

Rapport projet de LIF 13 : Jeu de cartes de type « combat de créatures »

Rémi Courvoisier

It's Magic!				
Rémi: 10pv				
Ressources: 0/1				
CL 3	AL rapi... 1	AL rapi... 1	SA vola... 2	
4 2	2 1	2 1	1 3	
RC 4	CL 3	CG 3	LD enc... 2	
3 4	4 2	2 4	5 1	
Audrey: 10pv				
Ressources: 1/1				

Table des matières

- I. Présentation du projet
 - A. Architecture générale et diagramme de classe
 - B. Extensions implémentées
- II. Analyse du code
 - A. MVC
 - B. Design pattern décorateur
 - C. Monte Carlo
- III. Conclusion

I. Présentation du projet

A. Architecture générale et diagramme de classe

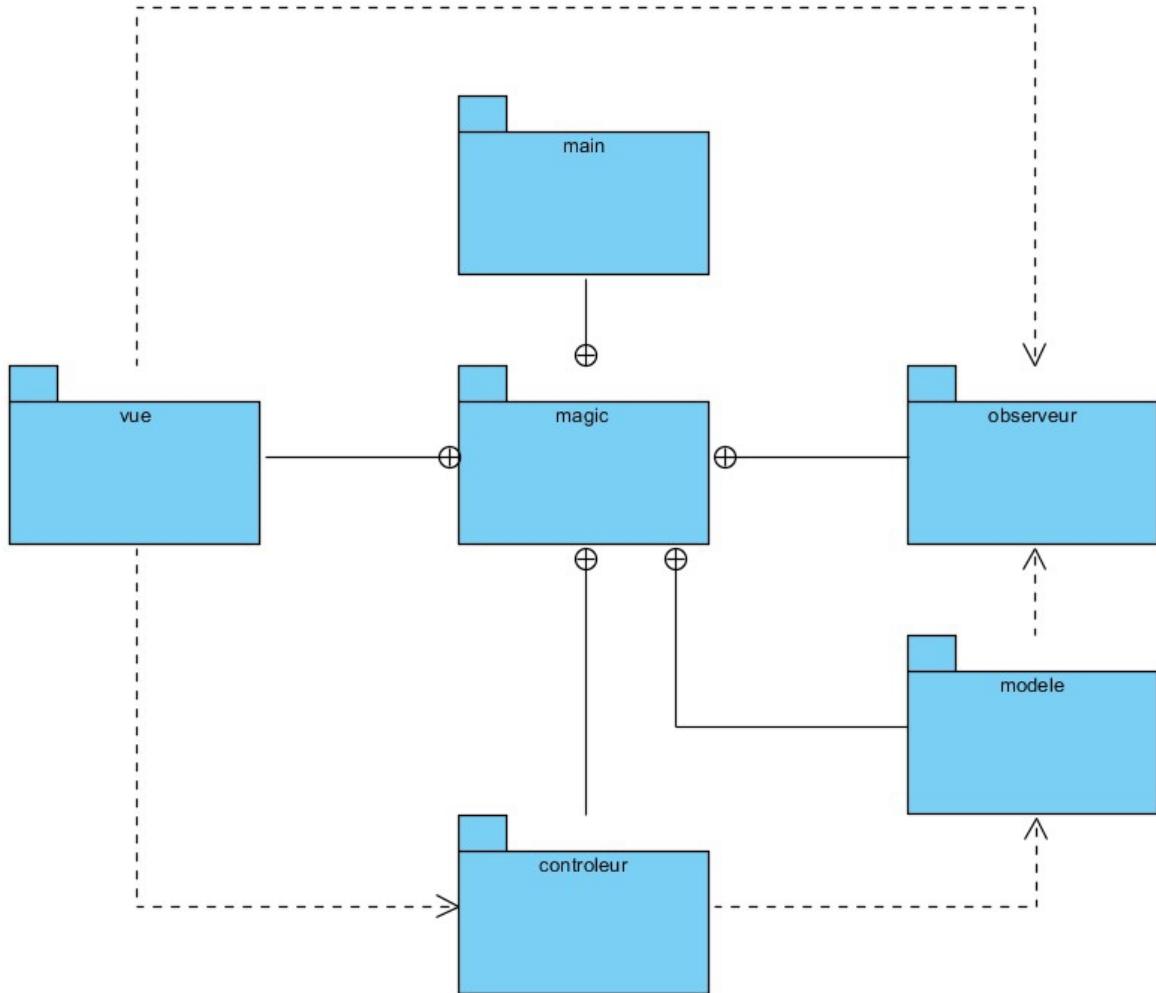


Illustration 1: Diagramme de packages

Le diagramme de classe du modèle est à trouver en annexe.

B. Extensions implémentées

La plupart des extensions proposées ont été implémentées. Nous avons :

- Une intelligence artificielle basée sur la technique de Monte Carlo (difficile). Lors de la présentation, cette extension était en cours d'implémentation. Elle est au final fonctionnelle, mais n'utilise pas de threads. En effet la mise en place du joueur Monte Carlo « simple » a déjà nécessité une importante quantité de travail et nous n'avons pas pu faire la version avec les threads.
- Le reste des règles enrichissant le jeu, de facile à difficile, sont présentes. Nous pouvons donc trouver la règle de fatigue qui s'applique à toutes les cartes sauf à celles ayant la faculté de rapidité (lors de l'invocation elles ne seront pas fatiguées et donc pourront directement attaquer). Les cartes rapides ont leur nom composé de « Rapide ».

On trouve également une carte « Totem » qui lors de son entrée en jeu apporte un bonus de 2 points de vie au joueur. Quant aux cartes « Vol », elles intègrent donc cette capacité qui permet un enrichissement des règles et de rendre le jeu plus tactique. On a également la carte « Ame soeur » qui permet de faire gagner +1 d'attaque et +1 de défense à toute carte invoquée après celle-ci, et tant que celle-ci est encore en jeu. Enfin nous avons intégré l'enrichissement des créatures pour leur ajouter des capacités supplémentaires. Ces enchantements sont « Enchantement Attaque », « Enchantement Défense », « Enchantement Vol » et « Super Enchantement » qui procurent respectivement +2 d'attaque, +3 de défense, la capacité vol et +4 d'attaque et de défense, à une autre carte du joueur déjà placée sur le terrain.

De plus, comme demandé lors de la présentation, lorsque l'on clique sur une case du plateau elle se colore, de telle sorte que le joueur peut repérer facilement ce qu'il est en train de faire (réalisé par Audrey).

II.Analyse du code

A. MVC

Le pattern MVC a été mis en place au début du projet comme demandé dans le sujet. Nous avons donc séparer les classes dans différents packages afin de mieux représenter celui-ci. On a donc un package Modèle, un package Vue et un package Contrôleur. Ensuite nous l'avons donc mis en place de sorte que la vue ne communique qu'avec le contrôleur qui traitent les données transmises avant de les envoyer au modèle, et le modèle notifie la vue lorsqu'il est modifié.

Dans l'utilisation qu'il en a été faite, il a été choisi qu'un minimum de classes traitent avec le modèle, c'est pourquoi seulement la classe Jeu et la classe Plateau sont observables et notifie la vue. Également il a été fait le choix, que plutôt d'utiliser différentes fonctions selon le type de notification, une seule a été mise en place avec un nombre d'arguments définis et pour une notification donnée certains peuvent être passés à null car ils seront non traités. Pour cela un code a été créé afin de reconnaître les notifications. Ce principe de fonctionnement pourrait plus s'apparenter à du langage C, et peut être que dans une prochaine intégration de ce design pattern une autre approche pourra en être faite.

B.Design pattern décorateur

Le pattern décorateur a été utilisé pour manipuler les enchantements. Grâce à celui-ci les cartes n'avaient pas besoin d'être remplacées par une nouvelles mais simplement la carte était enrichie avec une autre, ce qui rend le traitement bien plus facile. Nous ne savons pas si son utilisation a été optimale car peut être il n'est pas implémenté comme il aurait pu être attendu, cependant cela a rendu les enchantements opérationnels et simples d'utilisation.

Pour les effets « déclenchés », aucun design pattern n'a été utilisé. Une sorte de système de trigger ou l'utilisation d'un pattern s'en rapprochant aurait pu être mis en place, mais pour cause de temps et également de non perte de performance, il a été préféré de simplement rechercher sur le terrain parmi maximum quatre cartes possibles si une ou plusieurs devait déclencher un effet sur une autre.

C.Monte Carlo

Le joueur aléatoire Monte Carlo fonctionne de la manière suivante .

On calcule l'ensemble des coups possibles de la phase en court (exemple si c'est la phase d'invocation, on calcule tous les coups d'invocation possibles), grâce aux fonctions coupsInvocation/Defense/AttaquePossibles de la classe Plateau. Ces fonctions sont très longues, car un coup peut être composé de 0, 1, 2, 3 ou 4 actions (si on a 4 cartes sur son terrain qui ne sont pas fatiguées, alors on peut toutes les mettre en attaque, ou seulement 3, ou 2, ou 1 ou 0).

Pour chaque coup possible on clone le jeu, en transformant les deux joueurs en joueurs aléatoires faciles*, on applique le coup, et on fait tourner le jeu (tout seul) ; et ce 15 fois. On prend le coup qui a le plus de parties gagnantes ; si tous les coups donnent des parties perdantes alors on prend un coup nul ; si plusieurs coups ont le même nombre de parties gagnantes on prend le coup qui a le plus grand nombre d'actions.

*Si le joueur aléatoire facile peut jouer (par exemple il a 2 cartes non fatiguées sur le terrain en phase d'attaque), alors il joue forcément un coup à une action (il va attaquer avec une des 2 cartes), sinon il ne joue pas.

Cette extension a nécessité d'apporter plusieurs modifications à notre diagramme de classe. En effet nous avons ajouté une classe pour chaque type de coup (CoupInvocation, CoupDefense, CoupAttaque), et mis en place une classe JoueurAleatoireFacile et une classe JoueurAleatoireDifficile.

III. Conclusion

Malgré un temps assez court avant l'oral, nous avons réussi à nous organiser et à commencer à mettre en place la quasi totalité des extensions proposées. Au final nous avons un projet assez abouti ; chaque extension ayant été implémentée.

En ce qui concerne le binôme, le travail a été bien réparti avec Audrey sur le modèle de base puis sur l'IA difficile avec Monte Carlo qui a pris beaucoup de temps, et Rémi sur MVC et la création de la vue dans un premier temps puis l'ajout des différentes fonctionnalités dans un second temps. La communication se faisait bien dans le groupe, la forge de l'université a été utilisée pour partager le code ainsi que les réflexions nécessitant de l'aide se sont faites à deux.

Annexe

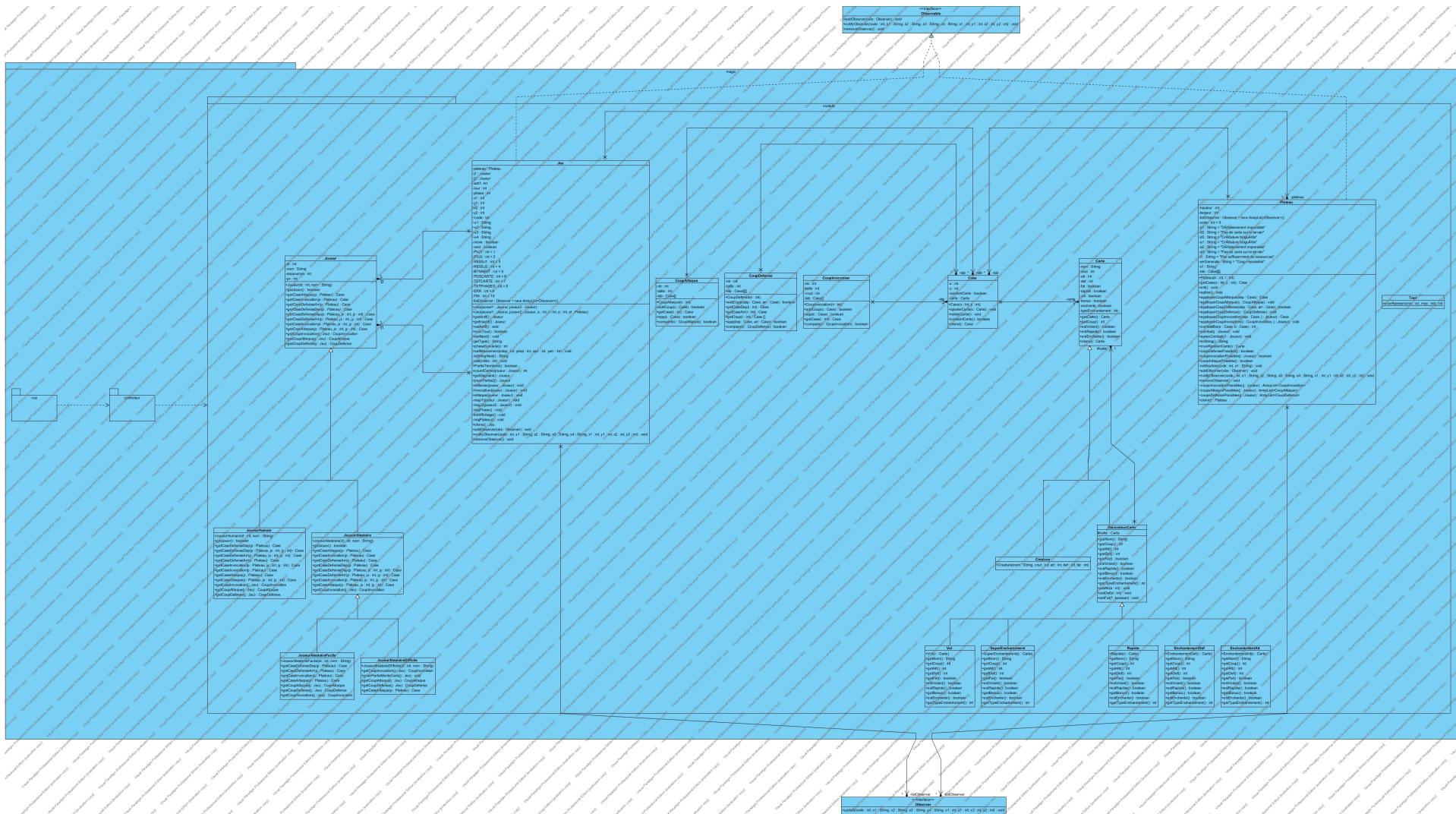


Illustration 2: Diagramme de classes du modèle