# Funo

## Rapport final du projet de programmation orienté objet LO02


**Réalisés par :**
Haythem BEN MESSAOUD
Florent LUCET

**Encadrés par :**
Guillaume DOYEN
Matthieu TIXIER

**Diplôme :**
Diplôme d'ingénieurs
(Spécialité ISI)

**Période :**
Automne 2013

# SOMMAIRE

# Introduction

**Le jeu du Uno est un jeu de cartes où le but principal est de se débarrasser des cartes contenues dans notre main.** C'est un jeu qui peut se jouer de 2 à 10 joueurs. Vient d'abord la phase de distribution des cartes, où un joueur s'occupe de mélanger l'ensemble des cartes et d'en distribuer 7 à chaque joueur, face cachée. Deux tas de cartes sont alors intégrés au jeu :

- La **pioche**, constituée du reste de l'ensemble des cartes une fois la distribution terminée. Les cartes sont faces cachées.

- Le **talon**, constituée de la carte située sur le dessus de la pioche. Les cartes sont faces visibles.

La première personne à jouer est celle située à gauche du donneur (joueur ayant distribué les cartes). Ensuite, chaque tour suit le même schéma : d'abord, le joueur peut jouer une carte s'il le veut et s'il le peut, c'est-à-dire s'il possède une carte de la même couleur, du même numéro ou du même symbole de celle sur le dessus du talon. S'il pose une carte directement, c'est alors au tour du prochain joueur, c'est-à-dire le joueur à sa gauche. S'il décide de ne pas – ou ne peut tout simplement pas – poser une carte, il en pioche une nouvelle dans la pioche. À nouveau, il peut décider de la jouer ou non en la posant sur le talon, s'il peut. Une fois son choix fait, c'est au tour du prochain joueur.

De plus, les cartes peuvent être de deux grands types. Elles sont soient normales (numérotées de 0 à 9), soient spéciales (avec un effet particulier). Ces dernières peuvent être de cinq types d'effet différents, et prennent effet quand un joueur les pose sur le talon :

- Cartes **"Passer un tour"** : le joueur suivant doit passer son tour.

- Cartes **"Inversion"** : le sens du jeu change. Si la partie ne comporte que 2 joueurs, la carte prend le même effet qu'une carte "Passer un tour".

- Cartes **"+2"** : le joueur suivant doit piocher 2 cartes et passer son tour.

- Cartes **"+4"** : le joueur suivant doit piocher 4 cartes et passer son tour. Le joueur l'ayant utilisé doit également choisir la couleur de la carte. Elle ne peut être jouée par le joueur que si aucune autre carte n'est jouable, bien que cette règle puisse être transgressée, auquel cas le joueur s'expose à une pénalité si le joueur suivant souhaite vérifier en regardant sa main. Si cette carte est la première carte du talon, on pioche à nouveau une carte dans la pioche pour la déposer sur le talon.

- Cartes **"Joker"** : le joueur l'ayant utilisé doit choisir la couleur de la carte. Elle peut être jouée à n'importe quel moment. Si cette carte est la première carte du talon, le donneur doit choisir la couleur de la carte.

Plusieurs règles viennent entourer et pimenter la structure de base du jeu :

- La **règle "+4 suspect"** : quand la carte "+4" est jouée par un joueur, le joueur suivant peut se montrer sceptique de son utilisation dans les règles et demander à voir la main du joueur ayant posé la carte "+4". S'il pouvait la jouer, c'est-à-dire si c'était la seule carte de sa main jouable, alors le joueur suivant doit piocher 6 cartes de la pioche au lieu de seulement 4. Sinon, c'est-à-dire si le joueur suspicieux avait de l'être, le joueur de la carte "+4" doit la reprendre et piocher 4 cartes à la place du joueur suspicieux.

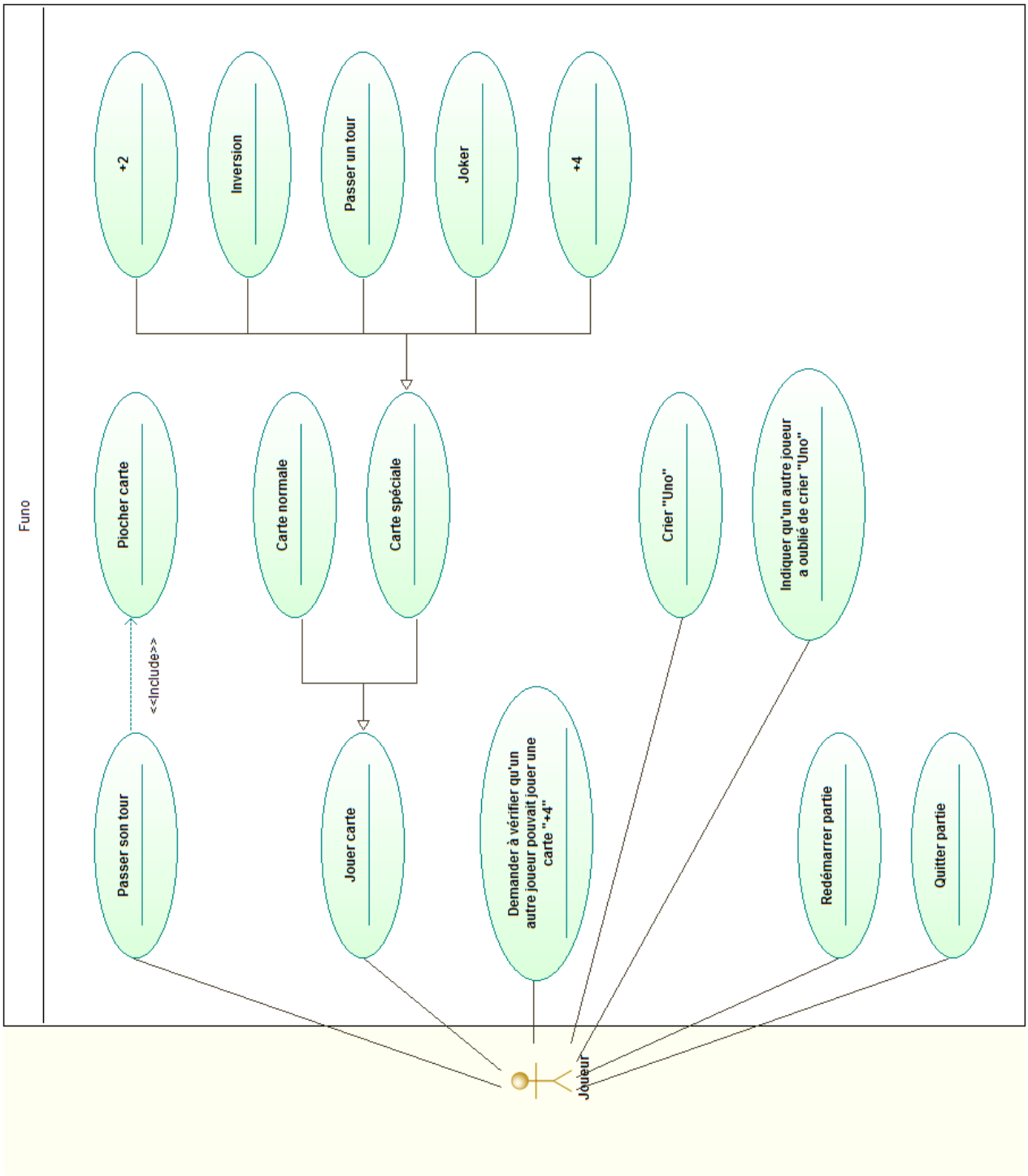
- La **règle "Uno"** : quand un joueur pose l'avant-dernière dernière carte de sa main, il doit crier "Uno". S'il l'oublie et qu'un autre joueur lui fait remarquer d'ici le dépôt ou la pioche d'un carte par le prochain joueur, celui ayant oublier de crier "Uno" doit piocher 2 cartes de pénalité de la pioche.

- La **règle "dernière carte posée : +2/+4"** : quand un joueur pose une carte "+2" ou une carte "+4" comme dernière carte, l'effet de la carte posée s'applique juste avant la fin de la manche, c'est-à-dire que le joueur suivant doit piocher 2 ou 4 cartes selon la carte posée.

- La **règle "pioche vide"** : quand la pioche est vide, la dernière carte du talon devient le nouveau talon, et tout le reste des cartes est mélangé pour former la nouvelle pioche.

**Enfin, le jeu peut se jouer de différentes manières.** Il est possible de jouer à deux (duquel cas les règles sont légèrement différentes d'un jeu de trois à dix joueurs), de faire un chacun pour tous classique, un jeu en équipes (seulement si le nombre de joueurs est pair) et un Uno challenge (où les joueurs sont éliminés au fur et à mesure des tours). De plus, le chacun pour tous classique peut se jouer de manière à ce que les points soient positifs (la somme des points données par les cartes des mains adverses est ajoutée au score du gagnant de la manche) ou soient négatifs (quand quelqu'un gagne la manche, la somme des points données par les cartes de la main de chaque joueur est ajoutée aux scores respectifs de chaque joueur).

# I) Diagramme de cas d'utilisation



Notre diagramme de cas d'utilisation met en avant les fonctionnalités recherchées dans le logiciel Funo.

La première fonctionnalité qu'il nous semblait impérative d'inclure, c'était le fait que le joueur puisse **jouer une carte**. Cette fonctionnalité est au centre du jeu de cartes de Uno, et même des jeux de cartes en général. Jouer une carte se fait en plusieurs étapes. Déjà, jouer une carte nécessite qu'on en ait la possibilité, c'est-à-dire qu'une carte en correspondance avec celle sur le dessus du talon soit présente dans la main du joueur. Si cela est le cas, le joueur peut choisir de joueur cette carte.

De plus, cette fonctionnalité est divisée en plusieurs catégories, elles-mêmes divisées en sous-catégories. Cela suit les règles du jeu, et correspond au fait qu'un joueur e joueur peut jouer soit une carte normale (une carte numérotée de 0 à 9), soit une carte spéciale (une carte avec un effet particulier). Les cartes spéciales peuvent être de cinq types d'effet différents, et nous ne les réexpliquerons pas dans les détails pour éviter d'être redondants avec l'introduction de ce rapport. Ces cinq types d'effet différents sont les cartes "+2", "Inversion", "Passer un tour", "Joker" et "+4".

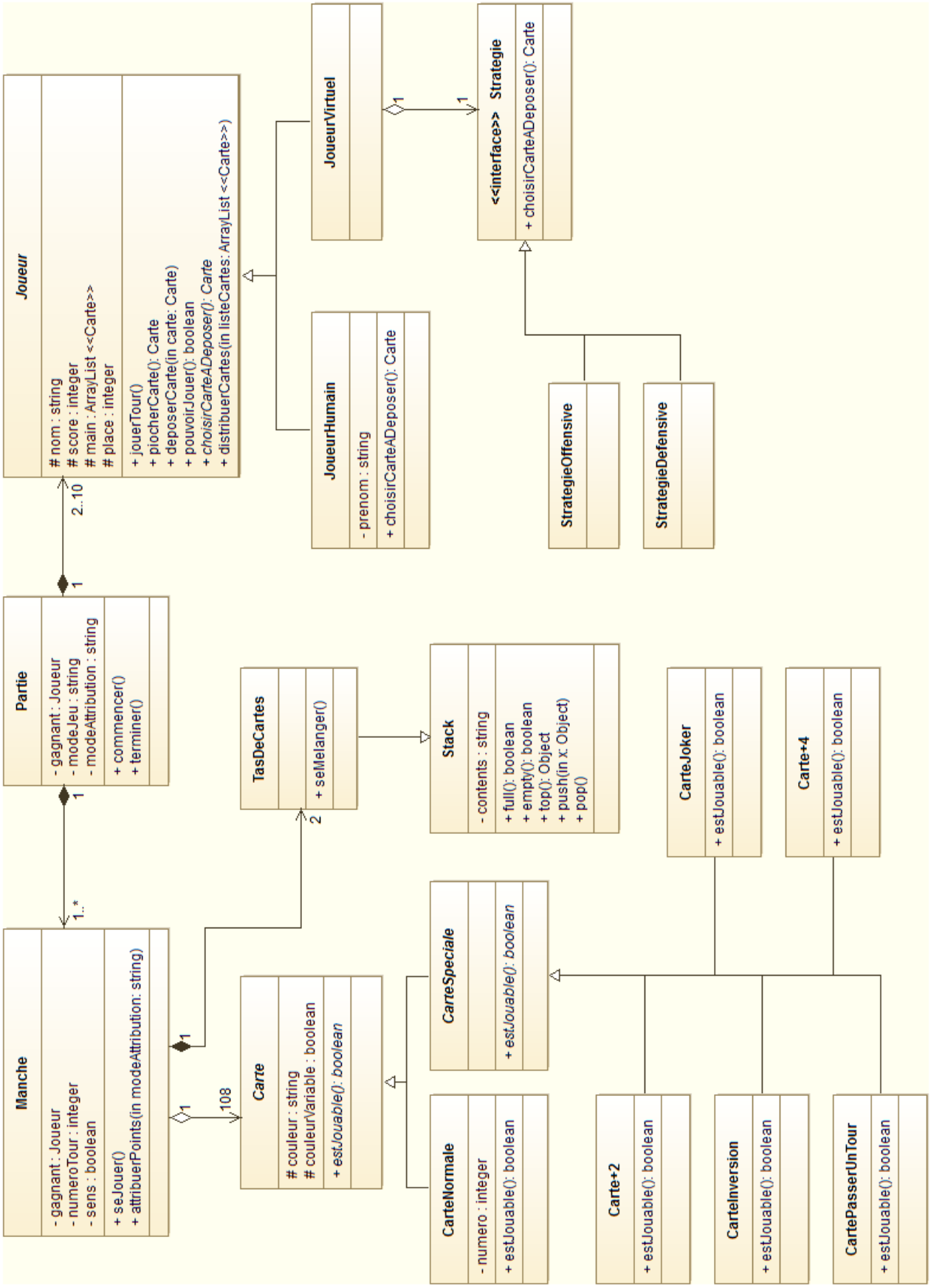
La deuxième fonctionnalité qui nous semblait primordiale, c'était celle qui consistait à ce que le joueur puisse **passer son tour**. Cela lui donne une liberté d'action au-delà du seul hasard, étant donné que même lorsqu'il peut faire autrement (jouer une carte), il garde la possibilité de passer son tour. Pour passer son tour, le joueur doit cependant piocher une carte auparavant.

Une troisième fonctionnalité, utilisée moins souvent que les deux autres mais néanmoins importante, est celle qui consiste à **vérifier qu'un joueur ayant joué une carte "+4" avait le droit de la jouer**. Dans ce cas-là, il faudrait permettre à un seul et unique joueur de voir la main d'un autre joueur. Ensuite, s'appliqueraient les règles de pénalités pour l'un ou pour l'autre, en fonction du droit de jouer la carte "+4" ou non.

Les quatrième et cinquième fonctionnalités sont au centre du jeu. Il s'agit pour un jouer de **crier "Uno"** quand il a déposé son avant-dernière carte, et pour un autre d'**indiquer qu'il a oublié de crier "Uno"** si cela est arrivé. Dans ce cas-là s'applique la pénalité qui consiste à faire piocher 2 cartes dans la pioche au joueur n'ayant pas crié "Uno" à temps.

Enfin, il est intéressant d'ajouter deux fonctionnalités, certes optionnelles, mais qui améliorent beaucoup l'ergonomie du logiciel. La première consiste à permettre au joueur de **quitter la partie**. En effet, un joueur peut avoir envie de quitter la partie pour revenir sur le menu principal. De plus, nous souhaitions aussi permettre au joueur de **redémarrer une partie**. Un joueur peut avoir envie de redémarrer une partie directement, sans passer par le menu principal.

# II) Diagramme de classes



Notre diagramme de classes met en avant les différentes classes que nous utiliserons lors de la programmation du logiciel Funo, ainsi que leurs relations. Chacune de ces classes montre le comportement d'un ensemble d'objets (instances) partageant des caractéristiques communes, tels que des responsabilités et des comportements.

Les constructeurs des classes ainsi que les accesseurs et mutateurs des différents attributs n'ont pas été ajoutés pour faciliter la lisibilité du diagramme. Ils seront bien entendu intégrés au projet final sous la forme de méthodes de visibilité publique. Pour les mêmes raisons, et aussi pour des raisons de convention, nous avons préféré ne pas intégrer la classe **Main** et les différentes classes composantes de la **Vue** et du **Contrôleur**.

Ce diagramme de classes contient 18 classes. Chacune d'entre elle mérite d'être expliquée pour éclaircir quelques points et expliquer certains de nos choix de solutions face à certains problèmes.

Notre classe principale est la classe **Partie**. Ce sera la première classe instanciée lors de la sélection d'une nouvelle partie via le menu principal, et c'est à partir d'elle que seront instanciées, en chaîne, toutes les autres classes. En effet, son instanciation va déclencher la création de 2 à 10 instances de la classe **Joueur** (en fonction du nombre de joueurs choisi lors de la sélection) et de 1 à une infinité d'instances de la classe **Manche**. Le fait que ce soient des instances qui soient créées est lié aux deux relations choisies, qui sont toutes les deux des compositions. Une partie contient bien des manches, et il nous semblait impossible qu'une manche puisse exister sans création préalable de partie, d'où le choix d'une composition (agrégation forte) plutôt que d'une agrégation faible. Une partie contient bien également des joueurs, mais le choix de la composition ne nous semblait pas aussi évident cette fois-ci. Si nous avions voulu permettre la création de joueurs via le menu, comme des sortes de comptes, il aurait fallu s'occuper d'une persistance des données. Dans ce cas-là, nous aurions utilisé une agrégation faible entre **Partie** et **Joueur**, puisqu'un joueur pourrait exister et durer avant même toute création de partie. Enfin, nous avons ajouté des attributs "gagnant", "modeJeu" et "modeAttribution" à notre partie, qui permettront respectivement d'afficher le joueur gagnant en fin de partie, de choisir le mode de jeu ("Chacun pour tous", "A 2 joueurs", "Jeu en equipes" ou "Uno challenge") et de choisir le mode d'attribution des points ("PointsPositifs" ou "PointsNegatifs"). Nous avons également ajouté des méthodes "commencer()" et "terminer()" pour respectivement commencer une partie et afficher un message une fois la partie terminée.

Une partie est donc composée de manche(s), qui ont chacune des attributs gagnant, numeroTour et sens. Le premier permet de connaître le gagnant du tour quand celui-ci est sur le point de se terminer et qu'il reste l'attribution des points, le second permet de faciliter le passage des tours entre les joueurs grâce à un système de modulo et d'ajout/retrait d'un ou deux "points", sachant que ce qui détermine le fait que ce sera un ajout ou un retrait qui sera effectué est fonction du troisième, qui correspond de la partie, par défaut à 0 (= le prochain joueur est le joueur à gauche du joueur actuel). Un seul point est ajouté/retiré en cas de simple passage au prochain joueur, deux points sont ajoutés ou retirés en cas d'utilisation des cartes "+2", "+4" ou "Passer un tour" (voire "Inversion" dans le cas du mode "Deux joueurs") qui font passer son tour au prochain joueur. La classe Manche comporte également deux méthodes : "seJouer()" et "attribuerPoints(String)". La première se déclenche quand la manche commence et continue tant qu'un joueur ne l'a pas gagné. La seconde se déclenche une fois qu'un joueur l'a gagné, les points sont ensuite attribués en fonction de la méthode d'attribution choisie en début de partie (points "positifs" ou points "négatifs"). Une manche est composée de 108 cartes et chacune des cartes ne peut être présente que dans une seule manche. Une manche est également composée de 2 tas de cartes et chacun de ces tas ne peut être présent que dans une seule manche. Le choix de l'agrégation faible pour la première relation et de l'agrégation forte (composition) pour la seconde s'explique. Il nous semblait possible que les cartes puissent exister indépendamment d'une manche, mais pas que les tas de cartes (la pioche et le talon) puissent exister indépendamment d'une manche.

Venons-en justement aux cartes. La classe **Carte** est mère de deux classes (**CarteNormale** et **CarteSpeciale**) et "grand-mère" de cinq autres classes (**Carte+2**, **CarteInversion**, **CartePasserUnTour**, **CarteJoker** et **Carte+4**). La classe **Carte** contient les attributs couleur et couleurVariable. Le premier correspond à la couleur de la carte, et la deuxième indique si la couleur de la carte peut varier. En effet, deux cartes spéciales, "Joker" et "+4", permettent au joueur qui les utilisent de choisir la couleur qu'elles prendront une fois sur le dessus du talon. De plus, la classe **Carte** comporte également la méthode abstraite "estJouable()". Ainsi, la classe **Carte** est une classe abstraite et la méthode "estJouable()" sera redéfinie dans les classes filles et "petites-filles". Cette méthode indiquera si la carte est jouable en fonction de la dernière carte du talon et des propriétés de la carte (une carte à couleur variable sera ainsi jouable à tout moment, même si l'une des deux peut donner lieu à des pénalités).

Nous pouvons maintenant aborder les deux classes filles de la classe **Carte**. La classe **CarteNormale** contient un attribut "numero" et une méthode "estJouable()". L'attribut permet de connaître le numéro de la carte, étant donné que seules les cartes normales sont numérotées, et la méthode permet de savoir si elle est jouable (la carte sur le dessus du talon a-t-elle la même couleur ou le même numéro ?). La classe **CarteSpeciale**, quant à elle, ne contient que la méthode abstraite "estJouable()", ce qui fait que la classe **CarteSpeciale** est une classe abstraite.

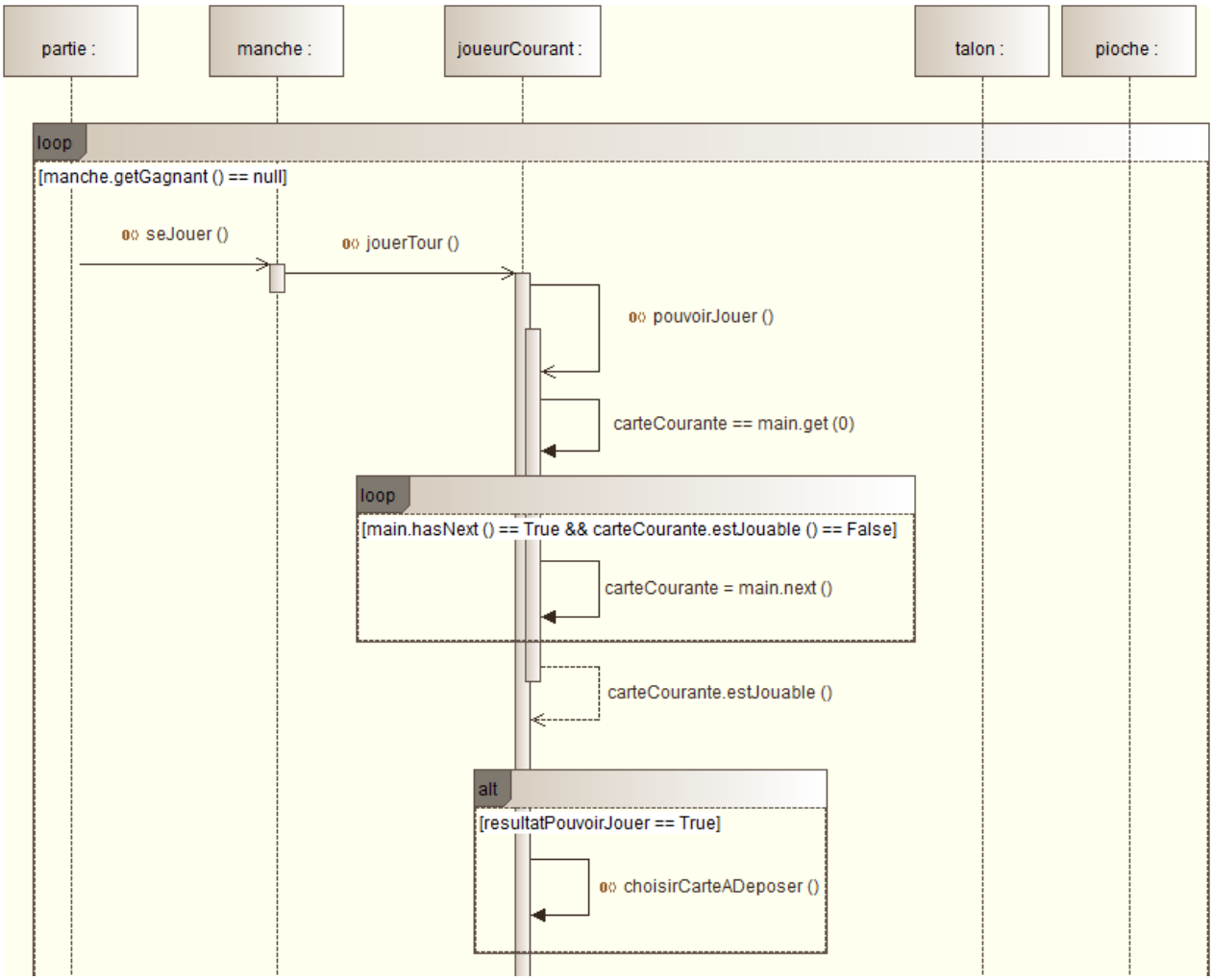
Restent les cinq classes "petites-filles" de la classe **Carte**. Chacune de ces classes n'a qu'une seule méthode, néanmoins très importante : "estJouable()". En effet, chacun des types de cartes spéciales redéfinit la méthode "estJouable()", car chacun des types de cartes spéciales est jouable sous des conditions assez variables, conditions d'autant plus variables entre les types de cartes spéciales présents 8 fois dans le jeu ("+2", "Inversion", "Passer un tour") et les types de cartes spéciales présents 4 fois dans le jeu ("Joker", "+4").

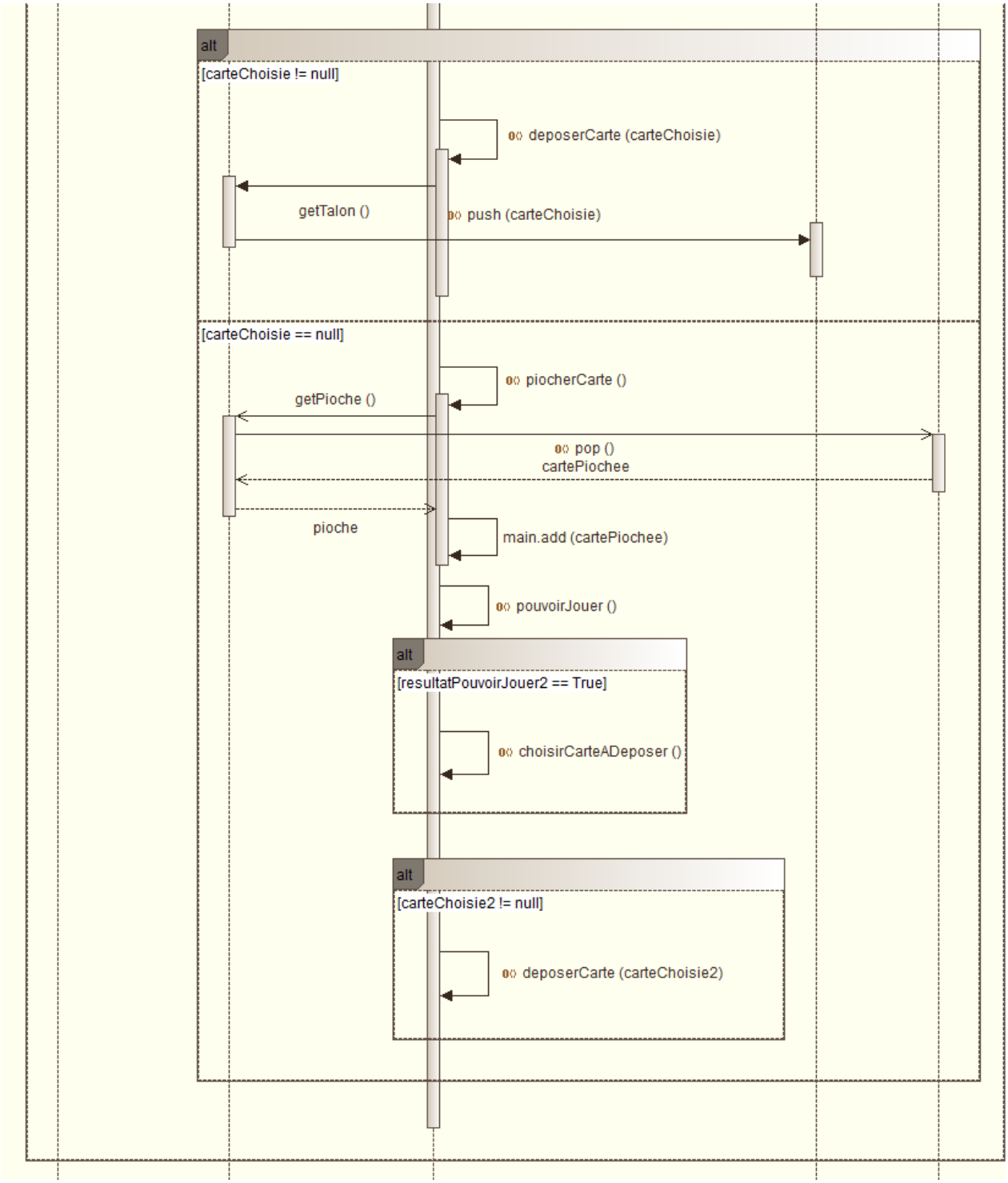
Passons maintenant à la classe **TasDeCartes**. Celle-ci contient une méthode : “seMelanger()”. Cette méthode permettra de mélanger un tas de cartes avec la méthode Collections.shuffle(Collections). Il peut servir pour, par exemple, mélanger la pioche avant de distribuer 7 cartes à chaque joueur, en début de partie ; ou mélanger le talon pour en faire une nouvelle pioche, quand la pioche n’a plus de carte. La classe **TasDeCartes** hérite en effet de la classe **Stack**, qui est intégrée au JDK Java. Cette classe correspond à une pile classique, donc suit l’ordre LIFO (Last In, First Out). Cela revient exactement à ce que sont une pioche ou un talon physiques. La pioche utilisera principalement “pop()” pour dépiler des cartes qui arriveront directement dans la main d’un joueur, tandis que le talon utilisera principalement “push(Object)” pour empiler des cartes qui viendront directement de la main d’un joueur. De plus, “top()” pourra être utilisé pour comparer la carte sur le dessus du talon avec les cartes de la main d’un joueur et “empty()” pourra être utilisé pour tester si la pioche est vide ou non.

Ainsi, la “partie **Manche**” des deux compositions de la classe **Partie** a été expliquée. Seulement, il existe aussi la “partie **Joueur**”. La classe **Joueur** possède les attributs “nom”, “score”, “main” et “place”. En effet, un joueur possède un nom, qu’il choisit en début de partie, pour se distinguer des autres. Un joueur possède également un score, celui-ci permettant de connaître son positionnement parmi les autres joueurs. Un joueur possède enfin une main de cartes et une place, cette dernière correspondant à son positionnement au niveau de passage de tour d’un joueur à un autre. La classe **Joueur** comporte également plusieurs méthodes : “jouerTour()” se déclenche au début du tour d’un joueur et termine juste avant que le prochain joueur ne joue son tour, “piocherCarte()” permet de déclencher la pioche par le joueur d’une carte dans la pioche, “deposerCarte(Carte)” permet de déclencher la pose par le joueur d’une carte sur le talon, “pouvoirJouer()” indique si le joueur peut jouer ou non, “choisirCarteADeposer()” est une méthode abstraite qui sera redéfinie dans les classes filles (un joueur humain et un joueur virtuel ne choisissant pas de la même manière) et qui permettra au joueur de choisir la carte qu’il souhaite déposer, enfin, “distribuerCartes(ArrayList <<Carte>>)” permet à un joueur de distribuer les cartes en début de partie. La méthode “choisirCarteADeposer()” étant abstraite, la classe **Joueur** est aussi abstraite. Il est donc logique que la classe **Joueur** ait des classes filles. Ses deux classes filles sont **JoueurHumain** et **JoueurVirtuel**.

Enfin, parlons de ces classes filles. La classe **JoueurHumain** concerne les joueurs humains, et comprend donc un attribut “prenom” permettant ainsi d’afficher tout à la fois le prénom et le nom d’un joueur s’il est humain. La classe **JoueurHumain** comporte également “choisirCarteADeposer()” redéfinie de telle sorte à ce que le joueur puisse saisir la carte qu’il souhaite déposer, si, bien sûr, il souhaite en déposer (pouvant aussi passer son tour). La classe **JoueurVirtuel**, quant à elle, est un peu particulière. En effet, elle est vide, mais est liée par une agrégation faible à l’interface **Strategie** qui comporte la méthode “choisirCarteADeposer()”. Cela est dû à l’utilisation du **patron de conception “Strategy”**. Ce patron de conception permet de fournir un plan pour l’implantation de plusieurs stratégies à une intelligence artificielle, ici nommée joueur virtuel. Les deux stratégies choisies sont une stratégie offensive (= classe **StrategieOffensive**) et une stratégie défensive (= classe **StrategieDefensive**), dont les classes héritent de l’interface **Strategie**. Le **patron de conception “Strategy”** a l’avantage de pouvoir facilement attribuer une stratégie à une intelligence artificielle et de rendre la programmation modulatoire et flexible, au centre du paradigme de l’orienté objet.

# III) Diagramme de séquence




alt
[carteChoisie != null]
deposerCarte (carteChoisie)
getTalon ()
push (carteChoisie)
[carteChoisie == null]
piocherCarte ()
getPioche ()
pop ()
cartePiochee
pioche
main.add (cartePiochee)
pouvoirJouer ()
alt
[resultatPouvoirJouer2 == True]
choisirCarteADeposer ()
alt
[carteChoisie2 != null]
deposerCarte (carteChoisie2)

Le diagramme de séquence sert à décrire les différents enchaînements entre classes et les utilisations de méthodes au fur et mesure de l'exécution du programmation. C'est une sorte de pré-programmation, où la structure générale de la programmation est déjà visible.

Dans notre diagramme de séquence, nous avons préféré représenter le déroulement complet de certaines fonctions uniquement une fois, pour plus de clarté et de concision. C'est pourquoi **pouvoirJouer()**, **piocherCarte()**, et **deposerCarte(Carte)** ne sont décrits jusqu'au bout qu'une seule fois. Ce diagramme de séquence, bien qu'assez global, ne comporte pas également quelques cas précis (tels que les différences entre modes de jeu, entre types de cartes spéciales qui seraient testées via un instanceof juste après le dépot, et au niveau du passage des tours d'un joueur à un autre qui serait réalisé grâce à un ensemble entre les attributs correspondants de Manche et de Joueur et l'utilisation de modulo).

Le diagramme de séquence commence par une boucle qui sert à faire continuer la manche tant qu'elle n'est pas terminée, c'est-à-dire tant qu'un joueur n'a pas gagner la manche. Ainsi, la partie fait appel à la méthode **seJouer()** de la manche actuelle après l'avoir instancié (pas affiché dans le diagramme de séquence puisque non directement lié au déroulement d'un tour de jeu). La manche fait ensuite appel à la méthode **jouerTour()** du joueur courant.

On teste alors si ce joueur peut jouer grâce à la méthode **pouvoirJouer()**. Pour cela **pouvoirJouer()** déclare une variable de type Carte, carteCourante, et lui attribue la valeur qui correspond à la première carte de l'ArrayList main (**main.get(0)**), c'est-à-dire la main du joueur. Ensuite, une boucle se déclenche : tant que la main du joueur contient une carte à la position suivante de liste (**main.hasNext()**) et tant que la carte courante n'est pas jouable (**carteCourante.estJouable()** == False), la carte courante devient la prochaine carte de la main (carteCourante = **main.next()**). Une fois que la main est entièrement balayée ou qu'une carte jouable a été trouvée, le programme sort de la boucle et renvoie la jouabilité de la carte courante (**carteCourante.estJouable()**), qui n'est pas forcément la dernière carte de la main.

Une fois la possibilité de jouer testée, on en utilise le résultat dans une condition. Si celle-ci est vraie, donc si le joueur peut jouer, il a le droit de choisir une carte de sa main pour la déposer sur le talon.

Ensuite, une autre condition s'ouvre, et deux cas se présentent à nous :

- Soit la variable correspondant à la carte choisie n'a pas la valeur "null", c'est-à-dire qu'une carte de la main a été choisie pour être déposée sur le talon. Dans ce cas, la méthode **deposerCarte (carteChoisie)** est appelée, puis la méthode **getTalon()** à partir de la manche actuelle, et enfin on empile la carte choisie sur le talon grâce à la méthode héritée de Stack **push(carteChoisie)**. Vient alors le tour du prochain joueur.

- Soit la variable correspondant à la carte choisie a la valeur "null", c'est-à-dire que soit aucune carte ne pouvait être jouée, soit aucune carte n'a été choisie dans la main. Dans ce cas, la méthode **piocherCarte()** est appelée, puis la méthode **getPioche()** à partir de la manche actuelle, et enfin on dépile la carte sur le dessus de la pioche grâce à la méthode héritée de Stack **pop()**. La valeur de la carte piochée remonte alors jusqu'au contenu de la méthode **piocherCarte()**, et la carte piochée est ajoutée à la main grâce à la méthode **main.add(cartePiochee)**. Ensuite est appelée à nouveau la méthode **pouvoirJouer()**, puis en fonction du résultat, est proposée ou non de choisir une carte avec la méthode **choisirCarteADeposer()**. Si le joueur peut jouer une carte et si une carte est choisie, celle-ci est alors déposée sur le talon avec la méthode **deposerCarte(carteChoisie2)** et c'est au tour du joueur suivant.

# IV) Modélisation UML finale

Les diagrammes conçus lors de la première phase du projet ont évolué au fur et à mesure de celui-ci. En effet, une première modélisation ne nous avait permis qu'une vue partielle de l'ensemble, et de nombreux points ont gagné à être revus.

**Le diagramme de cas d'utilisation a bien été le seul à ne pas du tout changer du début à la fin du projet.** En effet, les tâches réalisables au sein du programme avaient été déjà bien perçues au début du projet, même si elles gagnaient à être vues davantage en profondeur.

**Quant au diagramme de classes, c'est celui qui a vécu le plus de changements et d'adaptations à des situations nouvelles dont nous avons appris l'existence au cours du projet.** Celui-ci étant trop grand, il n'a pas pu être inséré dans le rapport. C'est ainsi qu'il peut être retrouvé en fichier indépendant, avec le rapport. Trois classes, en particulier, se sont vues largement gagner en taille : Partie, Manche et Joueur. Entre autres, plusieurs méthodes abstraites ont été insérées au sein de la classe abstraite Joueur : *choisirCouleurCarte (in listeCouleurs : ArrayList<<String>>, in situation : integer) : String* et *saisirUno() : boolean* tout particulièrement. En effet, ces méthodes gagnaient à être redéfinies dans les classes filles de la classe abstraite Joueur : JoueurHumain et JoueurVirtuel. Un joueur humain ne choisira pas la couleur de sa carte de la même manière qu'une intelligence artificielle ; comme il ne choisira pas une carte dans sa main ou ne saisira pas « UNO » de la même manière. Des méthodes de saisie et d'affichage ont également été ajoutés. Cela est dû au fait que la modélisation nous avait donné une vue générale du logiciel à réaliser ; et que sa réalisation, justement, nous a permis de creuser et de développer ce qui avait été fait dans la modélisation. Des méthodes ont ainsi gagné des sous-méthodes. Ainsi, pas moins de quatre méthodes sont réservées au donneur ; là où seule une existait dans la modélisation initiale.

**Le diagramme de séquences, quant à lui, a été très fortement détaillé.** Déjà dans la version initiale, le diagramme devenait difficile à lire de par sa complexité et son détail. Le nouveau diagramme de séquences n'a donc pas été réalisé, celui-ci étant assez imposant ; et le temps ne nous ayant pas permis de vous en présenter une version. Dans l'ensemble, le principe de boucle pour représenter l'enchaînement des tours des joueurs a été conservé, celui-ci ayant pour condition d'arrêt l'inexistence de cartes dans la main du joueur.

Il en ressort que globalement, les diagrammes initialement conçus ont été complètement changés, bouleversés dans ce qu'ils étaient à la base ; mais globalement, ils conservent la même structure générale, et ces diagrammes nous auront également permis de mieux nous guider au fur et à mesure de la réalisation du projet.

# V) État actuel de l'application

À l'heure actuelle, l'application comporte une grosse partie des fonctionnalités qui avaient été demandées. Pour présenter clairement ce qui a été fait et ce qui reste à faire, nous préférons distinguer le mode console et le mode graphique.

Au niveau du mode console :

- La **sélection des paramètres d'une partie** (nombre de joueurs, noms des joueurs, prénoms des joueurs humains, stratégies des joueurs virtuels, mode d'attribution de points et mode de jeu) se réalise sans problème.

- En début de partie, **l'ordre des joueurs est attribué aléatoirement** grâce à la méthode « shuffle ». Le donneur est alors choisi et doit appuyer sur la touche « Entrée » pour démarrer le mélange des cartes, la distribution de celles-ci aux joueurs, et la création de la pioche et du talon.

- La **sélection d'une carte** par un joueur humain peut être effectuée. Un pré-affichage de la main est également réalisée pour faciliter le choix d'une carte par un joueur humain, et seules les cartes jouables peuvent être choisies. En effet, si une carte non-jouable est choisie, un message demande à l'utilisateur de choisir une autre carte. Si aucune carte n'est jouable, le joueur passer automatiquement la première partie de son tour. S'il pioche une carte jouable ensuite, il peut décider de la jouer, le test de jouabilité de la main d'un joueur s'effectuant au début des deux parties du tour d'un joueur, si deuxième partie il y a. Un joueur virtuel choisira, quant à lui, des cartes plutôt offensives si sa stratégie est offensive (+4, +2, PasserUnTour), ou des cartes plutôt défensives si sa stratégie est défensive (Joker, Inversion, Normale).

- Les **différents effets des cartes spéciales fonctionnent**. Dans le cas de l'utilisation d'une carte +4 ou Joker, si le joueur est humain, le choix d'une nouvelle couleur de carte se réalise via une saisie ; si le joueur est virtuel, le choix d'une nouvelle couleur de carte se réalise en se basant sur la fréquence de ses cartes en main. Ainsi, le joueur virtuel choisira la couleur de carte qui est la plus présente dans sa main.

- Une **manche peut se jouer** du début de la manche jusqu'à la fin sans aucun bug.

- Une **partie peut se jouer** du début à la fin sans problème, avec attribution des points à la fin de chaque manche.

- Le **bluff résultant du dépôt d'une carte +4** a été implanté et fonctionne également.

- La **prise en compte du UNO** suite au dépôt de l'avant-dernière carte a également été réalisée. Pour ce faire, nous sommes partis du principe qu'un joueur humain devait saisir « UNO » ou « uno » dans les cinq secondes suivant son dépôt et valider sa réponse. Cette règle est explicitement introduite en début de manche. Un joueur virtuel répondra toujours « UNO » à temps.

- **Deux modes d'attribution de points sont proposés** : positif et négatif. Si le mode positif est choisi, les points seront « valorisants », c'est-à-dire que le joueur gagnant une manche gagnera des points en se basant sur la valeur des cartes des mains de ses adversaires. Si le mode négatif est choisi, les points seront « dévalorisants », c'est-à-dire qu'une fois une manche gagnée par un joueur, les autres joueurs gagneront le nombre de points correspondant aux cartes de leur main.

- **Trois modes de jeu sont proposés** : traditionnel, à 2 joueurs et Uno Challenge. Nous sommes partis du principe que le mode à 2 joueurs était automatiquement intégré au sein du mode traditionnel, si le nombre de joueurs était de deux. Il est également automatiquement intégré pour la dernière manche du mode Uno Challenge. Le mode Uno Challenge, quant à lui, ne peut être joué qu'en mode d'attribution de points négatif. En effet, dans ce mode, un joueur est éliminé quand il atteint 500 points.

Au niveau du mode graphique :

- La **sélection des paramètres d'une partie** (nombre de joueurs, noms des joueurs, prénoms des joueurs humains, stratégies des joueurs virtuels, mode d'attribution de points et mode de jeu) se réalise via une fenêtre de menu. L'utilisateur peut alors choisir le mode de jeu via une liste déroulante et le mode d'attribution de points via des boutons radios. De plus, il peut saisir les différentes caractéristiques d'un joueur dans une zone prévue pour cet effet, visible à gauche de la fenêtre. Lors de la validation de l'ajout du nouveau joueur, le programme vérifie que le nom n'existe pas déjà dans la liste, que le nom est saisi et que le prénom est également saisi dans le cas d'un joueur humain. Si l'une de ces conditions n'est pas remplie, un message d'erreur s'affiche et prévient l'utilisateur, lui demandant de changer ce qui ne correspond pas dans ses saisies, s'il souhaite ajouter un nouveau joueur. Une fois un nouveau joueur ajouté, le nom de celui-ci apparaît dans la liste de sélection de joueurs centrale. L'utilisateur peut alors sélectionner le nouvel arrivant, ce qui provoque l'affichage de ses données dans une zone prévue à cet effet dans la partie droite de la fenêtre. Il peut alors décider de modifier ses données et de valider les nouvelles saisies. Les mêmes vérifications que pour un nouveau joueur sont appliquées lors de la validation des modifications. Un joueur sélectionné peut également être supprimé de la liste par une validation sur un bouton poussoir de suppression. Enfin, une fois les choix réalisés et les saisies terminées, l'utilisateur peut lancer une partie. Ici encore, des réalisations sont effectuées : le joueur ne peut pas lancer de partie si le nombre de joueurs présents dans la liste de sélection est inférieur à 2 ou supérieur à 10. Les données de la partie sont alors mises à jour, la fenêtre de menu disparaît et la fenêtre de partie apparaît.

- La **fenêtre de partie ne montre pas de réponse interactionnelle**. En effet, si d'un côté, l'affichage de la fenêtre et de ses composants graphiques se réalise bel et bien, les interactions entre l'utilisateur et l'interface ne sont pas possibles arrivé à ce point. C'est donc la grande faille de notre application, et la partie qu'il restait à y réaliser.

- Il aurait pu également y avoir une **intégration d'écriture et lecture** dans un fichier, et donc de la sauvegarde durable de la partie, en utilisant l'interface Serializable que nous avions commencer à manipuler.

# Conclusion

Le projet pourrait être encore approfondi, et la partie graphique terminée. Un système d'écriture/lecture dans un fichier pourrait être mis en place grâce à l'interface Serializable ; et le projet pourrait également s'étendre vers une vision réseau du fameux jeu de cartes, sur la base de sockets. Néanmoins, malgré le manque de temps et le caractère incomplet de notre projet, le cahier des charges a été globalement rempli, et il suffirait de quelques jours supplémentaires pour fournir une interface graphique agréable à utiliser, complète et surtout fonctionnelle ; l'interface console remplissant déjà ces conditions.

Ce projet nous aura permis de suivre le développement d'un logiciel avec le langage de modélisation UML et le langage Java. Cela aura été enrichissant à plusieurs niveaux. Déjà, le projet nous aura appris à mieux nous organiser et nous structurer dans la programmation d'un logiciel, grâce à l'utilisation de méthodes. De plus, le format du projet ressemble dans les grandes lignes à ce qui pourrait nous être demandé en entreprise : travail en équipe, date butoir, modélisation précédent la réalisation. Également, l'utilisation d'UML et de Java nous aura permis de renforcer nos notions et nos compétences dans ces deux langages : l'un de modélisation, l'autre de programmation.

Au final, il est fortement probable que nous réutilisions à l'avenir ce que nous avons appris dans cette unité de valeur. Cela est d'autant plus vrai que le Java se dispute régulièrement la première place des langages les plus utilisés à l'heure actuelle. LO02 nous aura ainsi fourni des compétences à même d'être utilisées régulièrement, aussi bien dans notre travail que dans nos loisirs.