GAGNAIRE Romain (ISI)
BIERGE Laurent (SRT)

Université de Technologies de Troyes
Année 2013-2014

Livrable 01
Application - Jeu de cartes UNO

Sommaire

1- Introduction

Dans le cadre de la formation d'ingénieur à l'Université de Technologie de Troyes, nous a été proposé un sujet de **projet** nous permettant de mettre en œuvre les **connaissances acquises** depuis le début du semestre dans un domaine particulier informatique : celui du développement applicatif en langage de programmation **Java**.

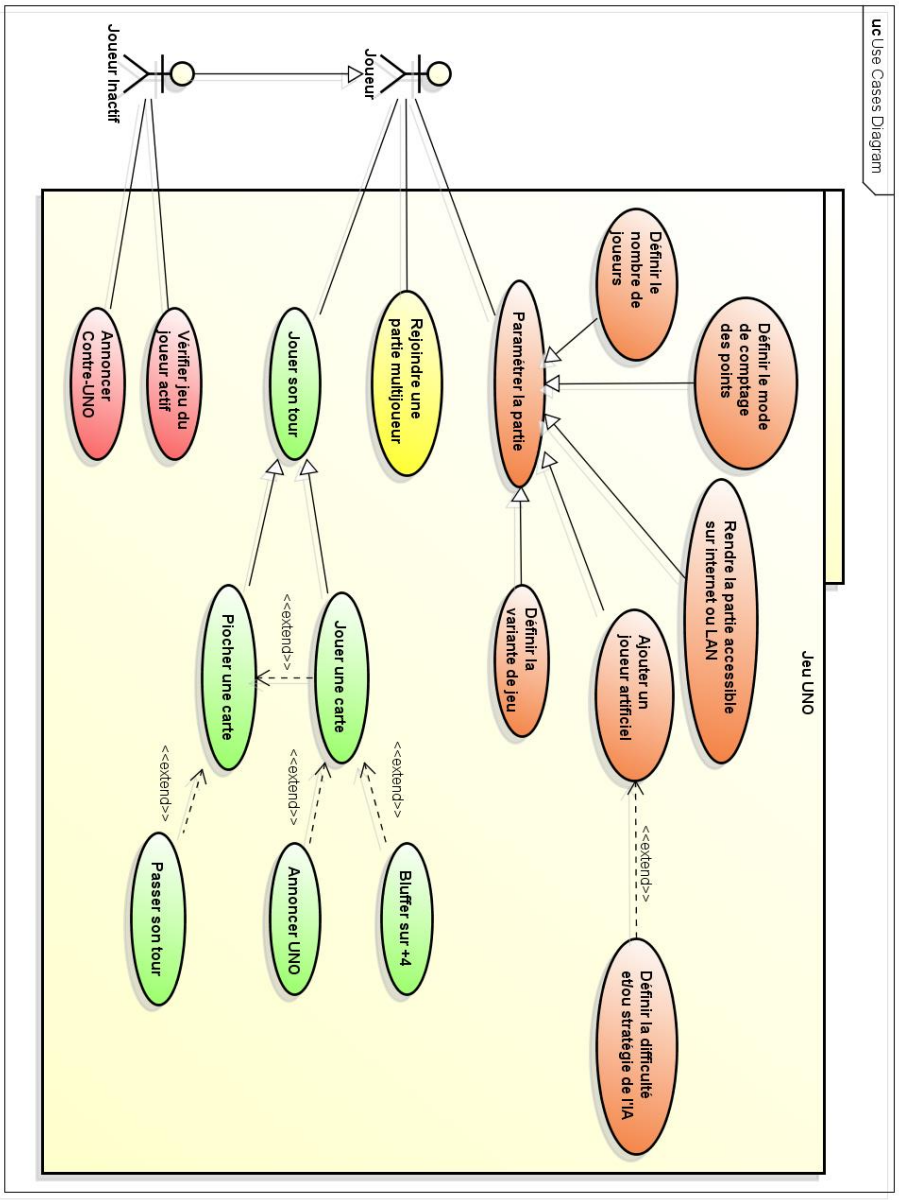
L'objectif est de créer en binôme une application retranscrivant les conditions de jeu du fameux **jeu de cartes UNO** de manière **modulable**, en procédant par étapes : d'abord en **lignes de commandes**, puis en rajoutant une **interface graphique**.

Le principe du jeu UNO est assez simple : il peut avoir de 2 à 10 joueurs possédant initialement 7 **cartes** qui sont distribuées depuis la pioche. Il existe 108 cartes différant par leur **couleur**, leur **numéro** ou leur **type** : il existe en effet 4 couleurs différentes, 9 numéros différents, et certaines cartes déclenchent des **effets spéciaux** quand elles sont jouées. Le but du jeu est de réussir à être le premier à jouer toutes ses cartes. Il existe également différentes variantes des règles, mais elles ne seront pas abordées dans ce document.

Une **partie** se joue en plusieurs **manches**. Une manche se termine lorsqu'un des joueurs a joué toutes ses cartes. Il existe différentes méthodes pour compter les points mais la règle générale reste la suivante : la partie se termine lorsqu'un joueur atteint 500 points. Le joueur qui a le plus (ou le moins) de points remporte la partie.

Dans ce premier livrable seront détaillés différents points de **modélisation**, qui serviront de base à la **conception**. Nous commencerons tout d'abord par expliciter quelles sont les interactions possibles entre l'utilisateur et le programme. Puis nous verrons quelles décisions cela a entrainé concernant les classes d'objets qui composeront le programme. Ensuite, le fonctionnement normal d'un tour sera mis en lumière afin d'exposer les principes de base du fonctionnement du logiciel. Enfin, dans une dernière partie, nous reviendrons sur la globalité de la mise en place de ce projet, en dressant un premier bilan sur ce qu'il adviendra de nos choix d'implémentation.

2- Diagramme de Cas d'utilisations



Avant de commencer, il est bon de se placer dans l'optique d'une **application en réseau** (accessible grâce à un terminal, ou ordinateur, différent de celui hébergeant la partie). De cette manière les rôles des différents acteurs de la partie sont mieux définis.

Tout commence avec le premier joueur de la partie, que nous allons nommer « **hôte** » par soucis de simplicité. L'hôte va d'abord configurer les différents **paramètres** de la partie : comme par exemple le nombre de joueurs maximum (de 2 à 7), la variante de jeu (jeu à 2 joueurs, jeu en équipe, challenge UNO, etc.) et la façon dont sont comptés les points.

Il pourra également vouloir rajouter des **joueurs non humains**, qui seront gérés par la machine, dont le **comportement** pourra être choisi parmi une liste de comportements prédéfinis (comme par exemple : une IA qui joue prioritairement ses cartes spéciales, ou ses cartes numérotées, etc.).

Bien qu'optionnel, on peut très facilement imaginer que l'hôte pourrait vouloir rendre la partie **accessible depuis l'extérieur**, lui permettant ainsi d'affronter des joueurs humains plutôt que des intelligences artificielles. Dans ce cas d'autres joueurs pourraient **vouloir rejoindre** sa partie, que ce soit par internet ou par un réseau local (LAN).

Dans tous les cas, le joueur ayant la main, qu'il soit humain ou pas, va vouloir **jouer son tour**. Deux cas peuvent alors se présenter : soit il dispose dans son jeu d'une carte jouable et il peut alors la jouer, dans le cas contraire il va devoir **piocher** une carte. Les deux mêmes cas de figurent se présentent à nouveau : soit il dispose désormais d'une carte jouable et il peut la jouer, ou dans le cas défavorable, il doit **passer son tour** sans jouer de cartes. A noter que certaines cartes permettent d'empêcher le joueur suivant de passer son tour, aussi, si jamais c'est le cas, le joueur ne pourra donc pas **défausser** de cartes ni piocher de cartes à ce tour-là.

Certains cas particuliers peuvent se présenter en cours de partie. Ce sont les seuls moments où les autres joueurs, dits « **inactifs** » (qui sont en attente de leur tour), **peuvent interrompre** la partie.

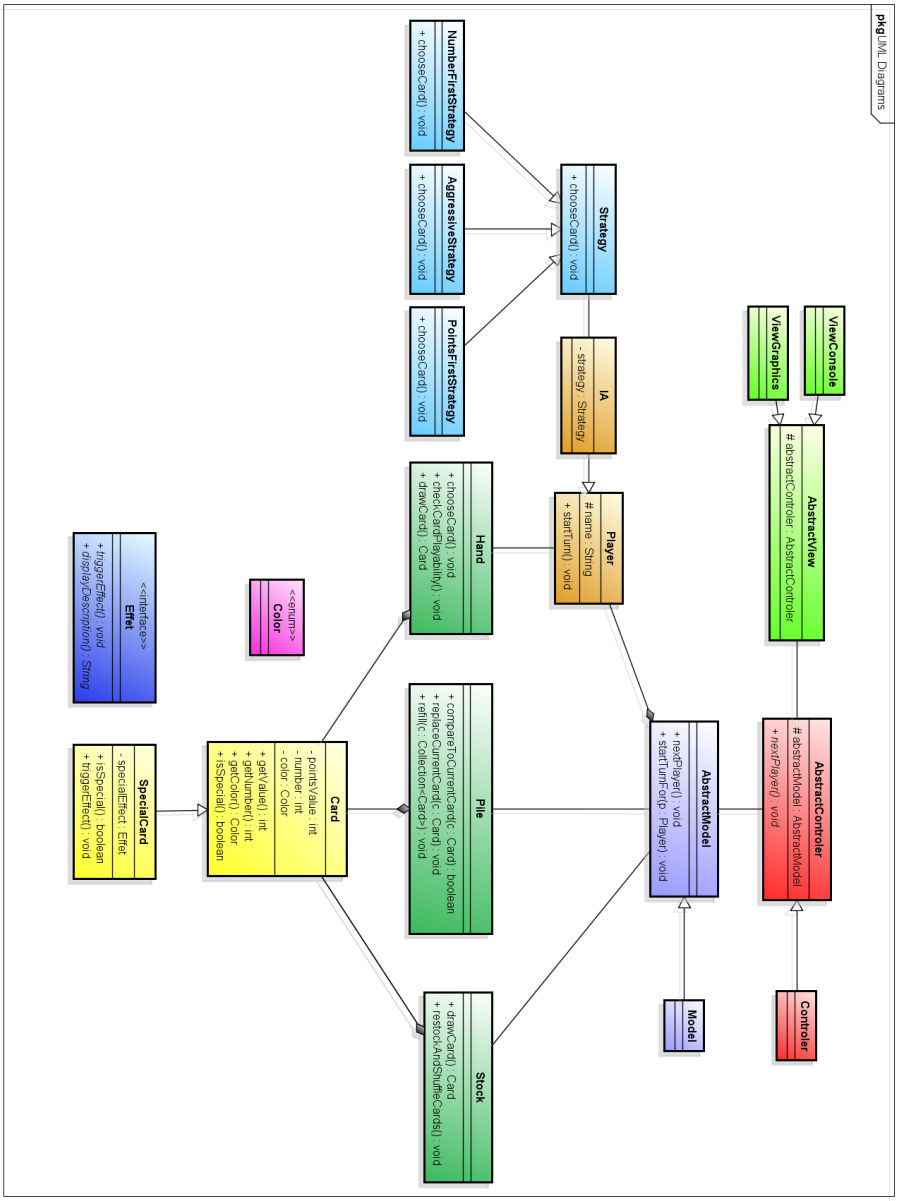
Le premier arrive lorsqu'un joueur utilise une carte spéciale appelée « **+4** ». Celle-ci est un joker permettant à la fois de choisir quelle sera la couleur à jouer, d'empêcher le joueur suivant de jouer, et de lui faire piocher 4 cartes. Or, cette carte ne doit être jouée –en théorie- **que si aucune autre carte n'est jouable**.

Cependant, il est possible de **bluffer**, et de passer outre cette règle. Dans le cas d'un joueur ayant utilisé une carte « +4 », il est possible pour n'importe quel joueur inactif de **demander à vérifier** que le jeu de cette carte était légitime : il est alors libre de vérifier le jeu de l'adversaire. Mais il est soumis à une **règle très simple** : si cette carte avait été jouée en tout état de cause alors le joueur suspicieux doit piocher 2 cartes (en plus des 4 initialement prévues, s'il s'agit du joueur suivant). Dans le cas contraire (s'il y a bluff avéré), le joueur fautif pioche 4 cartes et passe son tour.

Le deuxième cas correspond à celui où le joueur actif se débarrasse de son **avant-dernière carte,** ne lui restant ainsi plus qu'une unique carte dans la main. Il doit alors –en théorie- annoncer « **UNO** » avant de terminer son tour.

Si jamais il **omet cette annonce**, et qu'un de ses adversaires le remarque, il leur est possible le lui faire remarquer en annonçant « **Contre UNO** ». Le joueur fautif doit alors piocher deux cartes. A noter que toute **utilisation abusive** de l'annonce « UNO » (joueurs ayant plus d'une carte en main après avoir joué), ou de l'annonce « Contre UNO » (en direction d'un joueur ayant plus d'une carte dans la main après avoir joué) est **puni de la même manière**.

3- Diagramme de Classes



Par soucis de **simplification** du diagramme, les constructeurs des différentes classes ne seront pas indiqués. De la même manière, toutes les méthodes de chacun des objets ne sont pas reportées sur ce diagramme, ceci afin de ne pas le **surcharger** d'informations (qu'il reste à la fois lisible et compréhensible).

Afin de procéder à une analyse complète de ce diagramme, nous vous proposons de commencer au **plus bas niveau** (les classes d'objets de base du programme) pour remonter progressivement vers les **couches de plus haut niveau** (celles les plus évoluées).

Tout commence avec la classe « **Carte** ». Elle est constituée de tous les éléments qui la caractérisent : sa couleur, sa valeur (son nombre de points) et son numéro. Sa couleur correspond à une classe d'énumération, pour distinguer les différents cas très simplement.

Afin de différencier les cartes spéciales et les cartes numérotés, une méthode a été créée « **isSpecial** » renvoyant faux s'il s'agit d'une carte numérotée, et vrai si elle est spéciale. Une carte spéciale n'est rien d'autre qu'une carte classique avec un **effet supplémentaire**.

Le comportement de chaque effet sera **standardisé** par l'utilisation d'une interface. Jouer une carte spéciale entrainera le déclenchement automatique de l'effet associé par le simple appel à « **triggerEffect** », et ce quel que soit l'effet de la carte.

Chaque des cartes peut se trouver à un des 3 endroits suivants : dans la main d'un joueur (« **Hand** »), dans la pioche (« **Stock** ») ou dans le talon (« **Pile** »). Au départ toutes les cartes sont créées dans la pioche. Puis 7 cartes sont distribuées à chacun des joueurs (par appel à « **drawCard** »), les transférant depuis la pioche vers leur main. De la même manière, si un des joueurs doit piocher une carte au cours de son tour, il sera fait appel à la même méthode.

Lors du tour d'un joueur, s'il est amené à pouvoir jouer une carte, il est libre d'en sélectionner une parmi celles de sa main. Lorsqu'une carte est sélectionné (« **chooseCard** »), sera vérifié qu'elle est compatible (« **checkCardPlayability** ») avec la dernière se trouvant sur le talon (« **compareToCurrentCard** »), et tant que la carte sélectionnée n'est pas compatible, le joueur doit sélectionner une nouvelle carte. Lorsqu'il en sélectionne une valide, alors elle remplace est ajoutée au talon (« **remplaceCurrentCard** »).

Si jamais un joueur pioche la dernière carte de la pioche, elle doit être reconstituée à nouveau (« **refill** ») pour pouvoir continuer à jouer. On va donc récupérer toutes les cartes du talon, à l'exception de la carte au sommet servant de base, et les mélanger pour ensuite les insérer dans la pioche (« **restockAndShuffleCards** »).

Comme dit précédemment, chaque joueur possède un ensemble de cartes qui forment sa main. Il possède également un ensemble d'informations qui permettent de le différencie des autres comme son nom/pseudo. Les joueurs contrôlés par l'ordinateur (« **IA** ») sont eux aussi des joueurs à part entière. Ils disposent des mêmes capacités qu'un joueur normal, mais choisissent leur carte (« **chooseCard** ») en fonction de critères prédéterminés : leur **stratégie**. Celle-ci est définie lors de la création de la partie, et ne pourra pas être changée en cours de partie.

L'implémentation de la stratégie est régie selon un patron de conception spécifique appelé « **Strategy Pattern** ». Il permet de résoudre très simplement les cas où certaines classes ne seraient différenciées que par leur **comportement**. C'est le cas, comme son nom l'indique, avec la stratégie d'une intelligence artificielle, mais c'est aussi vrai pour les effets des cartes spéciales. Cette implémentation permet une **interchangeabilité facilitée** : il est très simple d'ajouter ou de supprimer des comportements sans avoir à répercuter des modifications sur le reste des classes, et de la même manière il est aisé de donner à chaque de **répondre de manière unique** aux comportements attendus.

Ensuite on arrive au plus haut niveau de l'application : la partie **Modèle-Vue-Contrôleur** (MVC). Son objectif est assez simple : restreindre au maximum les responsabilités de chacune des entités. On va différencier donc 3 **rôles** différents :

- Le support des données, va correspondre au modèle (« **AbstractModel** ») et ses réalisations.
- L'affichage de ces données et la réception des interactions provenant de l'utilisateur, va correspondre à la vue (« **AbstractView** ») et ses réalisations. On peut la voir comme une représentation visuelle du modèle
- Le lien entre le système et l'utilisateur du programme se fait par le contrôleur (« **AbstractController** ») et ses réalisations.

En effet, lorsqu'un utilisateur effectue un traitement sur la vue, celle-ci informe le contrôleur qu'une action s'est produite. Le contrôleur va donc réagir à cet **évènement** et effectuer toute une suite d'actions soit directement avec soit en utilisant les donnée provenant du modèle. Si des éléments du modèle sont modifiés, le contrôleur demande à la vue de s'actualiser afin de refléter avec exactitude les nouvelles données.

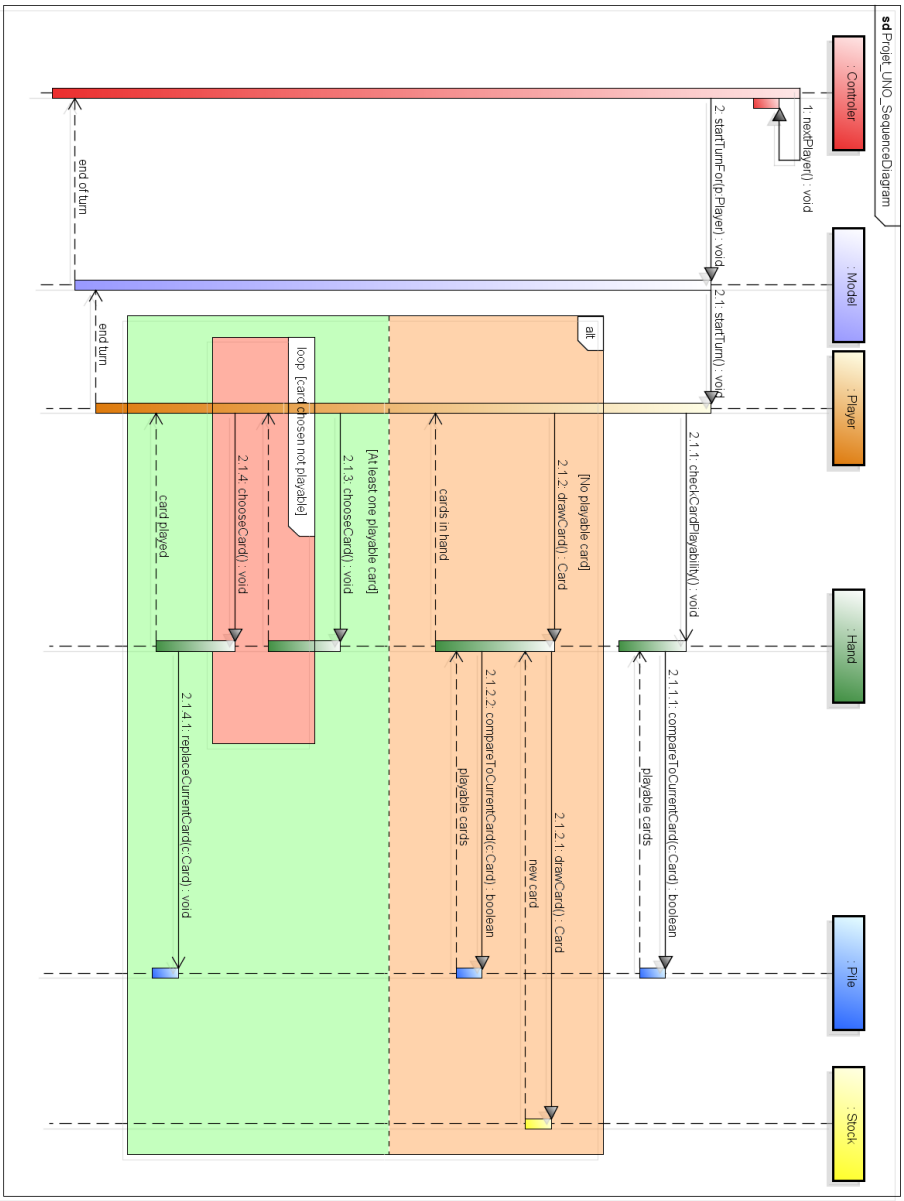
A noter que, pour fonctionner, ce patron de conception fait un usage implicite du patron **Observateur-Observé** (Observer pattern) avec de pouvoir coordonner les actions des différents éléments en fonction des interactions de l'utilisateur (évènements).

Les **raisons** derrière ce choix technique sont **multiples**.

Pour commencer il est bon de séparer les **responsabilités** de chaque classe. Par l'implémentation de ce patron de conception, l'objectif est bien de respecter autant que faire se peut le **principe de responsabilité unique**, stipulant qu'il ne doit exister qu'une et une seule raison de modifier une classe.

Une deuxième raison poussant vers le choix de ce patron est qu'il introduit une réelle **modularité** en ce qui concerne chacun des 3 éléments précédents. En effet, chacun d'entre eux doit être **interchangeable**, ce qui est attendu concernant la vue : on s'attend à pouvoir afficher les mêmes données que l'interface soit graphique ou en ligne de commande. On souhaite pouvoir choisir l'une ou l'autre très simplement, sans avoir à apporter de modifications majeures sur l'arborescence des classes ou leur code interne respectif. Ce patron de conception est loin d'être parfait, mais il n'en reste pas moins l'une des **meilleures approches**.

4- Diagramme de Séquence



A chaque tour de jeu, le même **cycle** va se répéter pour chacun des joueurs, jusqu'à ce que tous ait joué. Le contrôleur va donner la main au joueur suivant (ou donner la main au joueur initial si on est en début de manche – par le biais de la méthode « **nextPlayer** »). Il va indiquer au modèle quel joueur à la main (« **startTurnFor** ») en précisant par son numéro dans la partie : par exemple, si la main est donnée au deuxième joueur, son numéro sera 2. Le modèle va donc répercuter les modifications nécessaires pour que le joueur puisse jouer son tour.

Automatiquement, le système va vérifier si le joueur dispose d'une carte jouable (« **checkCardPlayability** »). Ceci sera opéré en comparant le comparant les cartes du joueur à la dernière carte de la pile (« **compareToCurrentCard** »). Typiquement deux cartes seront compatibles si elles ont la même couleur, ou le même numéro ou si elles sont du même types (ou dans certains cas, si elles n'ont pas de contraintes de précédence).

Dans le cas où le joueur ne dispose d'aucune carte jouable, alors il piochera une carte avant qu'il puisse réellement agir. A nouveau, le système va vérifier s'il dispose d'une carte jouable, en comparant la carte piochée à la carte du sommet de la pile (par appel à la même méthode « **compareToCurrentCard** »).

Dans le cas où le joueur ne dispose pas de carte jouable alors il n'a d'autres choix que de mettre fin à son tour.

Dans le cas contraire, le joueur reste libre du choix de la carte qu'il va jouer. Cependant, le système vérifiera que la carte peut effectivement être jouée (de la même manière que les deux comparaisons précédentes).

Et tant que le joueur ne choisit pas une carte compatible, le système l'informe de ce fait, et lui intime de choisir une autre carte. Ce choix de comportement a été préféré au fait de n'afficher au joueur que les cartes qui sont jouables pour des raisons très simples : le joueur reste conscient du nombre de cartes dont il dispose et il reste libre du droit à l'erreur (sans influence sur la partie)

Une fois qu'une carte compatible a été jouée, elle est posée sur la pile, prenant alors la place de la carte précédente (« **replaceCurrentCard** »). Puis le joueur doit terminer son tour.

5- Conclusion

La **modélisation**, et tout ce que ça implique, permet une meilleure appréhension du programme lors de son **développement**. Un diagramme de cas d'utilisation permet de s'approprier le cahier des charges et définir l'aspect fonctionnel du programme. Par ailleurs, un diagramme de classes traduit la nature des relations liant les classes de manière à identifier les différents points critiques. Et enfin, le diagramme de séquence permet de donner une vue rétrospective sur le diagramme de classe.

Néanmoins, certains défauts de conception ne se dévoilent que lors de l'implémentation. Il arrive souvent, au cours du développement d'une application, que la modélisation initiale **évolue** de manière à les résoudre et intégrer de nouvelles fonctionnalités.

De plus, le développement d'un programme étant **dynamique**, il est bon de procéder à **l'amélioration** de la syntaxe et de l'agencement interne des différentes classes, afin de maximiser son efficacité, ou limiter les dépendances, ou encore séparer le rôle de chacune des entités. Aussi, certaines classes seront amenées à **évoluer**, ou à **être scindées** en de multiples sous-classes.

Les **points sûrs** de la modélisation sont multiples.

Tout d'abord, toutes les cartes seront contenues dans des **collections**, qu'elles soient de type pile pour la pioche et le talon, ou encore de type collection à accès direct pour la main de chaque joueur. Cet aspect de la modélisation rendra le codage plus efficace et plus robuste.

Les cartes à effet spécial restent des cartes, et c'est la raison pour laquelle elles **héritent** de la classe mère carte, leur permettant de partager les mêmes types d'attribut, et certains **comportements**. De plus, cela permet d'avoir des collections **homogènes**.

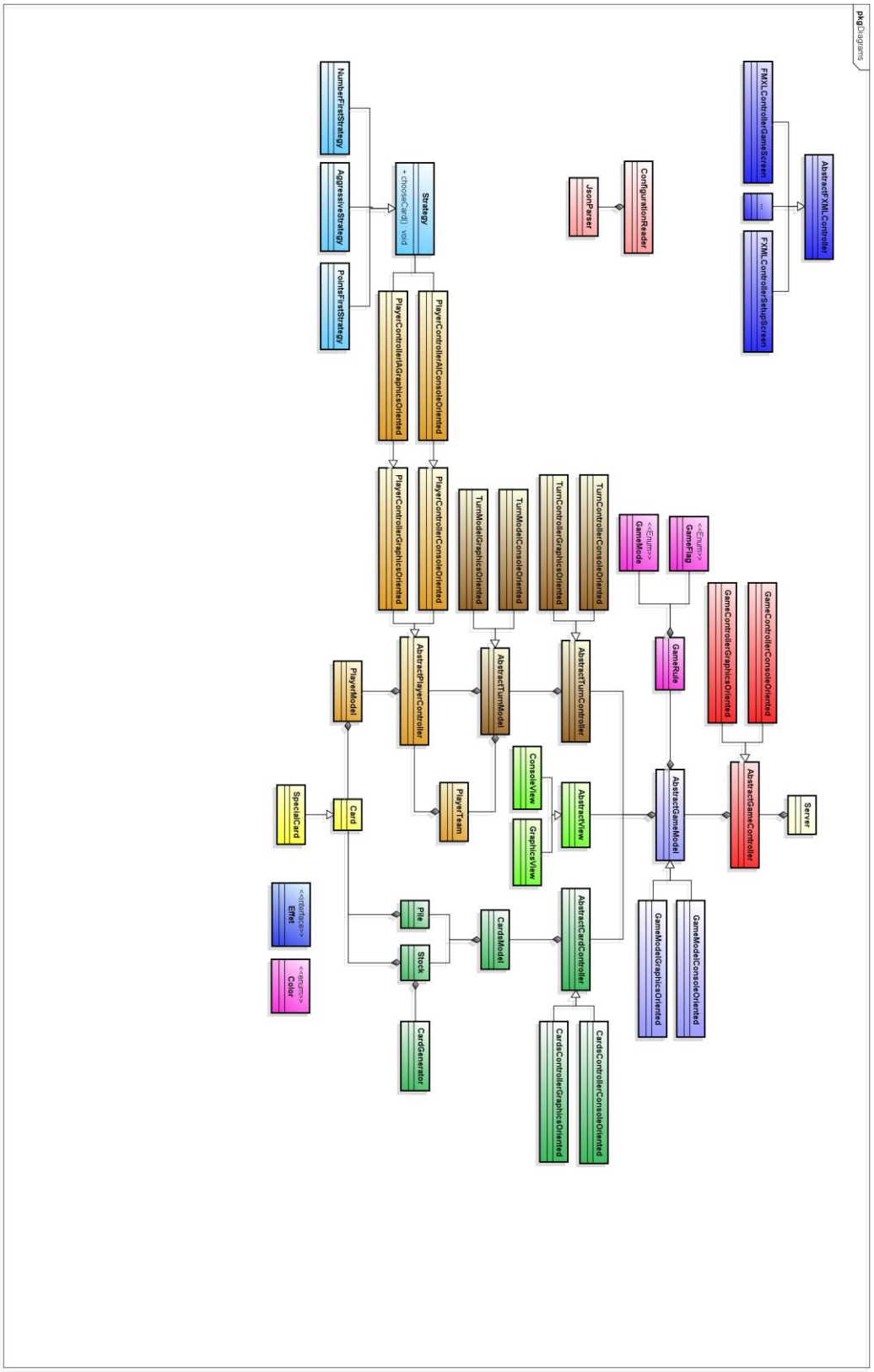
Ensuite, le déclenchement d'un effet est **normalisé**, de sorte à pouvoir déclencher n'importe lequel de manière **uniformisée**. Il est alors très simple d'en rajouter de nouveaux, ou d'en supprimer.

De la même manière, la gestion de la stratégie des intelligences artificielles, guidée par le patron de conception **stratégie**, restera immuable tout au long du développement.

La conception sous la forme d'un modèle **MVC** est elle aussi une base certaine. Cependant, il n'en pas impossible que les différentes entités soient **divisées** en plusieurs sous-classes permettant de mieux répartir les responsabilités et de **réduire** les classes à un état qui se veut le plus petit possible.

En conclusion, on peut dire que ce projet est une expérience **enrichissante** et très **motivante** qui aboutira sur une **meilleure compréhension** à la fois de la conception et du développement en langage de programmation **Java**.

6- Modélisation Finale



7- Etat Actuel de l'Application

Avant de commencer, il est bon de noter qu'un ReadMe digne de ce nom est disponible à l'adresse GitHub du projet : http://goo.gl/83zgWC

L'application en version console est complète à 100%. Voire même au-delà, puisque des modes de jeu supplémentaires (2 joueurs, en équipe) sont inclus. Elle dispose d'un affichage en couleur par le biais d'utilisation de code ANSI, aussi bien dans la console Windows (cmd), que Linux (shell) ou celle d'Eclipse.

L'application dispose des fonctionnalités suivantes :

- Maven pour la gestion automatisée des dépendances
- Journalisation grâce à Log4J, avec différents niveaux (debug, info, warning, error)
- Système création rapide par le biais de chargement d'un fichier de configuration formaté en JSON (utilisation de Gson pour le parsing)
- Tests unitaires afin de valider le comportement des méthodes (avec notamment injection de dépendance de sorte à pouvoir valider le comportement de la gestion de la réception d'information au clavier)

L'application avec interface graphique fait appel à JavaFX. Il s'agit d'un ensemble d'outils créés par Oracle permettant de concevoir et de créer des applications riches en java. Il s'agit d'une surcouche à AWT/SWING permettant des fonctionnalités de plus haut niveau. Un outil est mis à disposition par Oracle afin de faciliter le développement d'applications utilisant JavaFX : JavaFX SceneBuilder permettant de créer intuitivement des interfaces graphiques par le biais d'outils visuels. JavaFX permet également de définir des interfaces graphiques par l'utilisation de fichiers FXML et CSS simplifiant la syntaxe.

Cependant, par simple manque de temps, la gestion de l'interface graphique n'est pas complète à 100%. L'application JavaFX est une sorte de boite avec laquelle il est difficile de communiquer depuis l'extérieur. C'est pour cette raison, qu'un refactoring massif à dû avoir lieu, et que les méthodes permettant la gestion des parties et des tours de chaque joueur est presque totalement différent entre l'interface console et l'interface graphique.

Elle dispose à l'heure actuelle d'une gestion quasi complète d'un jeu à deux joueurs et peut être très facilement étendue vers un jeu à n joueurs (la seule limitation étant le design des différentes vues, avec coordonnées de placement des cartes des différents joueurs). La limitation n'est nullement située au niveau du code. Un simple délai supplémentaire aurait suffi pour l'accomplissement total du projet selon le cahier des charges.

Bien que le projet ne soit pas aussi abouti que nous l'aurions souhaité, il reste néanmoins un bon point de départ pour servir de vitrine à ce que nous sommes capables de faire, et entendons bien mettre à profit l'inter-semestre pour achever notre ouvrage.