Card values and create Card
        // TODO
  }
```



FR	Carré	Trèfle	Coeur	Pic
EN	Diamond	Club	Heart	Spade
Name	D3	CK	HJ	SA
Value	3	13	11	14

```java
  myReader.close();
} catch (FileNotFoundException e) {
  System.out.println("An error occurred.");
  e.printStackTrace();
}
```

Travail : À partir des indications fournies ci-dessus, développer le comportement souhaité de la classe `CardGame`. **Attention**, on implémentera pas pour le moment la méthode `run`, qui elle gère la boucle de jeu. Par défaut, elle retournera la valeur `null` (aucun vainqueur).

Il vous est maintenant normalement possible, d'instancier votre jeu dans la classe `GameServer` (classe principale) du package `ulco.cardGame.sever`. Voici un exemple d'instanciation :

```java
// Instanciation of the Card Game with 3 players and using specific game file
// Stored as Game instance
Game game = new CardGame("Battle", 3, "resources/games/cardGame.txt");
```

3.3 Gestion des joueurs

Nous avons maintenant notre jeu spécifique partiellement implémenté ! Malheureusement, nous n'avons pas de joueurs disponibles dans ce jeu... Comme pour la création de jeu via l'interface `Game`, une interface `Player` vous a été fournie. Elle définit le comportement attendu de tous types de joueurs.

3.3.1 L'abstraction des joueurs de plateaux

Nous allons définir dans un premier temps, comme cela avait été entrepris avec la classe `BoardGame`, une classe abstraite `BoardPlayer` qui implémente l'interface `Player`, qui va définir les comportements globaux des joueurs de jeux de plateaux.

Les attributs globaux sont :

— `String` **name** : le pseudo associé au joueur ;

— `boolean` **playing** : un statut du joueur permettant de savoir s'il joue (associé à un jeu), peut jouer, ou ne peut plus jouer ;

— `Integer` **score** : score associé au joueur (ce score sera calculé en fonction du jeu visé).

Les méthodes génériques sont :

— `public` **BoardPlayer(**`String` **name)** : Constructeur permettant d'instancier un joueur via son nom. Il faut également définir son score à 0, tout comme son statut de jeu. Comme étant pour le moment non associé à jeu, donc ne jouant pas ;

— `public void` **canPlay(**`boolean` **playing)** : spécifie si le joueur peut jouer ou ne peut plus jouer ;

— `public String` **getName()** : permet de récupérer le nom de l'utilisateur (accesseur) ;

— `public boolean` **isPlaying()** : permet de récupérer le statut de jeu de l'utilisateur (accesseur) ;

— `public String` **toString()** : affiche les informations du joueur telles que son nom et son score. On pourrait également y afficher son statut de jeu ;

Travail : À partir des informations ci-dessus, développer les comportements génériques des joueurs de la classe `BoardPlayer`.

3.3.2 Création de joueurs de cartes

La classe `BoardPlayer` maintenant développée, nous allons définir une nouvelle classe `CardPlayer` qui déterminera les derniers comportements demandés par l'interface `Player` (notion de contrat) que l'on spécifiera pour le jeu de cartes de la Bataille.

Nous allons ajouter l'attribut suivant :

— `List<Card>` **cards** : qui permettra de stocker les cartes en main du joueur.

Puis définir les comportements manquants suivants :

— `public` **CardPlayer(**`String` **name)** : Constructeur d'un joueur de carte, qui va faire appel au constructeur parent et initialiser la liste des cartes ;

— `public Integer` **getScore()** : permet de récupérer le score du joueur ;

— `public void` **addComponent(**`Component` **component)** : permet l'ajout une carte dans la main de l'utilisateur. Le score (nombre de cartes en main) est donc à mettre à jour ;

— `public void` **removeComponent(**`Component` **component)** : suppression d'une carte de la main de l'utilisateur. Le score est également à mettre à jour ;

— `public Card` **play()** : le joueur joue la première carte de sa main. Cette carte est donc supprimée de sa main ;

— `public List<Component>` **getComponents()** : récupère l'ensemble des cartes de la main de l'utilisateur ;

— `public void` **shuffleHand()** : permet de mélanger la main de l'utilisateur ;

— `public void` **clearHand()** : supprime l'ensemble des cartes de la main de l'utilisateur ;

— `public String` **toString()** : affiche plus spécifiquement (si besoin), les informations du joueur.

Travail : À partir des indications fournies ci-dessus, développer le comportement souhaité de la classe `CardPlayer`.

Il vous est maintenant normalement possible, d'instancier un ou plusieurs `CardPlayer` dans votre programme principal `GameServer` et de les ajouter à votre jeu. En voici un exemple :

```java
// Instanciation of the Card Game with 3 players and using specific game file
// Stored as Game instance
Game game = new CardGame("Battle", 3, "resources/games/cardGame.txt");

Player player1 = new CardPlayer("user1");
Player player2 = new CardPlayer("user2");

game.addPlayer(player1);
game.addPlayer(player2);

game.displayState();
```

L'affichage de l'état du jeu devrait vous fournir une sortie équivalente à la suivante :

```
-------------------------------------------
--------------- Game State ----------------
-------------------------------------------
CardPlayer{name='user1', score=0}
CardPlayer{name='user2', score=0}
-------------------------------------------
```

3.4 Boucle de jeu

La création de jeu de cartes de Bataille, tout comme les joueurs associés est maintenant possible. Une chose très importante reste à développer, la boucle de jeu. Cette boucle de jeu, fait dans notre cas référence à la méthode **Player player run()** de la classe **CardGame**.

Travail : Au sein de cette méthode, et en reprenant les règles du jeu (voir partie 3.1), développer le comportement attendu de l'intéraction entre les joueurs. En ajoutant ensuite le nombre attendu de joueurs, simuler le jeu de Bataille dans votre classe **GameServer** et afficher le gagnant.

Conseil : Il vous sera peut-être utile d'exploiter des structures dites de dictionaire, **Map**. Elles permettent de stocker un type d'objet en tant que clé, et de lui associer une valeur, qui elle aussi peut être un type d'objet spécifique.

Il est possible d'associer pour chaque joueur, la carte qu'il a pu jouer, en voici un exemple :

```java
// Store player and associate his played card
Map<Player, Card> playedCard = new HashMap<>();

for (Player player : players) {

        // Get played card by current player
        Card card = (Card) player.play();

        // Keep knowledge of card played
        playedCard.put(player, card);

        System.out.println(player.getName() + " has played " + card.getName());
}

// for each card played
for(Map.Entry<Player, Card> entry : playedCard.entrySet()){

  // get player (key of entry map)
  Player player = entry.getKey();

  // get played card of player
  // Same as: Card card =  playedCard.get(player);
```

```java
  Card card = entry.getValue();

  System.out.println(player.getName() + " has played " + card.getName());
}
```

4 Bonus

Pour les plus téméraires d'entre vous, où le jeu de la Bataille est fonctionnel, vous pouvez développer le jeu complet de la Bataille dont les instructions sont les suivantes :

- Créer une nouvelle classe **`BattleGame`**, qui héritera des comportements de la classe **`CardGame`**. Seule la méthode `run` devra être réimplémentée ;
- Cette nouvelle classe va gérer l'égalité autrement qu'aléatoirement lors d'une manche. Voici les indications de la gestion envisagée :
  - Si des joueurs sont en égalité (carte de même hauteur et la plus haute de la manche), il faut alors procéder à la Bataille classique. Une carte est alors placée (prochaine carte de la main d'un joueur), face cachée sur la carte déjà jouée pour chacun des joueurs en lice dans la Bataille ;
  - Une autre carte est de nouveau jouée (prochaine carte de la main d'un joueur également) face visible par chacun des joueurs et c'est elle qui déterminera si oui ou non un gagnant sort vainqueur de la Bataille. Il est possible qu'une nouvelle égalité soit présente entre certains joueurs, il faudra alors réitérer l'action, jusqu'à ce qu'un joueur sort vainqueur de cette terrible Bataille ;
  - À la fin d'une Bataille, le joueur vainqueur récupère l'ensemble de toutes les cartes jouées présentes pour cette manche.
- Il est possible qu'un joueur soit en pénurie de cartes durant une telle Bataille. Il vous faudra gérer ce cas de la manière suivante :
  - Si un des joueurs n'a plus de cartes en main, son adversaire doit lui fournir les cartes manquantes après qu'il ait joué lui-même joué. C'est-à-dire que le joueur va prendre la ou les prochaines cartes de la main de son adversaire pour terminer la Bataille ;
  - Si plusieurs joueurs n'ont plus de cartes en main, alors la procédure reste la même, c'est l'adversaire qui a encore des cartes qui doit fournir ses adversaires lors de la Bataille ;
  - Si plusieurs joueurs peuvent fournir un adversaire n'ayant plus de cartes, le choix sera alors fait aléatoirement ;
  - Un autre cas peut être le fait qu'aucun joueur présent dans la Bataille ne puisse continuer la Bataille (par manque de cartes). Dans ce cas, c'est le joueur (par conséquent non en lice pour la victoire de la Bataille, car non présent lors de l'égalité) ayant le plus de cartes qui fournira l'ensemble des cartes nécessaires jusqu'à ce qu'un vainqueur soit déterminé par la Bataille.