Binôme :
Ali Quenahat
Victor Paynat-Sautivet
Niveau : *ISI1*

LIVRABLE 3

Projet LO02

La Bataille Norvégienne

UV : LO02
Responsable : Guillaume Doyen
Semestre : Automne 2014

www.utt.fr
Université de technologie de Troyes • 12 rue Marie Curie • BP 2060 • 10010 Troyes cedex

Sommaire

INTRODUCTION

Pendant notre cursus à l'Université de Technologie de Troyes, l'UV LO02 nous a été proposé au cours du semestre d'automne 2014 en tant que TM de branche. Nous l'avons choisi afin d'appréhender la notion d'orienté objet.

Dans le cadre de cette UV, nous avons l'opportunité de développer une application à l'aide du langage Java et l'outil de développement Eclipse ayant pour but de se faire une expérience dans le développement orienté objet en passant par plusieurs étapes de conception.

Notre projet est de programmer le jeu de la Bataille Norvégienne. Nous vous proposons une présentation du projet vu de notre œil.

Chaque joueur dispose de 9 cartes, dont 3 cachées, 3 visibles et 3 tenus dans la main. A cette étape du jeu, il est possible d'échanger les cartes tenues dans la main avec celles qui sont visibles. Le but étant d'avoir des cartes fortes ou alors des cartes spéciales.

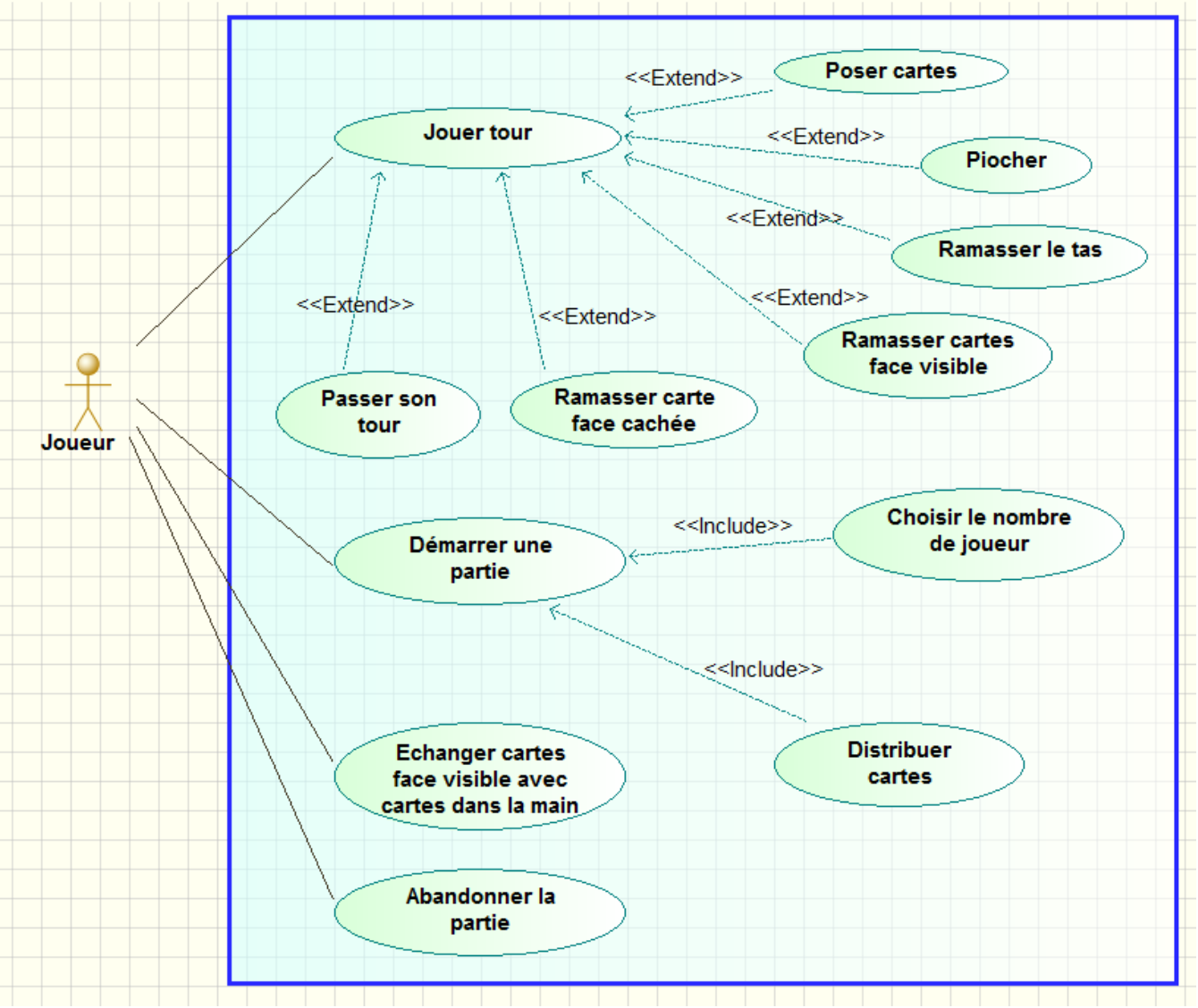
Celui qui commence, est le joueur qui se situe à gauche du donneur, il peut poser jusqu'à 3 cartes pourvues qu'elles aient la même valeur. Il doit piocher autant de cartes qu'il a posés tant que la pioche n'est pas vide. Le deuxième joueur doit poser une (ou plusieurs de même valeur) carte supérieure ou égale à celles posés par le premier joueur. Il en est de même pour les joueurs qui suivent. Lorsqu'un joueur ne dispose plus de cartes assez fortes, il devra ramasser tout le tas. Lorsque la pioche est vide, les joueurs devront vider leurs mains pour ramasser les 3 cartes visibles. Ensuite, au moment ou le joueur n'a plus de cartes, il doit piocher aléatoirement une carte parmi celles qui sont retournées. Dans le cas où la carte qu'il pioche n'est pas jouable, il doit ramasser le tas, et ne pourra ramasser une des cartes devant lui que lorsqu'il aura éliminé les cartes qu'il avait en main.

Dans ce jeu, il y a plusieurs cartes spéciales :
10 : Cette carte retire les cartes du tas et ne pourront être réutilisés.
7 : Elle impose au joueur suivant de jouer une carte inférieure ou égale à 7.
2 : Déposable sur n'importe qu'elle carte, elle permet de relancer le jeu à partir d'un 2.
8 : Le joueur qui suit doit passer son tour.
As : Elle donne la possibilité d'envoyer le tas à un joueur de notre choix. Seul un autre As ou un 2 peuvent le contrer.

Ce sera donc le premier joueur qui n'aura plus de cartes qui sera alors le gagnant.
Dans ce rapport nous vous proposons une modélisation UML du projet, dans un premier temps nous étudierons le diagramme de cas d'utilisation puis le diagramme de classe et enfin le diagramme de séquence.

I. Diagramme de cas d'utilisation

1. Le diagramme



2. Explications

Le joueur peut effectuer 4 actions principales :

- « Jouer tour » : Permet au joueur de jouer son tour en ayant plusieurs options qui sont « Passer son tour », « Ramasser cartes face cachée », « Ramasser cartes face visible », « Ramasser le tas », « Piocher », « Poser cartes ». C'est pourquoi nous avons utilisé la relation « extend »

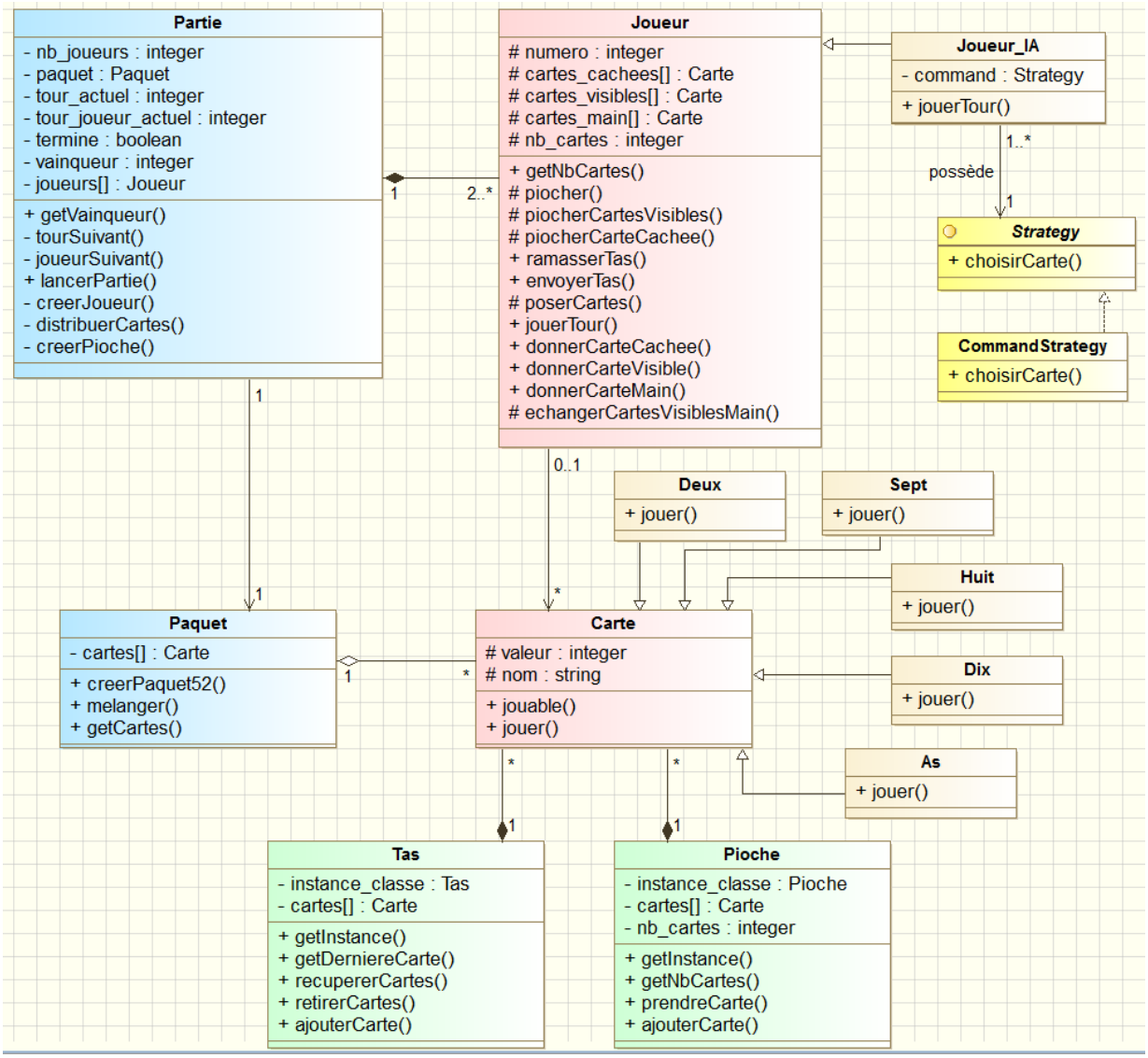
- « Démarrer une partie » : Pour démarrer une partie, le joueur doit choisir le nombre de joueur et il doit aussi distribuer les cartes. Ce qui explique pour nous avons utilisés la relation « include ».

- « Echanger cartes face visibles avec cartes dans la main » : Le joueur peut échanger ses cartes dans la main avec les cartes face visibles.

- « Abandonner la partie » : S'il le souhaite, le joueur peut abandonner la partie.

II. Diagramme de classe

1. Le diagramme



2. Explications

Nous avons décidé de créer les classes principales suivantes : Partie, Joueur, Paquet, Carte, Tas et Pioche. Nous avons des classes qui héritent comme Joueur_IA qui hérite de Joueur et qui modifie la méthode jouerTour(), ou encore les cartes spéciales, représentées par les classes Deux, Sept, Huit, Dix et As. Ces classes ajoutent un effet spécial sur la partie.

Une interface a été pensée, l'interface Strategy, qui se chargera de faire choisir aux joueurs virtuels la carte à jouer en fonction du Tas et des autres joueurs. Cette interface sera implémentée par une seule classe, CommandStrategy. Nous avons choisi de ne développer qu'une seule classe pour implémenter l'interface, le sujet ne

demandant pas plusieurs niveaux de difficulté pour les joueurs virtuels. Cette interface déterminera à quel joueur est envoyé le Tas lorsqu'un joueur virtuel pose un As, ou encore quelle carte poser. Le choix d'utiliser une interface est motivé par l'idée de pouvoir créer un jour, si on souhaite améliorer le programme, d'autres niveaux de difficulté, ce qui sera possible et facilité par l'interface.

La classe Partie sera la classe centrale du programme. Le "main" créera un objet Partie et appellera la méthode lancerPartie(), et tout se fera ensuite dans la classe Partie. C'est à dire que dans lancerPartie(), on demandera à l'utilisateur de choisir le nombre de joueurs, on créera le paquet de cartes, le tas, la pioche, les joueurs, puis on distribuera les cartes et à ce moment là on appellera la méthode echangerCartesVisiblesMain() pour demander au joueur s'il veut changer les cartes de sa main avec les trois cartes visibles qui sont devant lui. Une fois cela fait, le premier tour débutera simplement par l'appel de tourSuivant(). Nous avons voulu faire de Partie la classe qui se chargerait de gérer un peu tout en appelant les méthodes des autres classes car cela permet de centraliser les commandes principales. Du coup, si le joueur veut refaire une partie, cela se passera dans le "main", et tout simplement, un nouvel objet Partie sera créé. De la même manière, lorsque la partie est finie, le résultat (nom du vainqueur) est affiché depuis le "main" et c'est à ce moment qu'il est proposé au joueur de refaire une partie. Certaines méthodes sont en visibilité "private" car elle n'ont pas besoin d'être appelées depuis l'extérieur, c'est uniquement Partie qui appelle ses propres méthodes pour par exemple changer de tour, ou bien distribuer les cartes. Ce choix de visibilité est ici plus une sécurité, pour éviter de perturber la partie depuis le "main".

La classe Joueur est composée de nombreuses méthodes car les joueurs peuvent effectuer de nombreuses actions. En fonction de l'avancement de la partie, le joueur devra soit piocher dans la Pioche, soit les 3 cartes visibles devant lui, soit dans les 3 cartes cachées. On a mis la visibilité de certaines méthodes en "protected" car elles doivent être accessibles depuis la classe fille Joueur_IA dans le cas d'un joueur virtuel. D'autres fonctions sont en "public" car elles ont besoin d'être appelé depuis l'extérieur. Par exemple, lorsqu'un joueur pose un As, la méthode envoyerTas() est appelée depuis Partie, et il est demandé au joueur de choisir un adversaire à qui donner le tas de cartes. Lorsque le joueur a choisi son adversaire, il faut pouvoir appeler ramasserTas() sur l'objet qui correspond à l'adversaire, donc la visibilité doit être publique. De la même façon, lors de la distribution des cartes, on doit pouvoir donner des cartes au joueur et donc appeler donnerCarteCachee(), donnerCarteVisible() et donnerCarteMain() sur le joueur, depuis la classe Partie.

La classe Carte représente bien sûr chaque carte du jeu, et possède un attribut qui indique la valeur de la carte. De cette manière, un valet vaut 11, un as vaut 14... Cela nous permettra de faire une condition très simple pour savoir si une carte peut être jouée par dessus le tas dans la méthode jouable() qui renvoie un booleen.

La classe Paquet correspond au paquet complet des cartes qui seront utilisées lors de la partie. C'est à dire qu'en fonction du nombre de joueurs, un certains nombre de paquets de 52 cartes seront créés par cette classe et les cartes seront stockées dans le taleau cartes[]. Ensuite, la classe Partie distribura les cartes aux joueurs en les récupérant à l'aide de la méthode getCartes(). Ensuite, la Pioche sera créée et les cartes restantes lui seront ajoutées. Le Tas sera constitué au fur et à mesure que les joueurs posent leur cartes en appelant la méthode ajouterCarte(). La pioche elle, sera petit a petit vidée lorsque les objets instanciant la classe Joueur appeleront la méthode prendreCarte() de l'entité Pioche.

Les classes Tas et Pioche sont des singletons, il n'existera dans le programme qu'une seule instance de ces deux entités. Cela nous permettra d'une part de nous assurer qu'il existe bien une seule pioche et un seul tas, et d'autre part il sera possible depuis n'importe quelle classe, de récupérer la dernière carte posée sur le tas par exemple, ou alors de piocher une carte pour les joueurs, sans avoir besoin de stocker une instance du tas et de la pioche en attribut. Le patron de conception singleton s'est imposé comme une évidence en ce qui concerne ces deux classes.

RELATIONS ENTRE LES CLASSES

Partie - Paquet : il existe une association car Partie possède un attribut Paquet (pour pouvoir distribuer ensuite les cartes)

Partie - Joueur : relation de composition. Une partie est composée de plusieurs joueurs. (2 ou plus) Et un joueur n'est présent que dans une partie (car à la fin de la partie, l'objet Partie est supprimé et un nouveau est créé, avec de nouveaux objets Joueur).

Joueur - Joueur_IA : Joueur_IA hérite de Joueur.

Joueur_IA - Strategy : Joueur_IA possède une stratégie de jeu, qui va être gérée par la classe CommandStrategy, qui implémente l'interface Strategy. Une instance de CommandStrategy est stockée en attribut de Joueur_IA.

Joueur - Carte : association car chaque joueur possède plusieurs tableaux de Carte. En revanche une carte peut appartenir à un joueur, ou alors à aucun joueur et se situer dans le Tas ou la Pioche.

Paquet - Carte : relation d'aggrégation. Le paquet est constitué de plusieurs cartes, mais le paquet partage les cartes avec d'autres objets comme Joueur, Tas et Pioche. Une carte ne fait partie que d'un seul paquet.

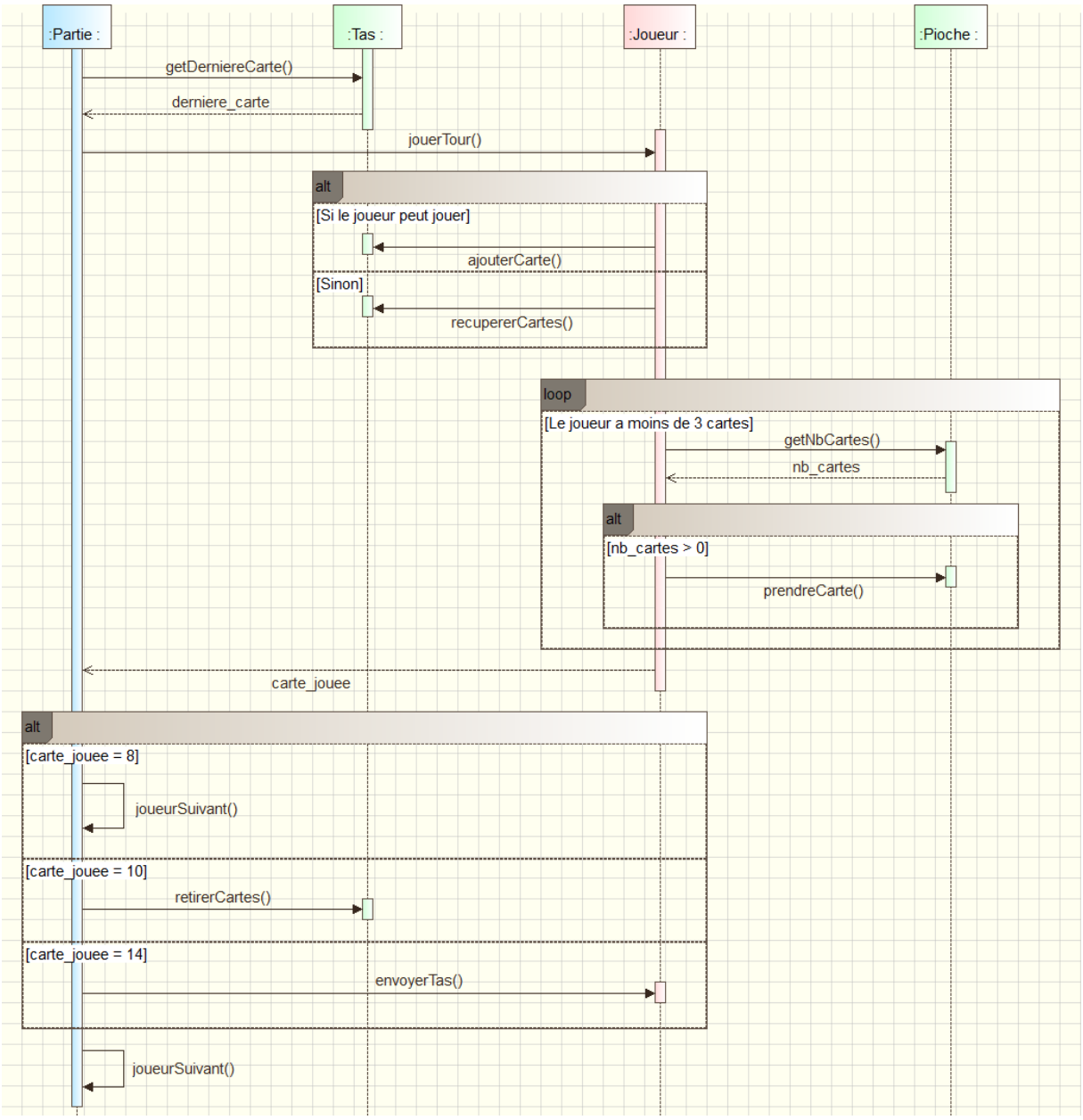
Tas - Carte : relation de composition. Un tas est composé de plusieurs cartes, mais si la carte se trouve dans le tas, alors elle ne peut être nulle part ailleurs, donc pas de partage.

Pioche - Carte : la pioche est composée de plusieurs cartes, mais si la carte se trouve dans la pioche, alors elle ne peut être nulle part ailleurs.

Carte - Deux,Sept,Huit,Dix,As : ces classes héritent de Cartes, elle modifient seulement l'effet de la carte sur la partie.

III. Diagramme de séquence

1. Le diagramme



2. Explications

Notre diagramme de séquence présente le déroulement d'un tour "classique" du jeu de la Bataille Norvégienne.
Nous avons choisi de représenter 4 entités : Partie, Tas, Joueur et Pioche. Ces quatre entités correspondent aux objets principaux qui interagissent entre eux pendant un tour du jeu. Pour rappel, la classe Tas et la classe Pioche sont des singletons, par conséquent nous ne manipulons qu'une seule instance pendant toute la partie.
La classe Partie étant celle qui se charge de gérer complètement les tours du jeu, c'est elle qui va appeler progressivement les méthodes des autres classes pour faire les actions.

Pour pouvoir jouer son tour, le joueur a besoin de connaitre la dernière carte qui a été posée. C'est la classe Partie qui va obtenir cette information avant de l'envoyer au joueur concerné, pour garder le principe que c'est l'objet Partie qui gère le tour. Nous avons choisi de faire comme cela car nous avons ainsi la possibilité de centraliser les actions dans la classe Partie, qui comme son nom l'indique doit gérer la partie. Les objets de type Joueur auraient pu aussi appeler la méthode getDerniereCarte() directement, mais nous voulions que tout soit visible et accessible de la classe Partie, vu que c'est elle qui affichera le texte dans la console et la fenêtre.
Ensuite, un appel à la méthode jouerTour() est fait depuis la classe Partie, pour lancer le processus de choix de la carte à jouer. La dernière carte du tas est transmise à la classe Joueur, pour que celle-ci vérifie si le joueur peut poser une carte par dessus le tas. Nous avons utilisé une alternative sur le diagramme pour symboliser la condition d'exécution. Si le joueur possède une carte qui peut être posée, alors elle est posée sur le tas (et si c'est au joueur virtuel de jouer, l'interface Strategy se charge de choisir la carte en question), si en revanche aucune carte ne convient dans la main du joueur, ce dernier doit ramasser le tas. C'est donc l'entité Joueur qui appelle les méthodes poserCarte() ou ramasserTas() de l'entité Tas.
Après, en fonction du nombre de cartes posées, il faudra que le joueur pioche une ou plusieurs cartes. Nous avons modélisé cela par la boucle "loop" dans le diagramme. Cette boucle va s'exécuter tant que le joueur a moins de trois cartes en main. La méthode getNbCartes() est alors appelée sur la classe Pioche pour vérifier s'il reste des cartes dans la pioche. Si c'est le cas, alors Joueur appelle la méthode piocher() de Pioche.

Une fois que le joueur a fini son tour, c'est à dire poser sa ou ses cartes, ou bien ramasser le tas, puis piocher, alors on renvoie la valeur de la carte posée à Partie (0 si le joueur n'a rien posé). Ensuite, certaines vérifications sont faites pour que les cartes spéciales fassent leur effet. Une alternative est là encore utilisée. Si le joueur a posé un 8, alors le joueur suivant passera son tour et donc Partie appelle sur elle même la méthode joueurSuivant(). Autrement, si le joueur a posé un 10, alors la méthode retirerCartes() est appelée sur le Tas. Cette dernière méthode se charge de supprimer les objets cartes du tableau dans la classe Tas. Enfin, si le joueur avait posé un As (donc la valeur de carte est 14), alors la méthode envoyerTas() est appelée sur Joueur. De cette manière, le joueur aura la

possibilité de choisir l'adversaire auquel il souhaite envoyer les cartes qui composaient le tas. Encore une fois, nous avons choisi d'appeler les méthodes depuis la classe Partie, c'est vraiment elle qui sera au cœur de notre application.

Pour finir, la méthode joueurSuivant() est appelée afin de changer de joueur. Du coup, si un 8 a été posé, le joueur change dans "l'alternative", sur le diagramme, et change encore à la fin du diagramme de séquence. Ce qui permet de bien voir qu'un des joueurs ne peut pas jouer (son tour est passé automatiquement pendant le tour du joueur précédent).

En résumé, notre application possède une classe centrale, la classe Partie, qui se charge de faire le "maître du jeu" en appelant les méthodes des autres classes. Tout gravite autour de la classe Partie. C'est un choix personnel, il y a d'autres manières de faire, mais cela nous semble plus logique ou plus intuitif de centraliser les actions à cet endroit. De cette manière, on peut imaginer que la classe Partie appellera au fur et à mesure les actions des autres classes, et pourra en même temps afficher les informations de la partie à l'écran, ce qui permettra d'avoir les appels aux méthodes d'affichage au même endroit.

IV. Modélisation de la version finale

A propos de l'interface Strategy : nous n'avions pas bien pensé la chose lorsque nous avons rendu le livrable 1 donc nous avons complètement revu l'interface et le fonctionnement de celle-ci. En fait, les classes qui implémentent l'interface ne sont plus des “niveaux de difficulté” comme nous l'avions pensé, mais ce sont des classes qui correspondent chacune à une action que peut effectuer le joueur virtuel. Ce qui se passe, concrètement : dans la classe JoueurVirtuel, on va créer un objet de type Strategy qui va correspondre à la classe qui effectue l'action désirée, par exemple on va créer un objet “ChoisirNbCartesAPoser” (cette classe implémente l'interface Strategy donc est de type Strategy) et on va appeler la méthode “faire(JoueurVirtuel jv)” définit dans l'interface en passant en paramètre l'instance de la classe JoueurVirtuel qui correspond au joueur qui est en train de jouer. Toutes les classes qui implémentent l'interface Strategy se contentent donc de définir la méthode faire().

A propos de la classe Launcher et de la classe centrale Partie : Finalement, c'est dans le “main” de la classe Launcher que l'utilisateur choisit le nombre de joueurs. Ensuite, un objet de type Partie est créé et on lui transmet le nombre de joueurs choisi afin que les joueurs et le paquet de cartes soient créés dans le constructeur de Partie. La distribution des cartes et le lancement de la partie se font lors de l'appel à la méthode launch() de Partie. Une fois la partie terminée, on récupère le vainqueur dans le main de Launcher et on l'affiche. Ce qui change par rapport à l'idée première est que le Launcher paramètre la partie et que Partie s'occupe ensuite de tout le déroulement qui suit. Dans l'idée première, c'était dans la classe Partie que l'on demandait à l'utilisateur de choisir le nombre de joueurs par exemple.

A propos des classes Joueur, JoueurVirtuel et JoueurPhysique : Nous avons décidé finalement de créer une classe JoueurPhysique pour bien montrer que les deux héritaient de Joueur et définissaient bien ses méthodes abstraites. Il y a trois méthodes abstraites qui sont gérées différemment si le joueur est physique ou virtuel : jouerTour(), echangerCartesVisiblesMain() et envoyerTasAAdversaire(). Les joueurs virtuels font appel à l'interface Strategy pour savoir ce qu'ils doivent faire, et le joueur physique choisit ses actions grâce aux options proposées à l'écran.
Egalement, nous avons créé deux énumérations, “TypeJoueur” qui permet de choisir le type du joueur (physique ou virtuel), et “NomJoueur” qui contient une liste de noms pour les joueurs virtuels. Cette liste permet d'attribuer un nom aléatoire aux joueurs virtuels de la partie, nous avons trouvé que c'était plus sympa que de les appeler “joueur 1”, “joueur 2”, etc..

A propos de la classe Config : Nous avons voulu faire quelque chose de pratique en créant une classe Config qui permet de configurer les deux principales options du programme (et non de la partie) : le nombre de joueurs maximal et le mode d'affichage du jeu. La classe Config contient deux attributs. La variable NB_MAX_JOUEURS permet de choisir le nombre maximum de joueurs dans la partie. Par contre, il faudra que l'énumération NomJoueur contienne au moins autant de noms que NB_MAX_JOUEURS. L'attribut MODE_AFFICHAGE est une simple chaine de caractères permettant de lancer le jeu en mode console ou bien en mode fenetre. Il suffit de lui donner la valeur “console” ou la valeur “fenetre” et de compiler. La classe correspondante sera chargée automatiquement.

A propos des classes Clavier, Ecran, Affichage, Console et Fenetre : Pour l'affichage console/fenetre et la récupération des saisies de l'utilisateur, nous avons choisi l'organisation suivante : une classe Clavier globale pour récupérer les saisies aussi bien en mode console qu'en mode fenêtre, et une classe Ecran composée uniquement de méthodes statiques qui correspondent aux opérations graphiques. Depuis la classe Ecran, on va appeler la méthode getInstanceModeAffichage() de la classe Config et cela va nous retourner l'instance de Console ou de Fenetre (qui sont tous les deux des singletons) en fonction du mode choisi dans l'attribut MODE_AFFICHAGE (voir paragraphe précédent sur la classe Config) et on va ensuite appeler la méthode correspondante à l'opération graphique. Les classes Console et Fenetre implémentent l'interface Affichage.
Pour l'exemple, la première opération graphique est d'afficher le “démarrage”. On va donc appeler depuis le “main” la méthode Ecran.demarrage() qui va elle appeler Config.getInstanceModeAffichage().demarrage(). Donc en fonction du choix du mode d'affichage, cela correspondra soit à console.demarrage(), soit fenetre.demarrage(). Dans Console, on va juste afficher un message de bienvenue sur le jeu, et dans Fenetre cela va générer toute la fenêtre et afficher les images. Voilà pour l'exemple.

A propos de la classe Carte et de ses classes filles : Nous avons modifié légèrement la classe Carte pour lui ajouter des choses. Premièrement, nous avons trois attributs : valeur, nom et enseigne. Ce dernier étant la couleur de la carte : carreau, coeur, pique ou trefle. L'attribut “enseigne” n'a finalement pas été utile car cette information n'est pas affichée dans la console et car dans l'interface graphique nous n'avons pas utilisé d'images pour les cartes. Une méthode jouableParDessus() est utilisée pour savoir si une carte (donnée en paramètre à la méthode) peut-être jouée par dessus la carte sur laquelle on appelle la méthode (qui est la dernière carte du tas). Cette méthode est redéfinie dans la classe Sept car lorsqu'on pose un 7, la carte suivante doit être de valeur inférieure donc en fait ce qui se passe c'est que la classe Sept redéfinit la méthode jouableParDessus() en inversant le résultat. La méthode jouable() permet de savoir si une carte peut être jouée. On l'appelle simplement et la méthode retourne true si la carte peut être posée par dessus le tas et false dans le cas contraire. Cette méthode est redéfinie dans la classe Deux par exemple, pour renvoyer true dans tous les cas vu que le 2 peut être posé sur n'importe quelle autre carte. La méthode jouer() permet de jouer la carte. Pour la plupart des cartes, cette méthode appelera simplement la méthode poser() pour ajouter la carte sur le tas, mais pour certaines cartes spéciales, cette méthode jouer() sera redéfinie pour ajouter des actions à faire, par exemple retirer toutes les cartes du tas si la carte à jouer est le 10.

A propos de JeuCartes, Paquet, Tas et Pioche : Des modifications de nom ont été apporté ici. La classe Paquet qui devait contenir l'ensemble de toutes les cartes en créant des paquets de 52 cartes en fonction du nombre de joueurs s'appelle maintenant JeuCartes. Et désormais, la classe Paquet est une classe abstraite dont héritent Tas et Pioche. Ces deux dernières classes sont toujours des singletons récupérables de n'importe où grâce à leur méthode getInstance(). La classe Paquet offre quelques méthodes communes aux deux classes filles, par exemple la méthode retirerCarte() qui permet de retirer la carte du Tas ou de la Pioche tout en renvoyant l'objet qui vient d'être retiré de l'ArrayList. La classe Tas possède bien la méthode getDerniereCarte() qui permet de récupérer comme son nom l'indique l'objet correspondant à la carte qui se trouve sur le dessus du tas.

V. Etat actuel de l'application

Nous avons pensé dès le début à avoir un mode console et un mode fenêtre dans notre application. Ainsi, il suffit de faire un simple changement dans la classe Config pour choisir le mode de l'application. En mode console, toutes les fonctionnalités du cahier des charges ont été implémentées. Aussi, toutes les entrées des utilisateurs sont vérifiées afin d'éviter les bugs. Nous avons essayé d'accorder une attention particulière à l'affichage dans la console pour rendre facile le suivi du jeu pour le joueur. De même nous sommes allé chercher des images libres de droits pour l'ergonomie de notre interface graphique.

Cependant en mode fenêtre (mode graphique) il y a quelques bugs. Il n'est pas possible pour le joueur physique d'échanger ses cartes. De plus, l'affichage des erreurs a été enlevé car les erreurs s'affichaient continuellement, bien que la gestion des erreurs marche toujours. Ces deux bugs sont difficiles à résoudre car ils impliquent de changer la structure du code, ce que nous n'avions pu faire par manque de temps mais le dernier bug ne pose pas de problème dans le jeu en lui même. Il persiste quelques rares bugs minimes qui interviennent dans des cas peu courant.

Des améliorations sont possible tel que l'intelligence artificielle car actuellement les joueurs virtuels n'échangent jamais leurs cartes, il serait possible d'échanger les cartes en adoptant une stratégie. Nous aurions pu aussi gérer la sauvegarde de la partie. Il aurait été préférable d'utiliser le design pattern MVC ce qui aurait pu nous éviter d'avoir des bugs. Il aurait été intéressant d'implémenter une mise en réseau. Toutes ces améliorations auraient été possibles avec un peu de temps supplémentaire.

CONCLUSION

A titre de conclusion, nous sommes vraiment content d'avoir aboutit à ce résultat après autant de travail même si venant du Tronc Commun nous appréhendions un peu le projet. Nous avons pu voir les différentes étapes d'un projet tel que celui-ci notamment la phase de modélisation UML qui fut très importante et a permis un gain de temps non négligeable.

L'application est totalement fonctionnelle en mode console, et la partie graphique marche relativement bien malgré les quelques bugs où nous avons fait de notre mieux pour la rendre la plus ergonomique possible.

Ce projet nous a permis de mettre directement en application ce qui a été vu en cours et en TD, ce qui a permis une meilleure assimilation. Cela sera bénéfique pour nos futurs stages en développement.

ANNEXES

Annexe 1 : Application avec interface graphique