



# SPLENDOR

LI Christine - LUC Clément - RAKOTOMAHEFA Fandresena  
L3 APP LSI2 – EFREI 2022-2023

# SOMMAIRE

## I/ INTRODUCTION

## II/ RÉALISATION DU PROGRAMME

- 1) Les classes
- 2) Les méthodes - quelques informations
- 3) Paramètres et améliorations

## III/ TESTS UNITAIRES

## IV/ CONCLUSION

## ANNEXE – COMPILEMENT – DIAGRAMME DE CLASSE



## INTRODUCTION

Pour ce projet de fin de semestre, il nous a été demandé d'implémenter un jeu : Splendor. Le principe du jeu de société est le suivant, acquérir un maximum de points de prestiges, et ce en achetant des cartes de développement ou en récupérant des tuiles de nobles. Celui qui parvient à avoir le plus de points gagne la partie.

Le jeu de Splendor est un jeu qui se joue de 2 à 4 joueurs.

En fonction du nombre de joueurs, les règles du jeu seront modifiées.

Nous allons donc voir comment se déroule la partie de jeu en fonction de ces différentes conditions.



## RÉALISATION DU PROGRAMME

### 1) Les classes

Nous avons décidé de créer différents objets. Pour commencer, on aura une classe pour chaque élément du jeu:

- ❖ Token.java : cette énumération permet de définir chaque couleur par un entier.
- ❖ DevCard.java : cette classe contient les paramètres d'une carte de développement dont :
  - le nombre de prestiges qui le définit (combien de point de prestiges le joueur va gagner en achetant une carte).
  - le niveau de la carte.
  - le bonus de la carte (par quelle pierre est définie la carte de développement, utile lors de la venue d'un noble).
  - L'illustration de la carte, chaque carte possède une illustration, parfois la même pour une série d'un même type de carte (ex : une mine).
  - un dictionnaire contenant les différentes ressources (en jetons et en bonus) afin d'acquérir cette carte.
- ❖ Tiles.java : cette classe permet de définir une tuile noble. Elle prend un prénom, un nombre de prestiges et un dictionnaire de ressources (comme pour les cartes).

- ❖ Player.java : cette classe permet de définir un joueur. Un joueur sera donc initialisé par :
  - un prénom.
  - un âge, afin de connaître le joueur qui commencera la partie, le plus jeune.
  - un dictionnaire contenant les différents jetons que possède le joueur.
  - une liste contenant les différentes cartes que le joueur a réussi à acquérir.
  - un dictionnaire contenant les différents bonus que possède le joueur. Utile lors de la venue d'un noble.

Pour ce qui est de l'implémentation et des différentes conditions de jeu (un jeu de 2 joueurs et un jeu de 4 joueurs n'aura pas les mêmes règles) la classes Game.java permet de respecter celles-ci.

- une liste contenant les cartes réservées par le joueur. Carte qu'il pourra acheter plus tard s'il le souhaite.
- une liste de tuile de nobles que le joueur a obtenu.

❖ Shell.java : cette classe permet de définir les couleurs de jetons. Notamment, permet un affichage en ligne de commande plus esthétique.

❖ Import.java : cette classe permet de gérer l'importation de fichier contenant les informations sur les cartes du jeu (les cartes de développement et les tuiles des nobles).

❖ Game.java : cette classe permet de gérer le jeu dans son ensemble, la gestion des tours de jeu, de la pile de cartes pour chaque niveau et ceux des jetons (vérifier qu'il y a assez de jetons pour pouvoir laisser au joueur de piocher dans la pile en question).

❖ Main.java : cette classe permet également la gestion du jeu. Elle va permettre de récupérer les valeurs que vont rentrer les joueurs afin de pouvoir déterminer combien il y a de joueurs et l'âge de ces derniers.

❖ Rules.java : cette classe permet de gérer tous les paramètres de jeu, c'est à dire, le nombre de joueur maximum, le mode de jeu choisi, le nombre de prestiges à avoir pour gagner une partie ...

## 2) Les méthodes - quelques informations

Afin de réaliser un programme de jeu optimal et clair, certaines méthodes d'implémentation ont été préféré à d'autres. Notamment, le choix des hashmap plutôt que les listes. En effet, on verra que cela nous a permis une meilleure manipulation, que ce soit pour la gestion des différents jetons ou des cartes de développement.

Comme énoncé précédemment, pour ce qui est des jetons ou des ressources des cartes de développement, nous avons préféré une implémentation avec des hashmap :

- les cartes de développement et les tuiles des nobles : les ressources des cartes, donc le nombre de jetons que la carte a besoin pour être acheté, ont été mise dans une hashmap. On sauvegardera le nombre de jetons pour chaque couleur de pierre. De cette façon, il va être plus simple de retirer et ajouter un ou plusieurs jetons lors de l'achat d'une carte.
- le joueur : les jetons des joueurs seront également réservés dans une hashmap, comme pour les ressources d'une carte. Les bonus que le joueur réussi à obtenir lors de l'acquisition des cartes de développement seront dans une LinkedHashMap. De cette façon, on va pouvoir acheter de nouvelles cartes à l'aide de bonus et de jetons. Néanmoins, il ne sera pas nécessaire de retirer des bonus. On souhaite seulement récupérer la valeur de chaque clé (le nombre de pierre de chaque couleur, puisque les bonus sont représentés par des pierres).

Pour ce qui est des jetons, on identifiera chaque jeton par une couleur (0 émeraude, 1 diamant, 2 saphir, 3 onyx, 4 rubis et 5 or) et une valeur, qui sera le nombre de jetons pour chaque pierre de couleur.

Afin d'acheter une carte de développement, il a fallu scindé la méthode permettant de faire cette action.  
En effet, on retrouve les méthodes suivante :

- `canBuyCardWithObject(Map<Token, Integer> cardresources, Map<Token, Integer> map)` : cette méthode prend deux Map en paramètre, une pour les jetons d'une carte (les ressources) et l'autre pour les jetons du joueurs. La méthode va donc faire la différence entre la quantité de jetons des deux Map différentes, la différence de chaque jetons sera mis dans une hashMap que la méthode renverra.

- `enoughToPay(Map<Token, Integer> map)` : cette méthode va permettre de vérifier si dans la map donné en argument il y a des jetons ou non.

- `leftToPay(DevCard card)` : cette méthode va appeler les deux méthodes précédentes et va retourner une map contenant le reste des ressources a payer après avoir regardé dans la liste des bonus du joueur. En effet, un joueur peut acheter une carte a l'aide de jetons mais également et en premier temps, du nombre de bonus que celui ci a. Donc si le joueur a assez de bonus pour acquérir la carte sans dépenser le moindre jetons, on revoie **null**. Sinon, s'il reste des jetons, il va falloir regarder dans les jetons du joueurs afin de voir s'il y a assez de jeton.

C'est donc à l'aide de ces méthodes que l'on va pouvoir gérer l'achat d'une carte.

### 3) Paramètre et amélioration

Pour ce qui est des améliorations, il sera possible que le joueur puisse voir au cours de la partie, et ce, à chaque tour de jeu, la liste de cartes et de jetons qu'il possède afin de mieux gérer ses éléments de jeu et d'ainsi avoir la meilleure stratégie pour gagner. Il lui suffira donc d'entrer les commandes suivantes :

- \* `counters` : permet d'afficher la liste de vos jetons
- \* `cards` : permet d'afficher les cartes de votre jeu

Afin de simplifier la compilation du programme de jeu, nous avons initialiser un Makefile. (cf. Annexe - Compilation)

# TESTS UNITAIRES

## \* Dans la classe Game.java :

Nous allons expliciter les fonctions d'actions du jeu.

Les fonctions permettant d'acheter des cartes de développement, de prendre des tokens ainsi que de réserver une carte de développement.

### ❖ actionSameCounters(Player player)

```
public boolean actionSameCounters(Player player) {  
    Objects.requireNonNull(player);  
  
    if (!verifyNumberCounters(player, defaultValue: 10)) {  
        System.out.println("> Vous avez plus de 10 jetons déjà !");  
        return false;  
    }  
    var userChoice = Import.inputStr(this.scan, Shell.LISTCOLORSSTRING, "Vous avez choisi de prendre 2 jetons de la " +  
        "même couleur : Veuillez indiquer la couleur : ");  
    if (this.countersStack.get(Token.values()[Shell.LISTCOLORSSTRING.indexOf(userChoice)]) < 3) {  
        System.out.println("Il n'y a pas assez de jetons dans la pile");  
        return false;  
    }  
    player.addToken(Token.values()[Shell.LISTCOLORSSTRING.indexOf(userChoice)], 2);  
    this.removeCountersFromGame(Token.values()[Shell.LISTCOLORSSTRING.indexOf(userChoice)], n: 2);  
    return true;  
}
```

Cette méthode permet de prendre deux jetons d'une même couleur. Elle renvoie un booléen (si le joueur peut prendre les tokens ou non)

En premier lieu, on vérifie que le joueur ne possède pas plus de 10 jetons dans son jeu, auquel cas, le joueur ne pourra pas prendre de tokens.

Ensuite, si le joueur possède bien un nombre de tokens inférieur à 10, il a donc la possibilité de choisir la couleur des deux tokens qui, rappelons le, est la même pour les deux tokens. Si ce n'est pas le cas, on revoit False.

Une fois la couleur du token choisi, on vérifie que dans la pile de jeu, qu'il y a au minimum 4 tokens de la couleur choisi par le joueur. Si ce n'est pas le cas, on revoit False.

Et enfin, si toutes les conditions précédentes ont été respectées, le joueur plus donc ajouter à son jeu, les deux tokens choisi et on retire les deux jetons choisis pas le joueur de la pile de jeu.

## ❖ actionDiffCounters(Player player, int nbCounters)

```
public boolean actionDiffCounters(Player player, int nbCounters) {
    Objects.requireNonNull(player);

    if (!verifyNumberCounters(player, defaultValue: 10)) {
        System.out.println("> Vous avez plus de 10 jetons déjà !");
        return false;
    }

    boolean noProblem = true;
    var temp = new ArrayList<Token>();
    for (int i = 0; i < nbCounters; i++) {
        var userChoice = Import.inputStr(this.scan, Shell.LISTCOLORSSTRING, "> Vous avez choisi de prendre "
            + (nbCounters - i) + " jetons différents : Veuillez indiquer la couleur: ");
        if (this.countersStack.get(Token.values()[Shell.LISTCOLORSSTRING.indexOf(userChoice)]) < 1 ||
            temp.contains(Token.values()[Shell.LISTCOLORSSTRING.indexOf(userChoice)])) {
            System.out.println("> Il n'y a pas assez de jetons dans la pile\n> Veuillez mettre 3 jetons différents");
            noProblem = false;
            break;
        }
        temp.add(Token.values()[Shell.LISTCOLORSSTRING.indexOf(userChoice)]);
    }
    if (noProblem) {
        temp.forEach(counter -> player.addToken(counter, 1));
        temp.forEach(counter -> this.removeCountersFromGame(counter, n: 1));
    }
    return noProblem;
}
```

Cette méthode permet de prendre trois jetons de couleurs différentes. Elle renvoie un booléen (si le joueur peut prendre les tokens ou non)

En premier lieu, on initialise un **drapeau** (booléen) qui va nous permettre de soulever un problème ou non lors des choix des tokens ainsi qu'une liste de tokens temporaire (**tmp**). Celle-ci va nous permettre de “retenir” les tokens choisis par le joueur.

On vérifie que le joueur ne possède pas plus de 10 jetons dans son jeu, auquel cas, le joueur ne pourra pas prendre de tokens.

Ensuite, si le joueur possède bien un nombre de tokens inférieur à 10, il a donc la possibilité de choisir les trois couleurs différentes des tokens.

Une fois que les couleurs de chaque tokens ont été choisi, on vérifie dans la pile de jeu l'existence de jetons des couleurs choisis par le joueur. Si ce n'est pas le cas, on met à False le **drapeau** qui nous permet de savoir s'il y a eu un problème lors de la récupération des tokens.

Et enfin, si toutes les conditions précédentes ont été respectés, on ajoute à **tmp** les tokens choisis par le joueur. On utilisera cette liste pour ajouter ces derniers dans la liste de jeu du joueur et les supprimer de la pile de jeu (plateau de jeu).

Dans le cas où le **drapeau** n'a pas été changé, donc à True, on ajoute au jeu du joueur les jetons qu'il a choisis et on les supprime du plateau de jeu.

### ❖ actionBuyCard(Player player)

```
public boolean actionBuyCard(Player player) {  
    Objects.requireNonNull(player);  
    if (this.actionBuyReserveCard(player)) {  
        return true;  
    }  
    System.out.println("> Dans les cartes Découvertes");  
    var userChoice = Import.inputInt(this.scan, "> Quelle Carte voulez vous acheter Niveau [1/2/3] :", 3);  
    int numberCard = Import.inputInt(this.scan, "Choisissez quelle carte de niveau " + userChoice +  
        " vous voulez", this.discoveryCards.get(userChoice).size());  
    var cardUser = this.discoveryCards.get(userChoice).get(numberCard - 1);  
    var payCard = player.leftToPay(cardUser);  
    if (payCard == null || !payCard.equals(Map.of(Token.values()[0], -1))) {  
        // la carte peut déjà être payé avec les bonus ou jetons  
        player.buyCard(cardUser);  
        this.discoveryCards.get(cardUser.getLevel()).remove(cardUser);  
        this.drawCardFromStack(cardUser.getLevel(), nbCardToExtract: 1);  
        if (payCard != null){  
            //la carte doit être payé avec un reste de jetons  
            player.removeAllTokenOfCard(payCard);  
            payCard.forEach((key, value) -> this.countersStack.put(key, value + this.countersStack.get(key)));  
        }  
        System.out.println("> Vous avez acheté la carte " + cardUser);  
        return true;  
    }  
    else {  
        // la carte ne peut pas être payé ni avec les bonus ni avec les jetons du joueur  
        System.out.println("> Vous n'avez pas assez de ressources pour acheter cette carte");  
        return false;  
    }  
}
```

Cette méthode permet d’acheter une carte parmi les cartes du plateau de jeu. Elle renvoie un booléen (si le joueur peut acheter des cartes)

En premier lieu, le joueur choisit le niveau de la carte à acheter ainsi que la carte du niveau choisi, qui est posé sur le plateau de jeu.

On retire la carte que le joueur souhaite acheter des cartes découvertes sur le plateau.

On regarde si le joueur peut récupérer la carte sans avoir à dépenser un seul jeton. Donc acquérir la carte à l'aide des bonus des cartes qu'il possède déjà.

Si le joueur ne possède pas assez de bonus, il doit compléter l'achat avec des jetons.

Ensuite, si le joueur possède bien un nombre de tokens ou de bonus suffisant pour acheter la carte, on revoit True. Sinon, on renvoie False

### ❖ actionBuyReserveCard(Player player)

```
public boolean actionBuyReserveCard(Player player) {  
    Objects.requireNonNull(player);  
  
    var userChoice = Import.inputStr(this.scan, List.of("oui", "non"), "Voulez vous acheter une carte réservé" +  
        List.of("oui", "non"));  
    if (userChoice.equals("non") || player.getReservedCards().size() == 0) {  
        System.out.println("> Quit");  
        return false;  
    }  
    int numberCard = Import.inputInt(this.scan, "Choisissez quel carte réservé vous voulez acheter [1-" +  
        player.getReservedCards().size() +"]", player.getReservedCards().size());  
    var card = player.getReservedCards().get(numberCard - 1);  
    var payCard = player.leftToPay(card);  
    if (payCard == null || !payCard.equals(Map.of(Token.values()[0], -1))) {  
        player.buyCard(card);  
        player.removeReservedCard(card);  
        if (payCard != null){  
            //la carte doit etre payé avec un reste de jetons  
            player.removeAllTokenOfCard(payCard);  
            payCard.forEach((key, value) -> this.countersStack.put(key, value + this.countersStack.get(key)));  
        }  
        return true;  
    }  
    else {  
        System.out.println("> Vous n'avez pas assez de ressources pour acheter cette carte");  
        return false;  
    }  
}
```

Cette méthode permet d'acheter une carte parmi les cartes réservées par le joueur tout au long de la partie. Elle renvoie un booléen (si le joueur peut acheter des cartes)

En premier temps, le joueur choisi s'il souhaite acheter un carte réservé. On revoit False s'il décide de ne pas en acheter ou alors qu'il n'a pas de cartes réservées.

Si le joueur décide d'acheter une carte réservée, il choisit laquelle et pareillement que pour la méthode précédente.

On regarde si le joueur peut récupérer la carte sans avoir à dépenser un seul jeton. Donc acquérir la carte à l'aide des bonus des cartes qu'il possède déjà.

Si le joueur ne possède pas assez de bonus, il doit compléter l'achat avec des jetons.

Ensuite, si le joueur possède bien un nombre de tokens ou de bonus suffisant pour acheter la carte, on revoit True. Sinon, on renvoie False

### ❖ actionReserveCard(Player player)

```
public boolean actionReserveCard(Player player) {
    Objects.requireNonNull(player);

    if (player.getReservedCards().size() >= 3) {
        System.out.println("> Trop de cartes réservés");
        return false;
    }
    DevCard cardUser;
    var userChoice = Import.inputStr(this.scan, List.of("stack", "discovery"),
        "choisissez une carte dans la pioche ou des cartes visibles" + List.of("stack", "discovery"));
    var level = Import.inputInt(this.scan, "> Choisissez quel carte de niveau vous voulez : [1/2/3]", 4);
    if (userChoice.equals("discovery")) {
        var numberCard = Import.inputInt(this.scan, "Choisissez quel carte de niveau " + level +
            " vous voulez", this.discoveryCards.get(level).size());
        cardUser = this.discoveryCards.get(level).get(numberCard - 1);
        player.reserveCard(cardUser);
        this.discoveryCards.get(level).remove(cardUser);
        this.drawCardFromStack(cardUser.getLevel(), nbCardToExtract: 1);
    } else {
        cardUser = this.cardStack.get(level).get(0);
        player.reserveCard(cardUser);
        this.cardStack.get(level).remove(cardUser);
    }
    System.out.println("> Vous avez réservé la carte " + cardUser);
    return true;
}
```

Cette méthode permet de réserver des cartes de développement. Elle renvoie un booléen (si le joueur peut réserver des cartes)

En premier temps, on vérifie si le joueur a un nombre de cartes réservées dans son jeu inférieur à 3. Auquel cas, s'il en a plus de 3, il ne pourra pas réserver davantage.

Si le joueur décide de réserver une carte et qu'il le peut, il a le choix d'en choisir parmi les cartes découvertes sur le plateau de jeu ou en prendre une aléatoirement dans la pioche de jeu (qu'importe le niveau de la carte).

On regarde si le joueur peut récupérer la carte sans avoir à dépenser un seul jeton. Donc acquérir la carte à l'aide des bonus des cartes qu'il possède déjà.

Dans les deux cas, le joueur doit préciser le niveau de la carte à réserver.

Ainsi, si tout s'est bien déroulé, le joueur peut réserver la carte et on renvoie la fonction à True.

### ❖ playGame(String phaseChoise)

```
public void playGame(String phaseChoise) {  
  
    if (this.players.size() < this.nbPlayers) {  
        throw new IllegalAccessException("Missing Players to Play a Game");  
    }  
    initGame(this.rules.getNbLevel(), this.rules.getNbCardsByLevel(), this.rules.getNbTokens());  
    boolean playing = true;  
    while (playing) {  
        for (int i = 0; i < this.players.size(); i++) {  
            var playerActual = this.players.get(i);  
            if (this.victory(playerActual)) {playing = false;}  
            System.out.println(this);  
            System.out.println(playerActual);  
            this.playRound(playerActual);  
        }  
    }  
    winner();  
    System.out.println(this.classement(this.players));  
}
```

Cette méthode permet de gérer un tour de jeu.

Tout d'abord, on vérifie qu'il y a bien le nombre de joueurs annoncé au début de la partie. On initialise le jeu (les règles, le nombre de cartes par niveau de jeu, le nombre de tokens ...). On instancie une variable booléen, un drapeau, permettant d'annoncer le début du jeu.

Donc on lance le jeu et les tours (à chaque joueur, on lance le programme de jeu). On vérifie si le joueur à gagné, donc si le joueur a atteint un nombre de prestiges maximum.

# CONCLUSION

Ainsi, l'ensemble des fonctions écrites fonctionnent comme souhaité. Ce projet nous permet d'approfondir nos connaissances et d'en apprendre davantage sur la gestion du travail en groupe, ainsi que de revoir toutes les notions vues au cours de l'année.

Les difficultés rencontrées ont été lors de la gestion de l'achat d'une carte. L'algorithme et l'écriture des méthodes nous ont pris plus de temps que pour le reste.

Nous avons également pris un moment afin de réfléchir à l'implémentation des bonus, faire ou non une hashMap afin d'y contenir chaque jeton.

N'ayant pas trouvé le temps de finaliser ce projet, nous voulions synthétiser les fonctions (ceux des actions de jeu) ayant la même structure afin d'éviter une toute répétition de code.

Pour finir, malgré quelques difficultés, nous sommes satisfaits des fonctions réalisées puisque nous avons pu reconnaître et en apprendre plus sur l'importance et l'utilisation de chaque fonction au sein de notre programme.

# ANNEXE - COMPIRATION - DIAGRAMME DE CLASSE

## Instruction de compilation :

## Instruction de compilation :

L'utilisation d'un Makefile permet de simplifier la compilation des différents fichiers et donc du projet.

◆ La commande suivante permettra d'exécuter le programme à l'aide d'un Makefile :

**make**      Ou alors  
**make execute**

En effet, les deux commandes ci-dessus permettent de compiler et de lancer le programme (ont la même utilité)

\* `make execute` : permet de d'exécuter le programme depuis la classe Main.java du projet.

On peut retrouver plusieurs commandes afin de rendre la compilation plus fluide :

- \* `make create jar` : créer une archive de tous les fichiers .class
  - \* `make clean`: permet de supprimer les fichiers .class
  - \* `make mrproper`: permet de supprimer tous les fichiers .class
  - \* `make help` : affiche un manuel d'aide à l'utilisateur

## Plugin couleur Eclipse :

<https://stackoverflow.com/questions/6286701/an-eclipse-console-view-that-respects-ansi-color-codes>

Lien du GIT :

<https://github.com/Fandresena02/Splendor>

# ANNEXE - COMPILEMENT - DIAGRAMME DE CLASSE

## DIAGRAMME DE CLASSE

