Comparaison des diagrammes de classe

Sommaire

Phase de conception

Vous trouverez ci-après les diagrammes produits lors de la phase de conception et rendus à l'occasion du premier livrable.

Diagramme de classes réduit, modélisant uniquement le cœur logique de l'application

diagrammes-classe/initiaux/réduit.png

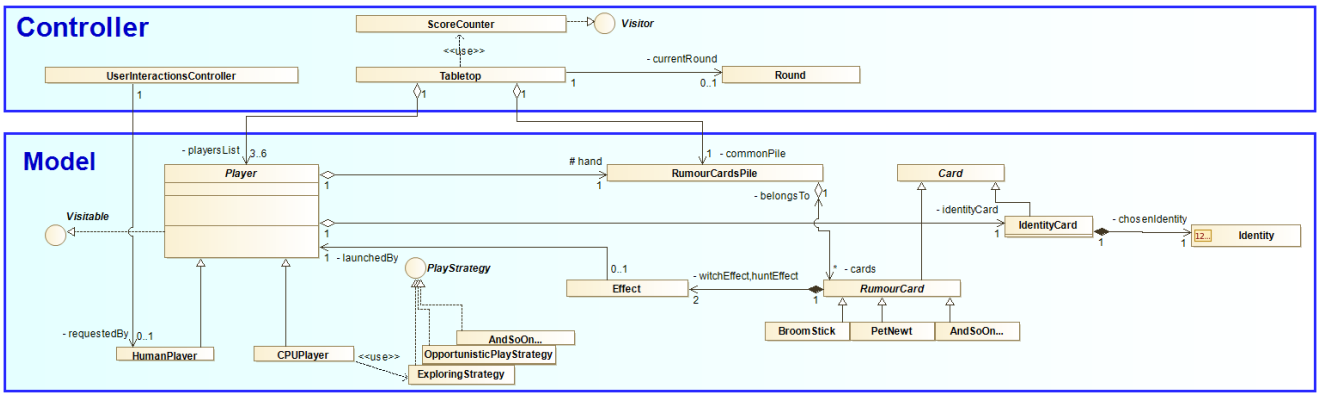
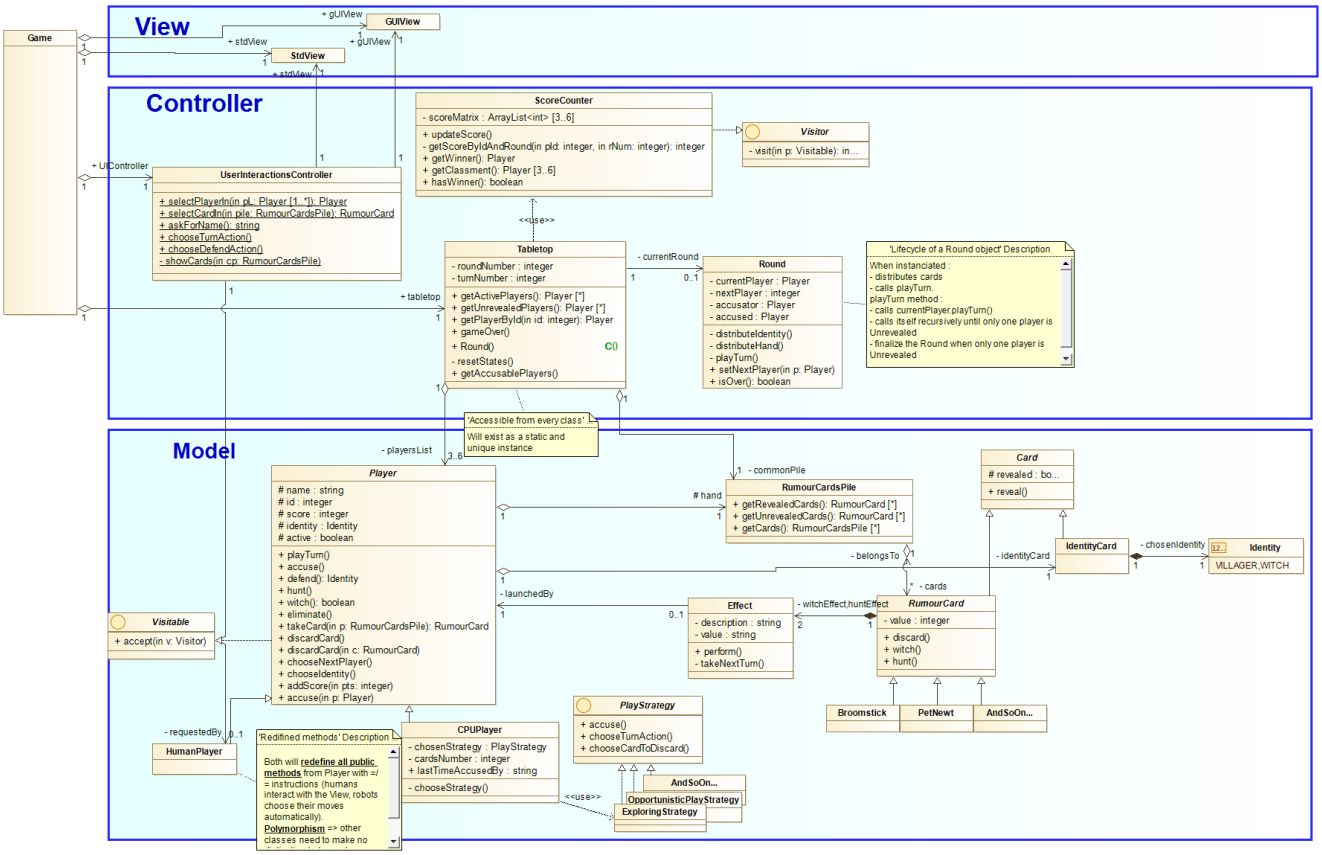


Diagramme de classe complet

diagrammes-classe/initiaux/complet.png



Diagrammes finaux

L'architecture du logiciel a subi des modifications conséquentes au cours de la phase de développement.

Diagramme réduit du modèle uniquement

diagrammes-classe/finaux/model.png

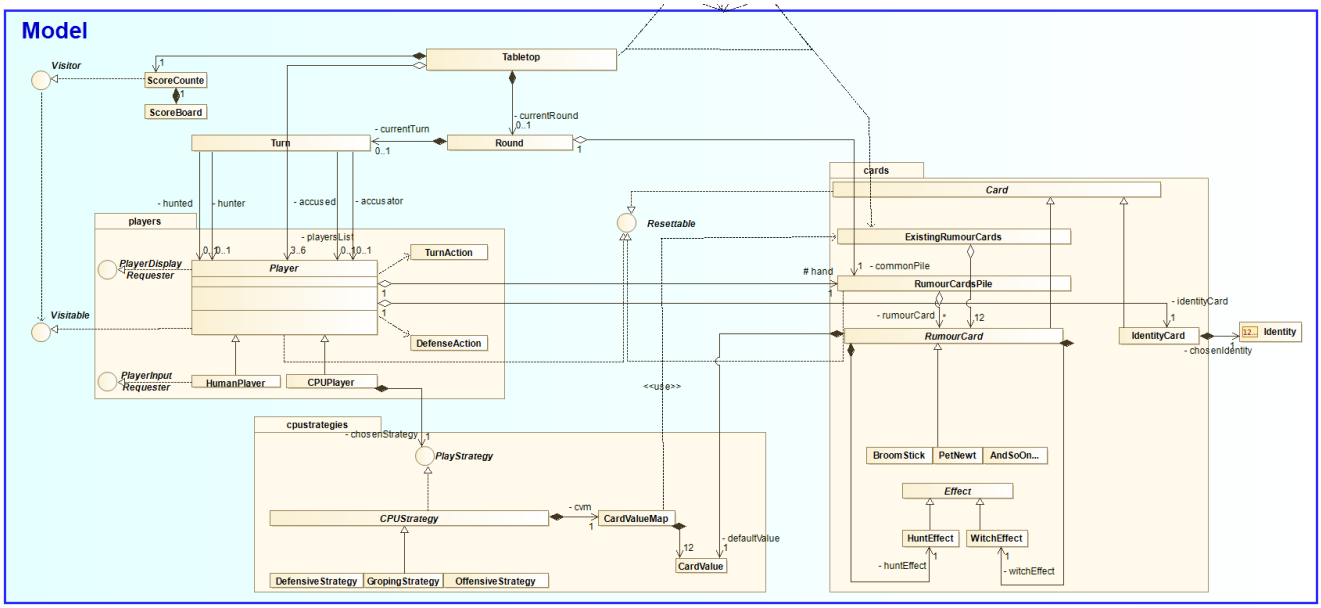
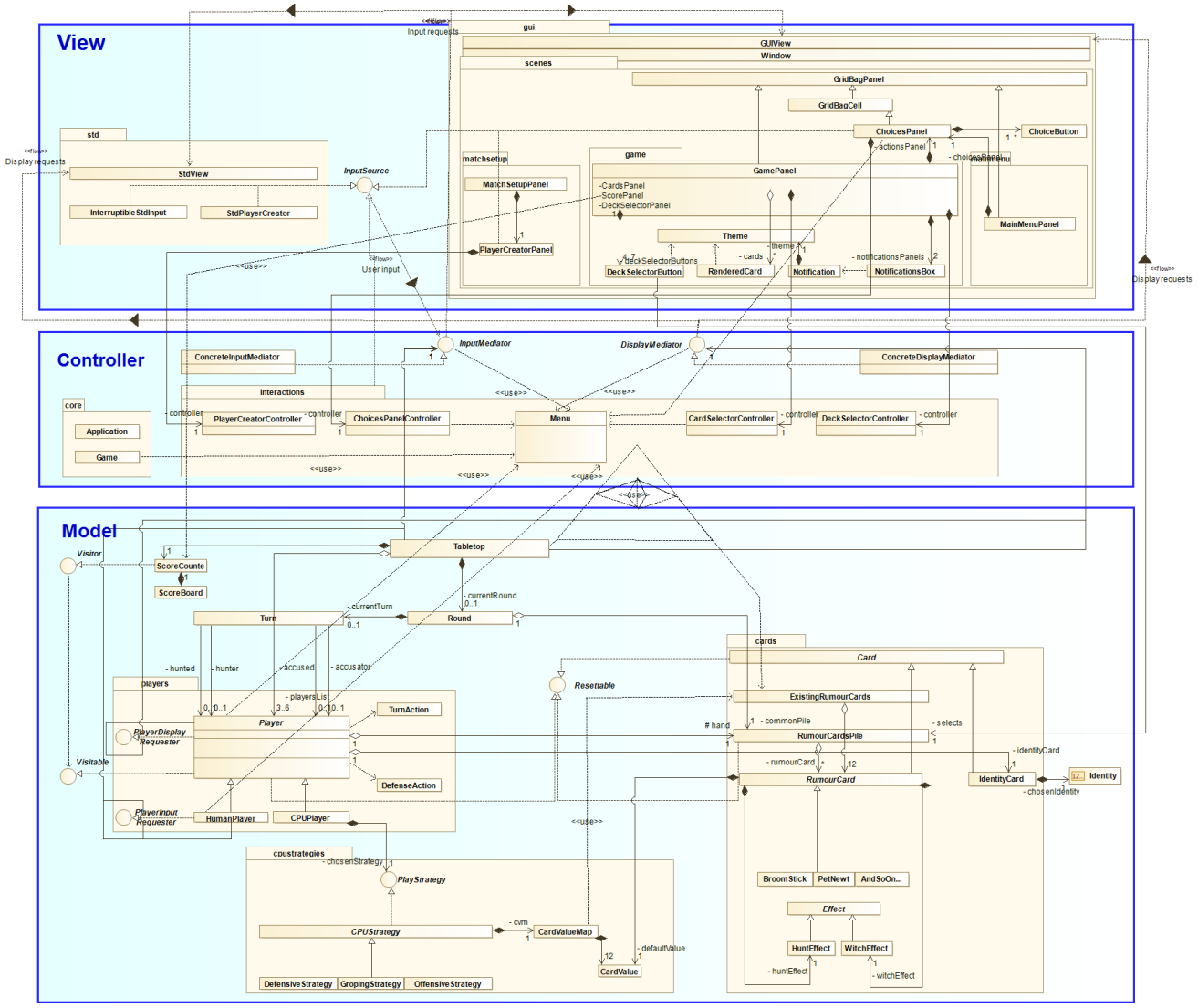


Diagramme réduit de l’ensemble de l’architecture

diagrammes-classe/finaux/full.png



Vous trouverez des versions en plus haute définition de ces diagrammes en dans le répertoire *diagrammes-classe/*.

Comparaison des architectures

Organisation en paquets

Afin d'organiser les classes par catégorie et par niveau de l'architecture MVC, nous les avons regroupées par paquets.
Malgré l'abscence de réelles relations de hiérarchie entre les paquets en java, nous avons cherché à utiliser des noms pleinement qualifiés les simulant.

- Il y a un paquet par niveau de l'architecture MVC.
- Les noms des paquets simulent une relation de hiérarchie, par exemple, le paquet `fr.sos.witchhunt.model` "contient" notamment les paquets `fr.sos.witchhunt.model.players` et `fr.sos.witchhunt.model.cards`.
- Les classes concernant un même ensemble de composants ou remplissant un rôle du même ordre sont regroupées par paquets.

  Par exemple, les classes *Round* et *Turn* figurent dans un même paquet en raison de leurs relations et de leur fonctionnalité qui consiste à gérer le flux de la partie.

Nous avons jugé utile de faire figurer cette organisation en paquets dans notre diagramme final.

Interface Resettable

Nous avons jugé utile de créer une interface nommée *Resettable* implémentée par certains éléments du modèle (joueurs, cartes et decks) pour spécifier explicitement que ces éléments sont destinés à être réinitialisés régulièrement, et notamment entre chaque round de la partie.

Par exemple, lorsque la méthode *reset()* d'un deck de cartes (*RumourCardsPile*) est appelée, celle-ci renvoie chacune des cartes du deck dans une liste contenant une instance unique de chaque type de carte *Rumeur*, utilisée pour la distribution des cartes au début d'un Round.

Autrement, lorsqu'un joueur utilise la carte *Pointed Hat* pour récupérer l'une de ses propres cartes, il appelle simplement sa méthode *reset()* pour la rendre jouable à nouveau.

Simplification des relations entre composants

Plusieurs associations présentes sur le premier diagramme de classes ont été supprimées car jugées inutiles et provoquant une redondance de l'information dans le programme.

Par exemple, les *Effets* ne sont plus directement associés aux joueurs les jouant, ils peuvent savoir qui en est à l'origine en interrogeant la classe *Turn* qui contient une référence au joueur accusateur/chasseur et au joueur accusé/chassé.

Il en va de même pour les cartes Rumeur, dont nous avons supprimé la référence au deck les contenant.

Classes gérant l'avancement de la partie

Nous n'avions initialement pas bien compris le rôle des contrôleurs dans le patron de présentation MVC et avions considéré les classes responsables de la gestion du flux de la partie comme en étant.

A présent, ces clases sont considérées comme faisant partie du modèle.

Patron de conception *Strategy*

Avant de commencer le développement du logiciel, nous avions à concevoir le fonctionnement de nos intelligences artificielles.

- Dans notre premier diagramme de classes, nous avons directement fait implémenter l'interface *PlayStrategy* par toutes les stratégies adoptables par les joueurs artificiels.
  Désormais, celles-ci héritent d'une classe abstraite *CPUStrategy* qui implémente elle-même *PlayStrategy*, en vue de factoriser le code **commun à plusieurs stratégies** et de mettre en place des intelligences artificielles **plus complexes.**

  Cela a notamment permis, par exemple, d'éviter que des joueurs artificiels ne jouent des cartes inutiles ou désavantageuses, voire suicidaires, en leur faisant **systématiquement évaluer leur situation, celle des autres joueurs et les effets des cartes qu'ils pourraient jouer** avant toute action.

- Des classes *CardValue* et *CardValueMap* ont été ajoutées pour permettre aux joueurs artificiels de réaliser de meilleurs choix de jeu en **attribuant à chaque carte des valeurs arbitraires.**

  Par exemple, un joueur artificiel au style de jeu prudent jugera ainsi les cartes lui permettant de remplir sa main ou de se protéger d'autres cartes comme étant plus intéressantes.

De plus, au cours de la phase de conception, nous n'avions pas une idée précise des différentes stratégies de jeu que nous définirions et de leurs spécificités.
A présent, il y a **trois stratégies** bien définies :

- Une stratégie **par tatônnement,** qui fait ses choix aux hasards et évite de dépenser ses meilleures cartes et qui est utilisée par défaut lorsque les conditions pour utiliser les autres stratégies ne sont pas réunies.
- Une **stratégie défensive**, qui fait des choix prudents et qui est choisie lorsque le round ou la partie deviennent serrés ou lorsqu'un joueur est en mauvaise position.
- Une **stratégie offensive**, plus encline à dépenser ses cartes et à prendre des risques pour tenter de marquer rapidement des points. Utilisée lorsque le joueur est en bonne position.

Effets

Nous avons au départ envisagé d'utiliser une seule et même classe pour représenter indifféremment les effets *Witch?* et *Hunt!* des cartes Rumeur.

Au cours de l'implémentation, il nous a finalement paru plus approprié de rendre cette classe abstraite et de la spécialiser pour les deux types d'effets possibles.

Enumérations

Les actions possibles pour un joueur pendant son tour, lorsqu'il est accusé ou lorsqu'un effet *Hunt!* lui pose un ultimatum, sont maintenant décrites par les énumérations *TurnAction* et *DefenseAction*.

Menus

Nous avons eu l'idée de modéliser aussi bien les menus permettant de naviguer dans l'application que les choix à réaliser au cours de la partie (sélectionner une carte, choisir un joueur à accuser ...) **à l'aide d'une même classe, Menu.**

En plus de **réduire le nombre de situations à gérer en terme d'affichage et de saisie**, cela nous a permis de **découpler efficacement le modèle et la vue** afin de respecter le patron de présentation MVC.

- Les classes du modèle représentent les **différentes possibilités** sous la forme d'un ensemble d'objets mais ne définit pas la manière dont ces choix doivent être représentés. Il s'agit seulement d'une retranscription des règles du jeu.
- **Chaque vue représente les Menus d'une manière qui lui est propre.**
- Les contrôleurs, en capturant un événement lorsque l'utilisateur agit sur la vue, permettent de sélectionner une entrée parmi un Menu.

Médiateurs faisant le lien entre la vue et le modèle

Nous avions dès le départ pris en compte les contraintes induites par la concurrence entre la vue Console et la vue graphique et par l'exigence de la mise en œuvre d'une architecture MVC, ce qui nous a permis d'avoir peu de changements à faire lors de l'implémentation de l'interface graphique.

Cependant, nous n'avions pas envisagé concrètement quelles solutions implémenter pour gérer l'affichage et la saisie simultanément sur les deux vues, et conserver l'indépendance du modèle par rapport à celle-ci.

La linéarité des événements survenant au cours d'une partie de *Witch Hunt* et la diversité des situations possibles requiérant un affichage un une entrée nous a motivés à mettre en place un système de modification utilisant des **médiateurs plutôt qu'un patron Observateur**.

Nous avons donc créé les interfaces *DisplayMediator* et *InputMediator*, des controlleurs qui spécifient une méthode d'affichage (resp. d'entrée utilisateur) envoyant des requêtes à la classe centrale de chaque vue.

La concurrence est gérée au moyen d'une saisie console interruptible, définie par la classe *InterruptibleStdInput*. Le médiateur de saisie sert également à synchroniser les vues lorsque l'une d'elles a répondu à une requête de saisie.