Rapport projet Blackjack

Antoine Collenot
Alix Danvy
Kenzo Lecoindre

Décembre 2022

1 Introduction

Dans le cardre du cours Méthode de conception dans lequel nous avons appris à utiliser des design pattern pour construire des applications maintenable, propre et efficientes, nous avons été amener à concevoir deux applications mettant en oeuvre ces concepts. La première est une application fournissant tout les composants et outils nécessaire à la construction d'un jeu de carte classique. Et la deuxième est l'implémentation d'un blackjack aux règle simple se basant sur la première.

Un mode d'emploi est disponible dans l'annexe pour les 2 applications.

2 Première application : Cards

Pour cette application nous avons basé notre architecture sur le modèle MVC (modèle-vue-controller) afin de garde une indépendance entre les composantes et laissez plus d'autonomie aux développeurs susceptible d'utiliser cette application, tout en leur offrant des fonctionnalités pré-implémenter selon leur besoin.

Cette partie ne possède pas de controller (hormis celui improvisé pour la démo), car elle n'a pas vocations à être utilisé en tant qu'application éxecutable par elle même.

2.1 Architecture : UML de cards

2.1.1 Package : model.cards

Notre modèle est assez riche de proposition de composants atomiques pré-fait en utilisant plusieurs pattern, tel que proxy, pour proposer une variétés d'options à l'utilisateur, tout en offrant un niveau d'abstraction supérieur avec tout les interfaces correspondant à ces éléments atomiques pour une plus grande liberté d'usage pour les cas plus précis.

2.1.2 Package : View

Ce package contient tout les composant permettant de créer une vue illustrant un jeu de cartes avec Swing. De plus nous proposons également une fenêtre préconfigurable, bien que limité tout de même pour une utilisation rapide dans des cas ne nécessitant pas plus de 4 joueurs et pour des jeux plutôt simple.

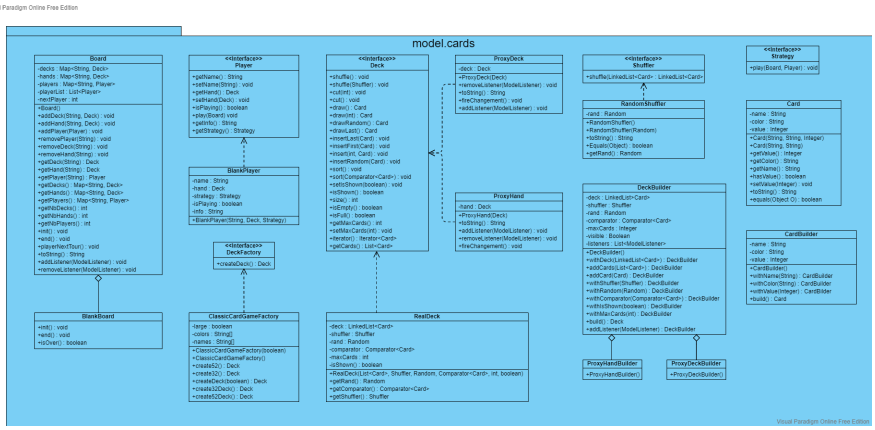

schéma UML du package model.cards

2.1.3 Package : util

Ce package contient tout les composant abstrait nécessaire à la MVC (Ecouteur de modèle et modèle observable). Il est utilisé dans les autres packages de l'application et est aussi à dispositions des développeurs utilisant cette application comme base de la leur.

2.1.4 Le projet complet

Dans ce schéma rassemblant les 3 packages vu précedement nous avons pris le partie de ne pas afficher toutes les associations et héritages entres les packages au vu d'une quantité d'information déjà très importante. Cependant, on peut se rappeler plusieurs choses :

1. tout le modèles est écoutable
2. les éléments de la vue son des écouteurs de ce modèle
3. on pourra toujours voir directement les associations dans les attributs visible de chacune des classes sur les diagrammes.

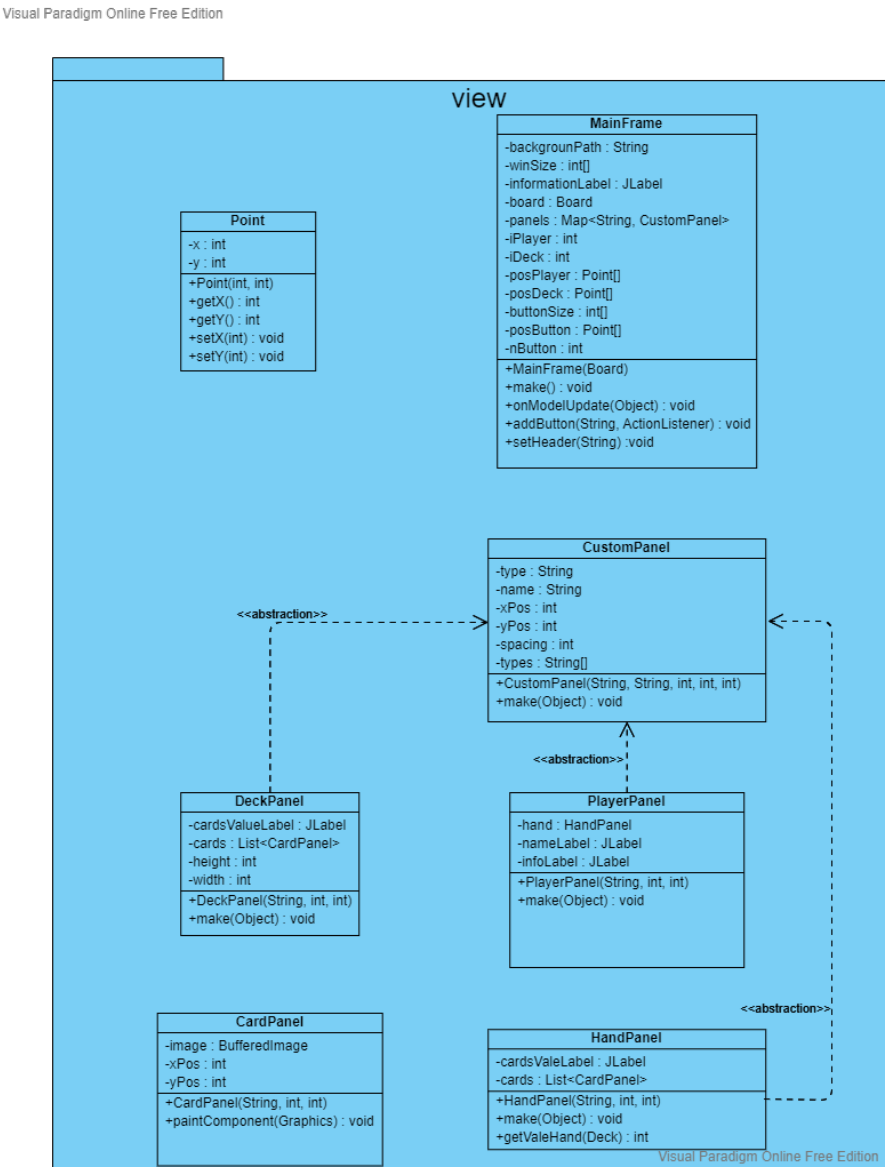

schéma UML du package view

2.2 Design pattern dans cards

Dans cette partie nous avons déjà commencé par mettre en oeuvre plusieurs des design patterns car ils nous semblaient pertinant et util pour ce projet :

— **Builder :** afin de nous assurer que les classes était bien instancié systématiquement avant toutes utilisations, nous avons mis en places des builder pour la class Card et Deck (avec CardBuilder et DeckBuilder). L'intérêt est double tant ces applications de builder nous facilite la vie lors de la

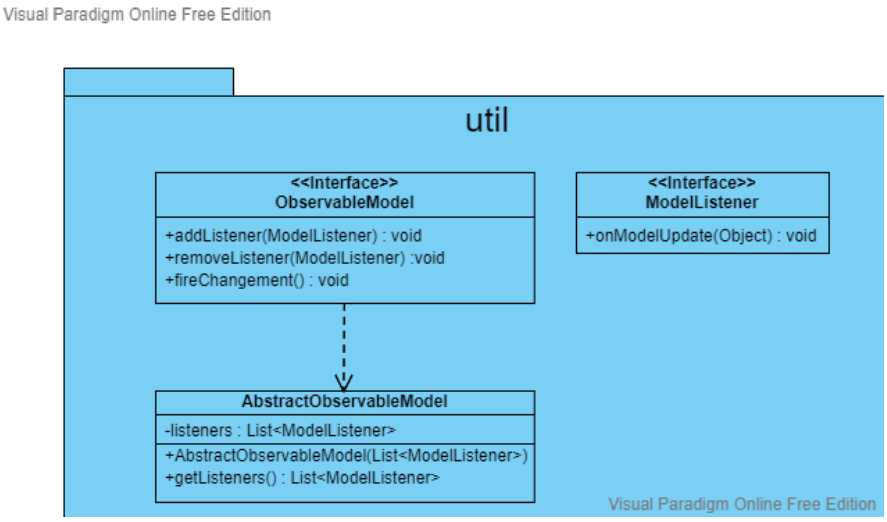

schéma UML du package util

construction et dans l'évitement d'erreurs de constructions.

- **Factory :** utilisé pour faciliter la construction d'objet courant, nous l'avons repris pour construire des jeu classique de 52 ou 32 cartes via la class ClassicCardGameFactory.
- **Proxy :** dans l'idée de bien séparer l'utilisation des paquets de cartes (Deck) en tant que main d'un joueur, visible et manipuable en interne, et en tant que paquet cachet, comme pioche disons, nous avons utilisé le pattern proxy pour envelopper les méthodes de Deck disponible et empêcher des manipulations non voulu. Note : ProxyHand et ProxyDeck possède tous les deux leur builder hérité du DeckBuilder.
- **Adapter :** dans le but d'utiliser une JFrame pour représenter un plateau de jeu, nous avons fait usage de ce pattern avec la classe MainFrame de la vue.
- **Template méthode :** dans le cas où notre jeu suivrait un schéma classique, la classe Board possède une méthode pouvant jouer une partie entière en s'initialisant, jouant tour après tour jusqu'à ne plus pouvoir, et de terminer la partie. Le tout en appelant des méthodes abstraites non définis dans cette class abstraite (init(), isOver() et end()).

3 Deuxième application : Blackjack

Dans cette applications, nous avons bien évidement repris les composants mis à notre disposition dans le but de réalisé une implémentation d'un blackjack avec des règles épurée servant de démonstration de ce qu'il est possible de faire avec les

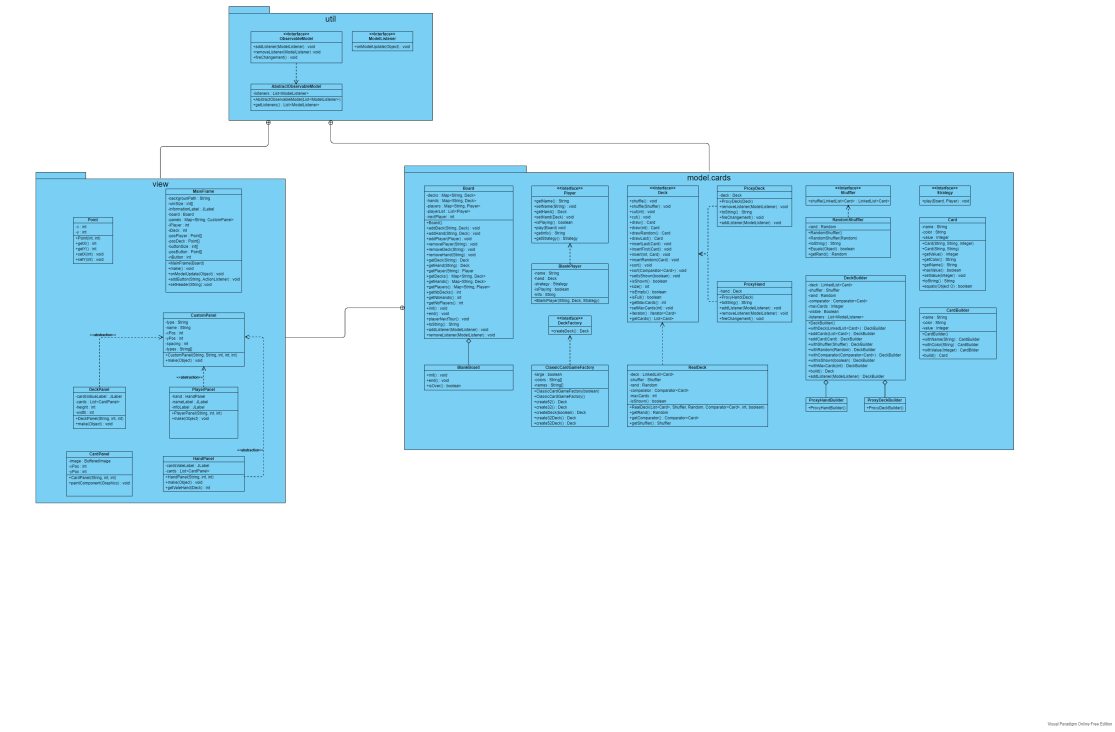

schéma UML de l'application cards

architecture vue en cours et en se basant sur notre première application.

3.1 Architecture : UML de blackjack

L'application est composé de 2 packages, controller et modèle (plus un autre contenant un éxecutable de l'application pour lancer une partie avec différents paramètes.). Une vue n'est pas requise car on peut reprendre la vue générale laissée par la précédente application et l'adapter à nos exigences ici.

Dans le package model.blackjack on retrouve tout les composant nécessaire à la constructions d'un blackjack.

Et dans le controller, on trouve le nécessaire pour faire jouer cette application et récupérer l'interraction possibles de joueurs humain via la fenêtre du jeu.

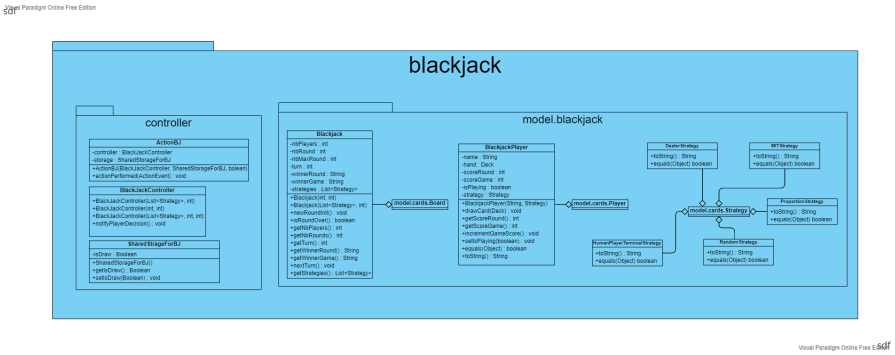

schéma UML de l'application blackjack

3.2 Design patterns utilisés dans blackjack

Dans cette partie nous reprenons principalement les patterns rendu possible par l'application précédente, à savoir le pattern Observer, avec l'architecture MVC, et le pattern Strategy, voir son extension State, dans le cas des actions réalisés ou non par state.

— **Strategy et State :** afin de choisir les actions réalisés par les joueurs, pouvant vouloir jouer de façon différentes (dans les cas des IA opposés à nos joueurs), on a décider de déléguer ces décisions à des classes externes qui implementent la classe Strategy disponible dans le modèle de cards. De plus, celle-ci s'adapte parfois de la conditions du joueur pour qui elles jouent et du plateau de jeu actuel, ce qui veut dire qu'elles utilisent le pattern State, une version de Strategy prenant en compte un environnement. On retrouve ce pattern avec DealerStrategy ou MITstrategy.

4 Tests unitaires

Durant l'entièreté de ce projet, nous avons fait usage de tests unitaires pour tester nos différents modèles. Ils se trouvent dans des packages à part de nos fichier sources pour bénéficier d'une meilleur propreté dans le rendu de chacune des applications sous forme de jar utilisable par d'autres.

Nos tests sont réalisé avec les librairies JUnit. Il faut donc les mettres dans les fichier lib de chacun des deux applications si vous voulez les lancer, à moins que vous n'utilisiez un IDE capable de le faire directement. Nous conseillons les versions junit-4.13.2 et hamcrest-core-1.3 car ce sont celle utilisé pour nos tests, mais d'autres versions ne devraient pas s'avérer incompatible.

5 Conclusion

Pour conclure, si nous avions eu parfois du mal à saisir l'utilité de toutes ces méthodes de conceptions en cours ou en tp, ces 2 projets ont achevés de nous convaincre de leur utilité, tant dans les opportunités qu'ils nous donnent que par le confort d'usage dans le développement en équipe.

Un mode d'emploi des applications est disponibles en annexe.

Annexe : Mode d'emploi

5.1 cards

Pour utilisé ce package, il est recommandé de lire la documentation si vous voulez l'utiliser dans vos projets. Cependant les noms sont assez clairs pour deviner quelle classe pourrait vous être utile durant vos recherches.

Notes :

— Pour lancer les tests, il faudra placer les librairies de junit nécessaire dans le dossier lib.

— Pour lancer et utiliser la vue, il est nécessaire d'avoir dans le fichier racide un répertoire assets contenant les images des cartes que nous utiliserons. Libre à vous de les changer mais faites attention au dimension et surtout aux noms.

Les commandes pour manipuler le projets avec ses sources (en utilisant ant) :

— Compiler le projet : $ ant compile

— Pour lancer les tests : $ ant test

— Pour générer la javadoc : $ ant javadoc

— Pour executer l'éxecutable principale du projet : $ ant run

— Pour générer le fichier dist contenant le jar du projet, y copiant les composant externe nécessaire, y générer la documentation. Pour ensutie copier le nécessaire vers le projet blackjack et lancer la démo : $ ant

blackjack

Même remarque que précedement, avec l'ajout en plus de la librairie cards et de images des cartes dans le dossier assets dans la racine.

Les commandes pour manipuler le projets avec ses sources (en utilisant ant) :

— Compiler le projet : $ ant compile

— Pour lancer les tests : $ ant test

— Pour générer la javadoc : $ ant javadoc

— Pour executer l'éxecutable principale du projet : $ ant run

— Pour générer le fichier dist contenant le jar du projet, y copiant les composant externe nécessaire, y générer la documentation. Pour ensutie copier le nécessaire vers le projet blackjack et lancer la démo : $ ant

Pour lancer le projet, on peut également donner des arguments pour changer la configuration de lancement du jeu, voir la doc.