Rapport Méthode de Conception

Antonin Montagne, Lou-Anne Gautherie, Nathan Sakkriou,
Yanis Habarek

25 Novembre 2022

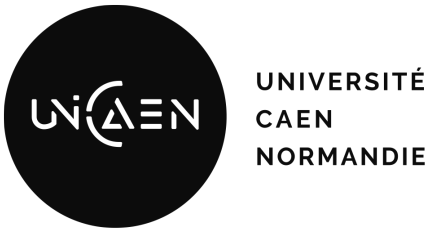


Table des matières

1 Introduction

1.1 Plan du rapport

Nous évoquerons d'abord quels étaient nos objectifs de départ (1.2), puis nous détaillerons les différentes étapes de la création de notre projet avec les rôles de chacuns (2). Ensuite nous présenterons l'architecture de ce projet (3), ainsi que les éléments techniques utilisées (4) dans notre code. Finalement nous terminerons par une courte conclusion (5).

1.2 Objectifs du projet

Nous avions pour but de créer un jeu de Blackjack. Certaines contraintes nous étaient données :

- Respecter la forme MVC.
- Créer une dépendance entre une librairie "`cartes`" et un jeu "`blackjack`".

2 Fonctionnalités implémentées

2.1 Description des fonctionnalités

A partir de l'accueil, il est d'abord possible de vérifier les règles du blackjack en cliquant sur le bouton "RÈGLES DU JEU".
Le clic du bouton renverra le joueur sur le site *guide-blackjack.com/regles-du-black-jack.html*, qui contient les règles de ce jeu.

Il est aussi proposé de lancer une partie à l'aide du bouton "NOUVELLE PARTIE".
Le clic du bouton ouvrira un formulaire qui demandera à l'utilisateur, combien de joueur veut-il rentrer pour cette partie ainsi que le style de cartes qu'il souhaitera. Une fois le nombre de joueur rentré, le jeu se lancera après avoir cliqué sur "COMMENCER". Le joueur commencera la partie, il peut soit tirer une carte, soit ne rien faire (boutons "HIT" et "STOP"). Si il choisit de ne rien faire, le tour passe au croupier. Le plus proche de 21 sans dépasser gagne la partie. Une fois la partie terminée, une fenêtre pop-up avec le nom de celui qui a gagné et un bouton "OK" qui renverra le joueur sur la page d'accueil.

Depuis l'accueil, il est finalement possible de fermer la fenêtre en appuyant sur "CLIQUER".

2.2 Organisation du projet

Pour commencer, Nathan a codé toute la librairie *cartes.jar*.
Il a d'abord définit ce qui est une carte dans la classe Card. Une carte est définit par une `value` et une `color`. Chaque carte est également définit par une image qui est récupérée par la `getImgPath`. Il y a deux styles différents de cartes : *classique* et *vintage*. Il a codé plusieurs fonctions de comparaisons comme la fonction `compareTo` qui compare les valeurs d'objets, et la fonction `equals` qui vérifie si deux objets sont égaux.

Ensuite il a codé la classe *Deck* qui définit un paquet de cartes. Un deck est représenté par une `ArrayList<Card>`. Cette classe contient une fonction `shuffleDeck` pour mélanger les cartes du deck, deux fonctions `addCardToEnd` et `addCardToHead` qui permettent respectivement d'ajouter une carte à la fin ou au début du paquet. Il y a également un fonction `drawCard` qui représente une carte piochée ; elle est enlevée du deck et retourne la carte piochée.

Ensuite il a codé une main dans la classe *Hand* ; une main est définit par une `ArrayList<Card>` et un `Deck`. Elle possède des fonctions pour :

- tirer une carte de la main (`getCard`),
- retourner les cartes de la main afin de voir leurs valeurs ou non (`changeShowHand`),
- ajouter ou retirer une carte de la main (`addCard` et `removeCard`),
- réinitialiser la main (`resetHand`),
- jouer une carte (`playCard`),
- et une dernière fonction de comparaison (`compareTo`)

Finalement il a créer une classe *DeckFactory*, qui génère des *Deck* prédéfinis. Elle contient deux fonctions qui initialisent des deck :

1. `get52DeckCard` ; un deck de 52 cartes (as à roi),
2. `get40DeckCard` ; un deck de 40 cartes (as à 10).

Dans le package *Blackjack* se trouve la classe *Party* codée par Nathan. Cette classe représente simplement une partie de Blackjack. Les éléments de cette

classe sont :

- un `Deck deck` qui est le jeu de cartes,
- une `ArrayList<Hand> listHand` qui est la liste de toutes les mains de tous les joueurs,
- une `ArrayList<Integer> listSumHand` qui est la liste des sommes des mains de tous les joueurs,
- une `Hand dealerHand` qui est la main du dealer,
- un `Integer sumDealerHand` qui est la somme de la main du dealer,
- un `Integer NB_CARD_INITIAL_DRAW` qui est le nombre de cartes initial d'une main au début de partie (initialisé ici à 2),
- un `Integer AS_VALUE` qui est la valeur que l'as peut prendre (soit 1, soit 11),
- et un `HashMap<String, Integer> RELATION_NUMBER_VALUE` qui est un dictionnaire contenant les relations entre les valeurs des cartes.

Plusieurs méthodes y sont définies :

- `generateRELATION_NUMBER_VALUE`, génère dans la `HashMap RELATION_NUMBER_VALUE` la relation entre les "values" d'une carte et leurs valeurs numériques au blackjack.
- `addHand` ajoute une main dans la partie,
- `initialDraw` représente le tirage initial de 2 cartes par personne et d'une carte pour le croupier,
- `getSumHandValue` permet d'insérer dans les attributs correspondant la somme des mains des joueurs et du croupier,
- `getSumHandValueOfOneHand` permet d'obtenir la valeur de la main donnée,
- `getSumDealerHand` calcule la somme des cartes du croupier,
- `handChoice` définit le choix du joueur lorsque c'est son tour (tirer une carte -> return true, ou ne rien faire -> return false),
- `askChoiceConsole` demande au joueur le choix qu'il veut faire (méthode valable uniquement en console), s'il veut tirer, il doit tapper 1, s'il veut ne rien faire, il doit tapper 2.
- `allHandTurn` permet de faire jouer tous les utilisateurs chacun leurs tour (méthode valable uniquement en console),
- `dealerTurn` permet de jouer le tour du dealer,
- `AIPlayerTurn` permet de définir la manière de jouer des intelligences artificielles,
- `checkWhoWin` vérifie les conditions de victoire du croupier et des joueurs ;

si la somme des cartes du croupier vaut plus de 21, tous les autres joueurs gagnent, si un joueur est plus près de 21 que les autres, il gagne, et si un joueur dépasse 21, il perd.

— `completeTurn` permet de jouer un tour complet ; les joueurs jouent, le croupier joue, et on vérifie qui a gagné à la fin du tour.

— et `launchGame` permet de lancer une partie (méthode valable uniquement en console) ; les cartes sont distribuées à l'aide de la méthode `initialDraw`, les sommes des mains sont obtenues grâce à `getSumHandValue` et `getSumDealerHand`.

Passons maintenant à la vue, codée par Yannis, Antonin et Lou-Anne. En premier lieu, dans le package *View* se trouve la classe *BasicPage* codée par Yannis, qui est la fenêtre de base utilisée dans toutes les autres classes de *View*. Elle contient des balises HTML et du style CSS, mais aussi les images de fond durant tout le lancement du jeu. 3 images y sont définies :

1. l'image de fond lors du lancement à l'aide de la méthode `setBackgroundLaunch`,
2. l'image de fond de la page d'accueil à l'aide de la méthode `setBackgroundHome`,
3. et l'image de fond du jeu à l'aide de la méthode `setBackgroundGame`.

Ensuite la classe *Home* codée également par Yannis, est extends de `BasicPage` (comme toutes les autres classes), et définit la page d'accueil. Cette classe possède un controller nommé *ControlHome*. Ce controller définit l'image de fond de l'accueil, et les évènements liés aux bouttons présent sur l'accueil (les boutons sont "règles du jeu", "nouvelle partie", et "quitter"). La classe *Home* définit les emplacements et apparences des boutons qu'elle contient.

Antonin et Lou-Anne se sont occupés de la classe *GamePage* et de son controller *ControlGamePage*. Le controller possède plusieurs méthodes d'affichage :

— `printOneHand` permet d'afficher une main grâce à des coordonnées données, chaque carte est un boutton qui possède une image,

— `printAllHandAndBackGround` affiche toutes les mains (dont celle du dealer),

— `actionHit` définit l'action de la pression du bouton "hit" ; le joueur tire une carte, celle-ci s'ajoute dans sa main,

— `actionStand` définit l'action de la pression du bouton "stand" ; le joueur arrête de piocher, et c'est au tour des autres joueurs de jouer,

puis celui du dealer, et enfin la partie se termine. Un pop-up apparait avec les résultats et les options "Rejouer" ou "Quitter",
- `actionGetHandValue` affiche la valeur d'une main grâce à des coordonnées données,
- `actionPrintPictureDealer` affiche l'icone du dealer à coté de sa main,
- et enfin `cleanPanel` nettoie tous les composants du panel sauf les boutons "Hit" et "Stand".

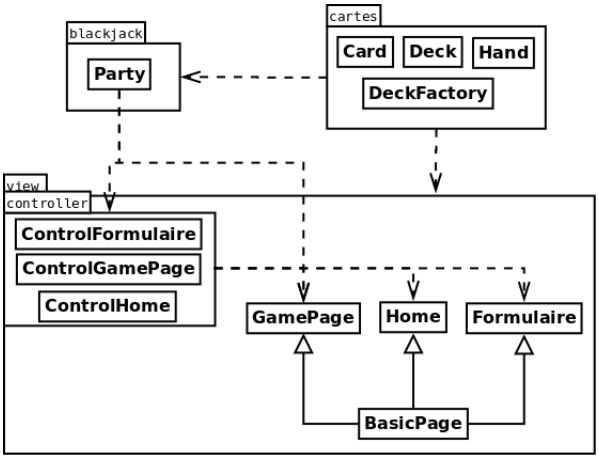
La classe *GamePage* créée les boutons "Hit" et "Stand" en leur associant une image, et elle définit les coordonnées de ces boutons. Elle appelle ensuite les fonctions de son controller.

Yannis s'est finalement occupé du formulaire de début de jeu. La classe *Formulaire* possède donc un controller nommé *ControlFormulaire* comme les autres.
Son controller donne donc un style au formulaire grâce à des balises HTML et CSS mais aussi une image de fond. Il définit initialement le nombre de joueur à 1, si l'utilisateur choisit d'augmenter le nombre de joueur, le nombre de mains augmentera grâce à la fonction `actionGoGameButton` qui définit l'action du bouton "Commencer". La méthode `actionReturnGameButton` définit l'action du bouton "Retour" qui renvoie sur la page d'accueil. L'action de la checkbox est géré par `actionCheckVintageButton` et `actionCheckClassiqueButton` ; cette checkbox demande à l'utilisateur le style de cartes qu'il souhaite. L'action du bouton du nombre de joueurs est géré par `actionListeGameButton`.
La classe *Formulaire* créée une JComboBox liste pour le nombre de joueurs, mais également des *JCheckBox checkVintage* et *checkClassique* pour le style des cartes. Elle ajoute les options sur la combobox, et appelle les fonctions d'action des boutons de son controller.

Antonin s'est occupé des tests et du *build.xml*, tandis que Lou-Anne s'est occupé du rapport.

3 Architecture du projet

Diagramme de tous les packages (et leurs dépendances) :



4 Eléments techiniques

4.1 Paquetages utilisées

Dans le package */lib* se trouve les paquetages JAR nécessaires au lancement du jeu :

nom librairie	**utilité**
cartes.jar	gestion jeu de cartes
hamcrest-2.2.jar	modules projectAPI
junit-4.13.2.jar	tests java
org-netbeans-modules-java-j2seproject-copylibstask.jar	tests
test.jar	nos tests

TABLE 1 – Liste des archives .jar utilisées

4.2 Pattern

La classe *DeckFactory* est un pattern de construction. Elle génère des *Deck* prédéfinis de 52 ou 40 cartes.
Dans la classe *Party* du package *blackjack*, nous avons définit un pattern de strategy. En effet, cette classe contient la main du croupier, et les mains des IA, qui joue leur tour, une fois celui de l'utilisateur terminé.

5 Conclusion

Nous avons donc un jeu complet avec intelligences artificielles et patterns, c'était bien l'objectif que nous nous étions fixés. Nous n'avons malheureusement pas eu le temps d'implémenter plus de patterns mais nous sommes quand même fier de ce que nous avons produit.