ACDC-TheGame

Un arbitre, une IA et une IHM pour *The game*.

IMT - FIL1 - 2020

JALLAIS Adrien : adrien.jallais@protonmail.com

Partie 2 - Version 1.4

Introduction

Utilisation de l'application

Les fichiers sources sont accessibles dans le dossier suivant : Code/FIL A1 ACDC Partie2 Jallais Adrien.
Pour savoir comment lancer l'application, reportez-vous au fichier suivant : module-info.java.

Auteurs

La dimension *frontend* ou Interface Homme Machine (IHM) a été réalisée par Adrien Jallais. Cette IHM utilise une API réalisée par Nicolas Kirchhoffer, à laquelle Adrien Jallais a apporté quelques modifications.

Résultats

Aperçu

L'*Illustration 1-1* présente ce que la fenêtre cliente affiche au cours d'une partie en **mode Solo**.

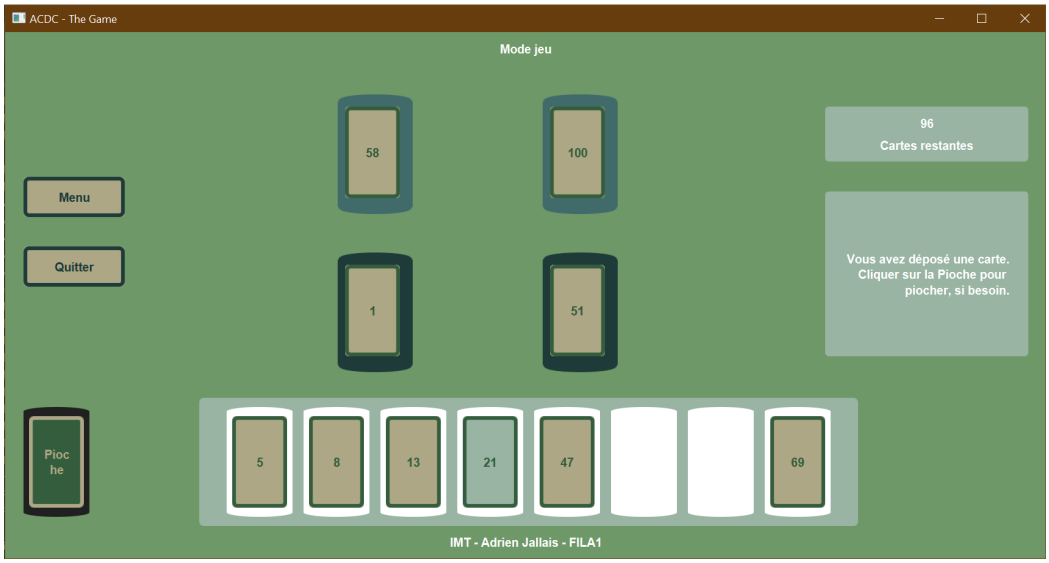

Illustration 1-1 : *Capture de la fenêtre au cours d'une partie en mode Solo.*

L'*Illustration 1-2* présente ce que la fenêtre cliente affiche au départ d'une partie en **mode Démonstration**.

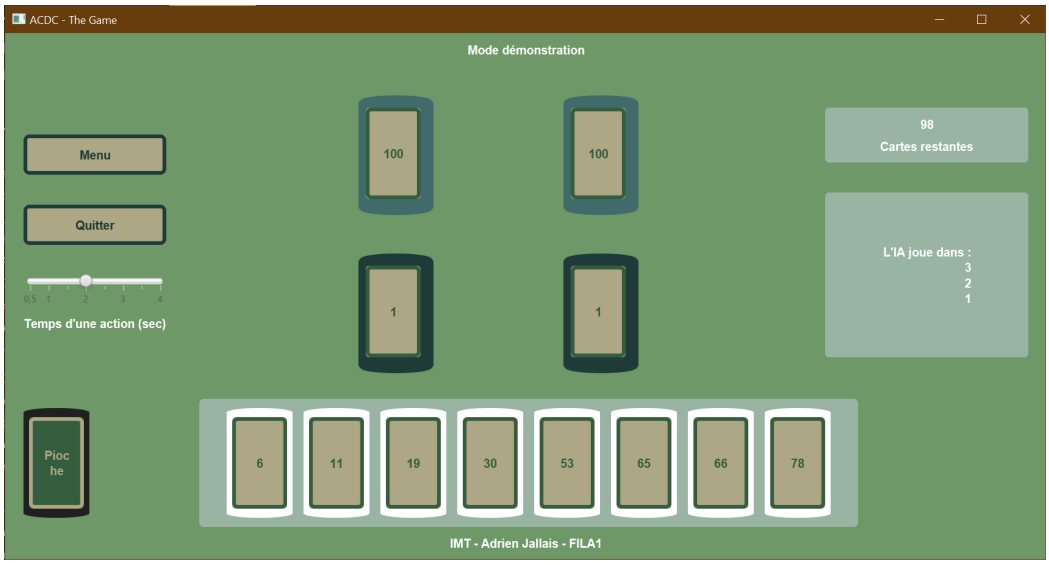

Illustration 1-2 : *Capture de la fenêtre au départ d'une partie en mode Démonstration.*

Progression et suivi du projet

Un fichier décrivant les logs réalisés quotidiennement est disponible dans le fichier suivant : log.Jallais.Adrien.json. En complément, le *Tableau 1* illustre ces logs pour mieux visualiser la cinétique de développement du projet.

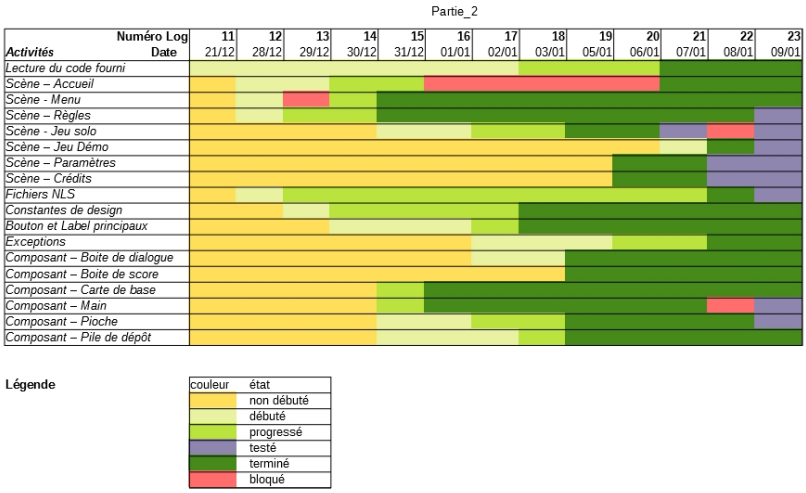

Tableau 1 : *Grille de progression du développement de l'application. Les logs représentent un jour de travail.*

Avec le *Tableau 1*, on observe que les premières scènes qui ont été mises en place sont celles qui demandaient le moins de complexité. En effet, dans le but de monter en compétence de manière graduelle avec la librairie JavaFX, les scènes d'accueil et du menu ont été réalisées en premier car elles comportaient une infrastructure simple : un à deux composants d'agencement impliqués, et des composants interactifs basiques (bouton) avec des actions similaires (changement de scène). La réalisation de ces scènes a permis également de poser les bases pour la mise en place d'un design homogène entre les différentes scènes de l'application. En effet, la mise en place d'un *wrapper* commun aux scènes est rapidement apparue nécessaire pour définir une structure commune, ainsi que la mise en place de constantes communes, notamment pour les couleurs utilisées et l'espacement entre les composants d'une scène, comme c'est le cas pour les boutons et les labels.

Diagramme de classe

L' *Illustration 2* est un diagramme de classe UML généré avec ObjectAid UML Explorer. Les relations entre ses entités étant ajoutées de manière automatique, il est rapidement devenu surchargé et illisible. Afin d'améliorer sa lisibilité, les caractéristiques suivantes ne sont pas montrées :

- les relations de dépendance entre les classes (au profit de celles entre les packages),
- les méthodes de visibilité publique des classes implémentant des interfaces (afin d'éviter une répétition entre ces deux entités),
- les méthodes des classes du package view.scene,
- les classes du package api.

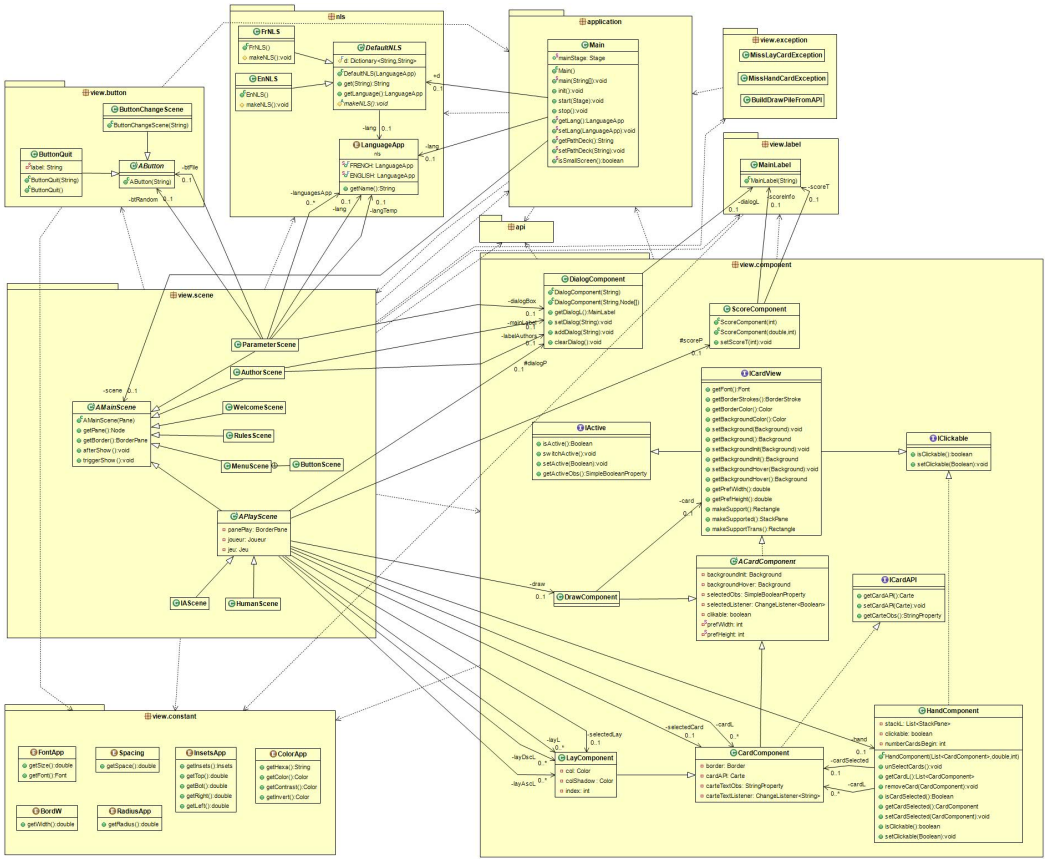

Illustration 2 : *Diagramme UML de classe de l'application.*

Les composants spécifique aux scènes de jeu sont rassemblés au sein du package view.component, afin d'optimiser leur réutilisation au sein des scènes de type *APlayScene*. Les attributs des classes concrètes, de types *ACardComponent* de ce package, sont affichés dans le but de soulever le lien entre ces classes et le l'API. On voit notamment que les classes *CardComponent* et *LayComponent* possèdent un attribut de type *Carte* ou de type *int* respectivement, qui permettent de savoir à quel élément du jeu ils font référence. De plus, un des attributs de *HandComponent* est de type *List*, dont chacun des éléments possède notamment un *CardComponent*, et donc par extension un attribut de type *Carte*. En opposition, *DrawComponent* ne possède pas d'attribut de jeu mais il est initialisé avec une variable de type *ICardView* qui permet de régir son style.

D'autres composants ne sont pas spécifiques aux scènes de jeu : view.button, view.label. Comme ces derniers n'ont ni la même finalité ni la même tendance à être modifié, ils n'ont pas été rassemblés dans le même package.

L'ensemble des scènes sont réunies au sein d'un même package : view.scene. On remarque qu'ils sont tous de type *AMainScene*, qui est un *BorderPane*, permettant ainsi de définir un masque de scène, utilisé notamment pour l'ajout d'une signature en bas de fenêtre. Les scènes de jeu (c'est-à-dire de type *APlayScene*), possèdent deux attributs : un de type *Jeu* et un de type *Joueur*, afin de permettre une interaction avec l'API. L'attribut *Joueur* est déterminé par les scènes concrètes l'implémentant (*IAScene* ou *HumanScene*), et l'attribut *Jeu* est également défini par le chemin du fichier constituant la pioche. La classe *APlayScene* sollicite l'attribut *Jeu* en utilisant ses méthodes afin de piocher, obtenir les cartes de la main du joueur actuel, connaître l'état du plateau et de son achèvement (victoire ou défaite).

Le réglage graphique des packages abordés ci dessus, est défini à partir de constantes énumérées au sein du package : view.constant. On y trouve notamment la police, les couleurs des composants de notre application.

Le package view.exception, rassemble les exceptions nécessaires au traitement de celles soulevées par l'API, pouvant survenir lors de la création d'une pile ou de l'action d'un tour. Il permet notamment de régler les messages associés, en fonction de la langue choisie par l'utilisateur.

En effet, deux dimensions linguistiques sont prises en charge par cette application. Le package nls (correspondant à l'abréviation : National Langage Support ou Soutien aux Langues Nationales) rassemble les éléments qui définissent le contenu des messages affichés à l'interface, afin de faciliter leur modification et leur traduction.

Discussions

Analyse du code fourni

Modifications apportées

On peut retrouver les commit associés aux modifications de l'API en recherchant les mots clés suivants : *feat(gameAPI)* ou *fix(gameAPI)*, correspondant respectivement à l'ajout d'une nouvelle fonctionnalité ou la résolution d'un bug.

Il a été rajoutées des méthodes, qui permettent les fonctionnalités suivantes :

- savoir si une partie est gagnée ou non avec `isVictoire` : en effet, je choisis d'affirmer que c'est à l'API et son arbitre de dire si la victoire est remportée ou non, et pas seulement d'afficher le score (lien) ;
- connaître le nombre maximum de cartes autorisé par joueur `getNbCartesMax` : en effet, pour initialiser une nouvelle main après avoir pioché et qu'il n'y a plus assez de cartes dans la pioche pour atteindre le nombre maximum de carte autorisé et l'IHM doit afficher cette information, je choisis d'affirmer que c'est à l'API et son arbitre de donner cette indication. J'ai pu utiliser du code existant pour écrire cette méthode (lien) ;
- vérifier la validité de la pioche fournie au format *.txt* : en effet, la méthode `fromFile` ne vérifie pas la présence de doublons ni le nombre de cartes produites (lien) ;
- générer une pioche non pas seulement à partir d'un fichier, mais également de manière aléatoire : afin de limiter mon impact sur l'API, je n'ai pas créé de nouvelle méthode mais seulement ajouté un cas à `fromFile`, qui renvoie une pioche générée aléatoirement lorsque le chemin indiqué est nul (lien).

Il a été apporté des modifications au code de l'API, afin de permettre la résolution des bugs suivants :

- la méthode `passerTour`, ne permettait pas d'incrémenter le nombre de tours si il n'y avait qu'un seul joueur (lien) ;
- dès lors que l'incrémentation des tours était de nouveau fonctionnelle, la méthode `jouer` levait l'exception : *"Ce n'est pas votre tour !"* (lien) ;
- la méthode `isPartieFinie`, ne prenait pas en compte le fait que le joueur actuel est autorisé à passer son tour et donc à piocher, s'il a posé un nombre de cartes suffisant (lien) ;
- les piles de dépôt étaient initiées, non pas à 1, mais à 0 (lien) ;

- lors de la fin d'un tour, le jeu était amené à piocher dans tous les cas, malgré que la pioche était vide (lien).

Point négatifs du code fourni

La méthode fournie `fromFile` pour générer une pioche est simple d'utilisation et évite de mettre en place un patron de conception complexe comme celui de la *Factory*. Cependant, comme on l'a mentionné plus haut, cette méthode n'était pas assez robuste. Par ailleurs, celle-ci ne remonte pas les exceptions (elle ne fait que les afficher), ainsi il n'est pas possible d'afficher à l'IHM un dialogue adapté pour conseiller l'utilisateur dans la soumission d'un nouveau fichier.

L'API pourrait utiliser des références envers instances de la classe Tas au sein de la liste `List<Tas>` plutot que d'utiliser l'indice associé.
En effet, pour certaines méthodes publiques, comme `jouer(int tasId, Carte carte, Joueur joueur)`, les arguments n'ont pas un degré de complexité équivalent (*int* est un type des plus basique alors que *Carte* est un objet). En effet, si l'on souhaitait un maximum de simplicité, on aurait pu choisir le prototype suivant : `jouer(int tasId, int valeurCarte, Joueur joueur)` ou bien un degré de rigueur plus élevé avec `jouer(Tas tas, Carte carte, Joueur joueur)`.
De plus, pour le constructeur de *CarteIA* il serait plus rigoureux de demander dans les arguments d'indiquer un *Tas* et pas seulement un indice.

Il a été réalisé une grande abstraction pour les tas en créant une interface mais cela n'a pas été fait pour les autres composants du jeu, comme celui du joueur.
De plus, il me semble que l'API pourrait utiliser des abstractions de plus haut niveau, en utilisant non pas les classes abstraites (*Tas*) mais plutôt les interfaces (*ITas*). En effet, la classe *Jeu* comporte un attribut de type `List<Tas>` et non de type `List<ITas>`.

La pioche et les tas de l'API contenant un attribut de type `List<Carte>`, il aurait pu être plus efficace, et aussi plus sûr d'utiliser un attribut de type `Deque<Carte>` (notamment pour la méthode piocher). En effet, avec la méthode `List<Carte> getCartes`, il est possible de parcourir l'ensemble des cartes de la pioche et donc d'en prendre connaissance afin d'adapter une possible stratégie.

Les *getteurs* des attributs de la classe *Jeu*, en particulier getTas et getPioche pourraient, pour plus de sécurité, renvoyer ces collections au sein de la méthode suivante : `Collections.unmodifiableCollection()`.

Points positifs du code fourni

Premièrement, la documentation des méthodes est exemplaire, ce qui permet une lecture du code de l'API plus facile, et permettra, à l'avenir, une plus grande maintenabilité du code.

De plus, le service IA de résolution d'un tour IA est très performant. Il est très pertinent d'avoir mis en place un héritage entre *JoueurIA* et *Joueur*, ce qui permettra à un humain seul de jouer avec une IA. Par ailleurs, l'héritage entre les classes *Carte* et *CarteIA* est également judicieux car il permet de définir et de garder en mémoire, beaucoup plus facilement, un poids associant une carte à un tas.

Par ailleurs, en tant que développeur *frontend*, a priori j'aurais préféré une méthode me donnant le meilleur couple carte-tas à jouer pour une configuration de jeu donné, afin de découper plus finement le tour réalisé par l'IA, ou bien pour conseiller un utilisateur novice lors de ses premières parties. Cependant pour la réalisation d'un coup de *recul*, qui est d'ailleurs l'élément essentiel pour battre le jeu, cette vision par tour est plus pertinente.

En outre, l'API utilise de manière pertinente les levées d'exceptions, pour la méthode `jouer` par exemple. Ainsi un affichage précis et adapté de celles-ci est donc possible à l'IHM.

Malgré des consignes indiquant le développement d'un jeu avec un seul joueur, le code fourni prend déjà en compte une possible extension vers un mode à plusieurs joueurs. En effet, le code met en place l'attribut suivant : `List<Joueur>`, ainsi que le dénombrement de tour et la possibilité de vérifier qu'il n'y a que le joueur dont c'est le tour de déposer des cartes.
Par ailleurs, cette prise en compte d'une possible évolution du jeu s'illustre aussi par le rassemblement des tas de dépôts de carte au sein d'une liste et non d'un tableau, ce qui rend possible la modification du nombre de tas de dépôts.

Les attributs de la classe *Jeu* sont accessibles via des *getteurs* (comme `getPioche`) ce qui permet de limiter le nombre de méthodes à mettre en place pour faire parvenir une information pertinente aux classes externes (comme `getPioche().size()`), et donc d'obtenir un code léger mais puissant pour autant.
En conclusion, de manière globale, la rédaction du code est légère et efficace, ce qui en permet une lecture agréable et rapide.

Choix réalisés

Une partie des choix qui ont été faits au cours du développement de cette IHM ont été expliqués dans la partie de ce rapport traitant des modifications apportées à l'API.

Par ailleurs, bien que l'API permette de jouer à plusieurs joueurs, il a été choisi de se focaliser sur le but premier du cahier des charges, qui était de fonctionner avec un seul joueur, afin de respecter les délais de livraison de l'application.

L'utilisation de feuille de style *.css* a été écartée car il n'est à ma connaissance pas possible de combiner l'utilisation de valeurs calculées avec ce format de fichier. Souhaitant utiliser des constantes de design récupérées à partir de classes d'énumération, j'ai décidé de ne pas baser le style de mon application sur des fichiers *.css*.

Bilan de l'application

Points faibles de l'application

La classe *APlayScene* propose un mode de fonctionnement plus chargé qu'il ne pourrait l'être. En effet, la scène *IAScene* n'a pas besoin des évènements déclenchés par la souris pour montrer comment jouer. Ainsi, l'instanciation de *IAScene* pourrait solliciter moins de mémoire, si la classe *APlayScene* ne posait que les fondements communs des scènes *IAScene* et *HumanScene*.

Malgré que l'application prenne en compte la taille de l'écran à son démarrage, notamment en modifiant l'affichage de l'image de fond du plateau de jeu, cette modification de l'affichage n'est pas dynamique.
Par ailleurs, les composants du plateau de jeu n'adaptent par leur taille à celle de l'écran, et leur agencement n'est pas non plus sensible à ce paramètre. On ne peut donc qualifier le design de l'application de *responsive*.
Cependant, comme la taille de ces composants est définie au sein d'une même classe (*ACardComponent*), le développement de la fonctionnalité qui répondra à ce cas d'utilisation en sera facilité.

Après avoir fait essayer cette IHM à des utilisateurs, il semble que la possibilité de déposer les cartes en mode *drag and drop* soit une fonctionnalité importante à mettre en place, car ce comportement semble naturel pour l'utilisateur.

Points forts de l'application

Le point fort essentiel de cette IHM est qu'elle présente les fonctionalités demandées par le cahier des charges : jouer une partie en mode Solo et observer une IA jouer.
De plus, l'ajout d'un slider sur la scène de l'IA permet de modifier la vitesse de démonstration de jeu de l'IA.

Par ailleurs, une fonctionnalité majeure a été réalisée en supplément, qui permet de modifier la langue d'affichage de l'application (en anglais ou en français).

Cette IHM est basée sur l'utilisation de composants réutilisables et de constantes de design qui permettent son homogénéité.

Au cours du développement des différentes scènes, il a été abordé une diversité importante des outils de JavaFX (par exemple : agencer des composants, régler l'exécution de l'application avec des timeurs, lire un fichier audio, évaluer les touches claviers sélectionnées par l'utilisateur). La réalisation de cette IHM m'a donc permis de monter en compétences sur différents outils proposés par le framework de JavaFX.