Projet UV LO02 :

La Bataille Norvégienne

Troisième Livrable

Charlélie Borella

Arnaud Pecoraro

Automne 2014

Introduction

Dans le cadre de l'unité de valeur LO02 : Principe et pratique de la programmation orientée objet, il nous est proposé la réalisation d'un projet de jeu de cartes : la Bataille Norvégienne. Ce travail, effectué en binôme se décompose en phases distinctes, allant notamment de la modélisation UML de l'application à son développement en java. Ce document est le troisième livrable de l'application. Il se concentre principalement sur les modifications opérées sur la modélisation envisagée dans le premier livrable et sur l'état actuel du projet.

Ce document se compose de cinq parties distinctes. Les trois premières reprennent le contenu du premier livrable et les deux dernières l'état actuel de l'application.

Table des matières

1 Diagramme des cas d'utilisation

Dans une première partie nous allons nous concentrer sur les cas d'utilisation. En tout premier lieu, Il faut identifier les cas propres à la séquence d'initialisation, au lancement de la partie tels que décrits dans le schéma ci-dessous.

L'utilisateur humain peut au lancement de l'application effectuer plusieurs actions. Le choix de démarrer une partie inclut la distribution des cartes à chacun des joueurs. De même le choix du nombre de joueurs artificiels nécessite de définir la stratégie de jeu de ces derniers. Il est également possible pour les joueurs en début de partie d'échanger leur cartes avec celles qui composent leur tas visible.

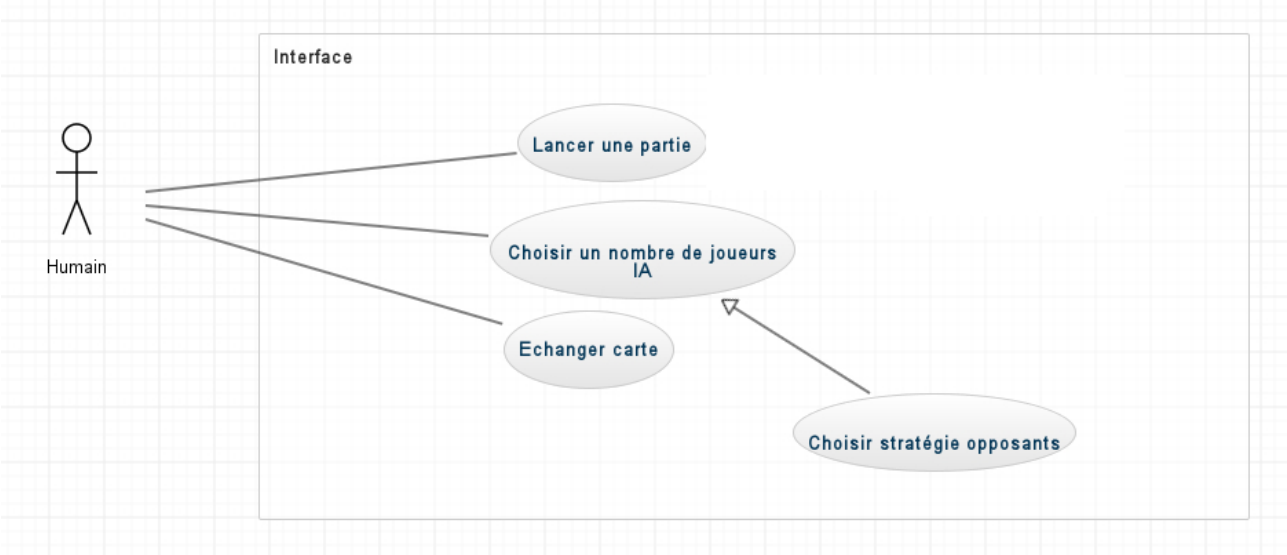

FIGURE 1 – Cas d'utilisation de l'interface

Le diagramme suivant se concentre sur les cas d'utilisation qui se produisent au cours d'un tour de jeu. On retrouve ainsi les différentes interactions entre le joueur et les différents tas de cartes qui composent le jeu. Le joueur a tout d'abord la capacité de poser une ou plusieurs cartes. Cette action inclut le choix des cartes à poser et s'étend à l'action de choisir un joueur qui va prendre le Tapis lorsqu'un As est posé et à l'action de bêcher lorsque la carte posée est un dix.

Après avoir posé une ou plusieurs cartes, le joueur doit compléter sa main pour avoir de nouveau trois carte. Le cas d'utilisation piocher permet au joueur de prendre des cartes dans la pioche. Une fois que celle-ci est vide il recomplète sa main avec ses tas visibles et cachés, on définit donc deux cas d'utilisations à cet effet.

Des cas d'utilisations sont prévus pour suivre la pose d'un As, le cas cocogner, et celle d'un 8, le cas passer son tour. La condition de victoire, ne plus avoir aucune carte en main est matérialisée par le cas être Danish.

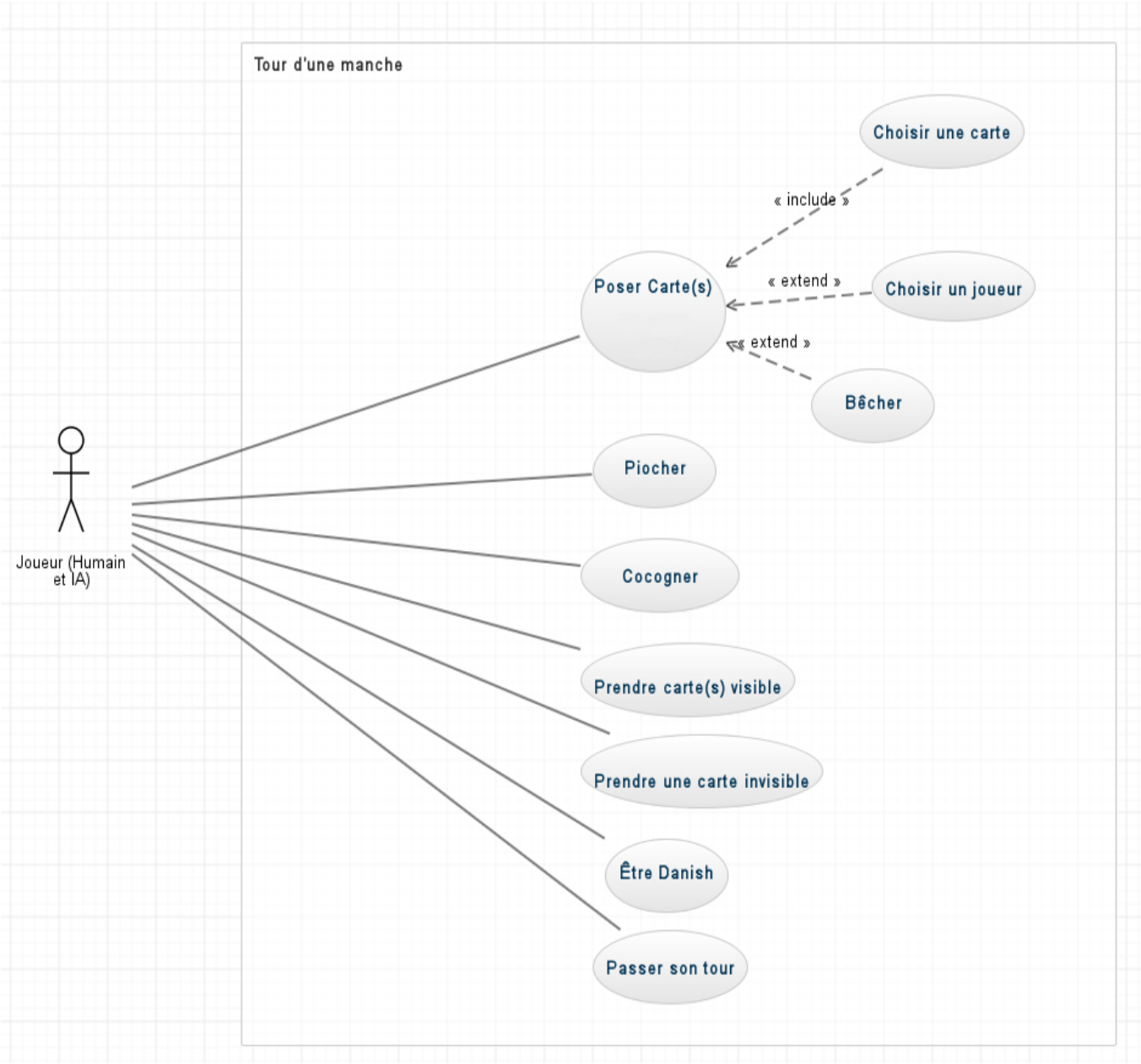

FIGURE 2 – Cas d'utilisation tour de jeu

2 Diagramme de classes

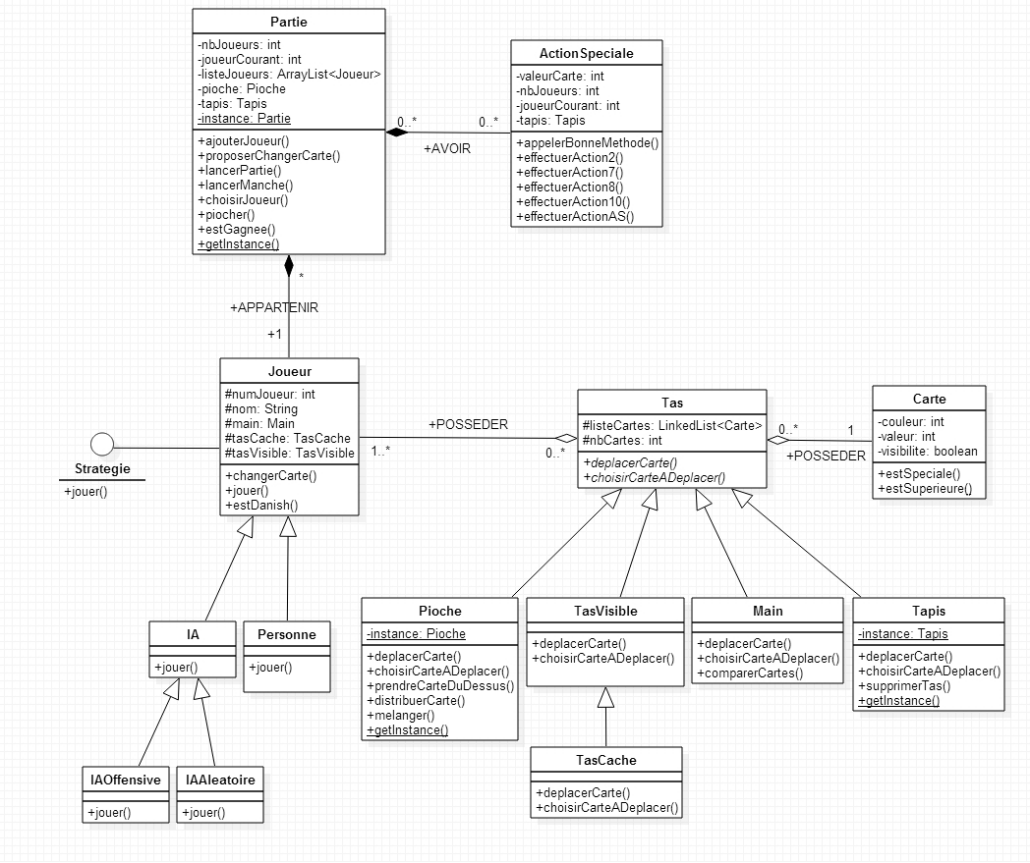

FIGURE 3 – Diagramme de classes

2.1 Classe "Tas"

La classe *Tas* est une classe mère abstraite qui possède plusieurs filles. Nous avons fait ce choix car le jeu comporte plusieurs tas de cartes : la main, la pioche, le tapis, les tas cachés et visibles. Nous ne souhaitions pas implémenter directement cette classe, l'instanciation d'un objet Tas n'est pas pertinente.

Présentation des attributs :

L'attribut *listeCarte* est protected. Ce choix est motivé par une facilté d'accès et de modification accrue dans les classes filles. La visibilité private aurait compliquée l'implémentation, d'autant plus que ces attributs sont essentiels dans les classes héritées. Il regroupe une liste chaînée d'objet de type carte. Nous avons fait le choix d'une LinkedList parce qu'elle implémente l'interface queue. Cela nous permettra d'accéder en FIFO aux données ce qui

semble la manière la plus naturelle de gérer un tas de carte. On tire toujours la carte du dessus et on ne prend jamais de carte au milieu du paquet.

L'attribut *nbCarte*, il est un entier qui stocke le nombre de cartes présentes dans l'attribut listeCarte.

Présentation des méthodes :

La méthode *deplacerCarte()*, c'est une méthode abstraite et publique. Cette méthode doit être surchargée dans les classes filles pour qu'elle puisse déplacer une carte d'un tas à un autre. Chaque classe fille possède sa propre façon de déplacer une carte puisqu'elle le fait toujours d'un tas A vers un tas B différent. Exemple pioche vers main, main vers tapis etc..

La méthode *choisirCarteADeplacer()*, c'est une méthode publique. Cette méthode sera appelée dans *deplacerCarte()*, elle permet de retourner une carte que l'on souhaite déplacer.

2.2 Classe "Pioche"

La classe *Pioche* est une classe fille de la classe *Tas*, elle matérialise la pioche du jeu.

Présentation des attributs :

L'attribut statique *Instance* est utilisé dans le cadre du patron de conception Singleton. *Instance* couplé à la méthode statique "getInstance()" permet donc de vérifier qu'il n'existe bien qu'une seule instance de la classe ou d'en créer s'il n'y en a pas. En effet la pioche du jeu doit être unique.

Présentation des méthodes :

La méthode *prendreCarteDuDessus()* retourne la carte du dessus de la pioche. Elle sert donc à piocher.

2.3 Classe "TasVisible" et "TasCache"

Les classes *TasVisibles* et *TasCache* sont presque identiques, elle diffère par l'implémentation de leur méthode et la visibilité de leur cartes. *TasCache* spécialise *TasVisible*.

Présentation des attributs :

Pas d'attribut particulier en plus de ceux hérités.

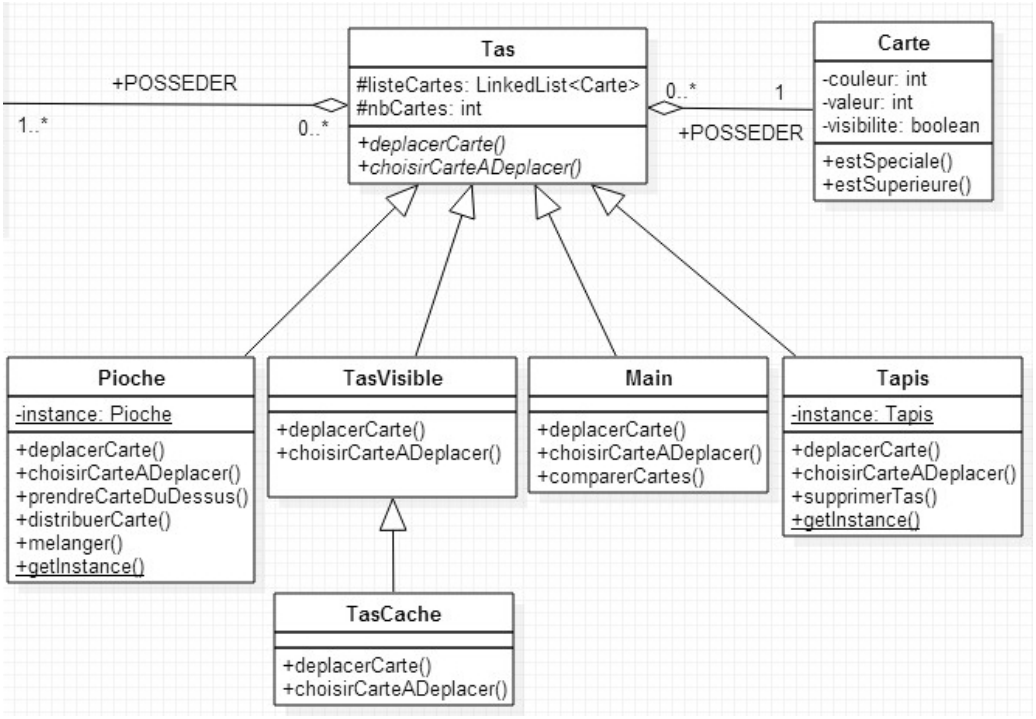

FIGURE 4 – Classe abstraite Tas et classe Carte

Présentation des méthodes :

Pas de méthode particulière en plus de celles héritées.

2.4 Classe "Main"

La classe *Main* joueur représente l'ensemble des cartes qui composent la main du joueur. Elle hérite de Tas puisqu'elle est constituée d'une collection de carte.

Présentation des attributs :

Pas d'attribut particulier en plus de ceux hérités.

Présentation des méthodes :

La méthode *comparerCarte()* permet de comparer la carte que le joueur souhaite poser avec la dernière carte du tapis, afin s'assurer qu'elle est bien supérieure et donc posable. Si la carte ne peut pas être posée, la méthode retourne booléen false.

2.5 Classe "Tapis"

La classe *Tapis* représente l'ensemble des cartes jouées durant une manche par les joueurs. Elle hérite de *Tas* puisqu'elle est composée d'une collection de carte avec un objectif plus spécifique.

Présentation des attributs :

L'attribut statique *Instance* est utilisé dans le cadre du patron de conception Singleton. *Instance* couplé à la méthode statique "getInstance()" permet donc de vérifier qu'il n'existe bien qu'une seule instance de la classe ou d'en créer s'il n'y en a pas. En effet le tapis du jeu doit être unique.

Présentation des méthodes :

La méthode *comparerCarte()* permet de comparer la carte que le joueur souhaite poser avec la dernière carte du tapis. Si la carte ne peut pas être posée la méthode retournera un booléen false.

2.6 Classe "Carte"

La classe *Carte* matérialise une carte. Elle est rattachée par une association à *Tas* avec une flèche vers *Tas* car celui-ci possède des cartes alors que la classe *Carte* n'a pas d'information sur la classe *Tas*.

Présentation des attributs :

L'attribut *couleur* permet de représenter la couleur de la carte sous forme d'entier via un formalisme prédéfini, par exemple : rouge=0, noir=1 etc..

L'attribut *valeur* permet de représenter la valeur de la carte sous forme d'entier via un formalisme prédéfini, par exemple : as=1, 1=1, valet=11 etc..

L'attribut *visibilite* permet de savoir si la carte est retourner ou non via un booléen, par exemple : true = visible, false= retourné, invisible.

Présentation des méthodes :

La méthode *estSpeciale()* permet de déterminer si la carte a une fonction particulière. Si la méthode retourne vrai alors on pourra faire appel à la classe *ActionSpeciale* pour gérer sa spécificité.

La méthode *estSuperieure()* prend une autre carte en paramètre et s'assure de faire la comparaison entre entre celle-ci et elle-même. La méthode retourne un booléen vrai si la carte passée en paramètre est supérieure.

2.7 Classe "Partie"

Partie constitue la classe principale de l'application. Elle contient tous les éléments nécessaires au bon déroulement de la partie. Elle comprend notamment des méthodes pour l'initialisation de la partie, tel que les opérations d'ajout de joueur et de changement de carte initial. Elle gère également les manches et leur déroulement. Nous avons considéré comme manche,

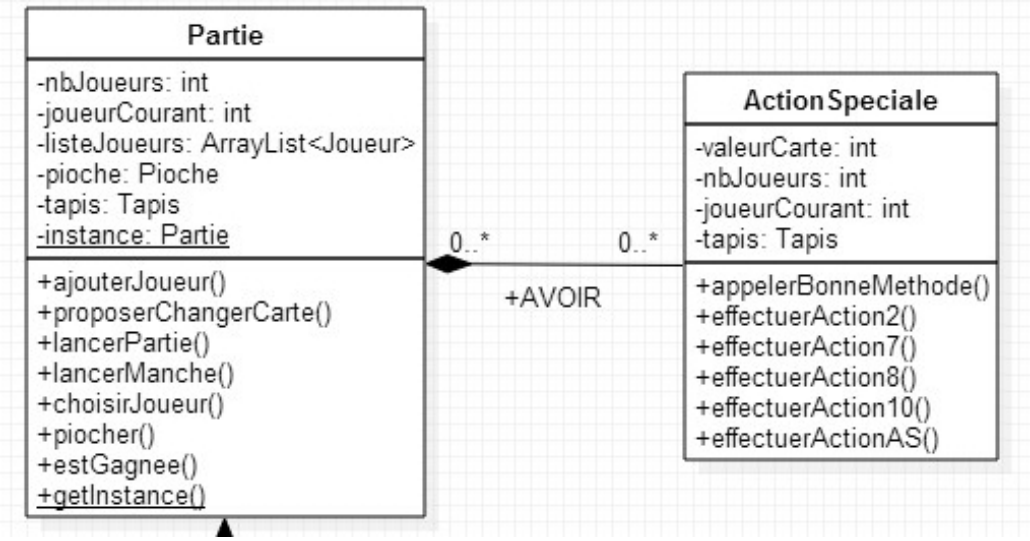

FIGURE 5 – Classes Partie et ActionSpeciale

chaque période de jeu séparant une "cocogne", soit la disparition du tapis central. Dés qu'une manche prend fin, la boucle de fonctionnement en relance une.

Présentation des attributs :

Naturellement, il nous est nécessaire d'avoir un attribut regroupant les différents joueurs de la partie. Nous avons donc utilisé une collection, l'attribut listeJoueurs contenant tous les joueurs participant à la partie. On note également la présence d'instances des classes filles de Tas, matérialisant la pioche et le tapis.

Présentation des méthodes :

La méthode *lancerPartie()* correspond à la boucle de fonctionnement global du jeu. On ne peut en sortir qu'en cas de victoire de l'un des joueurs sur ses adversaires, par un test effectué grâce à la méthode *estGagne()*. C'est à l'intérieur de cette méthode que sont lancés les manches successives.

La méthode *choisirJoueur()* permet notamment dans le cas d'une action spéciale de désigner un joueur qui va prendre le tas, cocogner et donc par conséquent provoquer la fin de la manche.

2.8 Classe "ActionSpeciale"

Cette classe permet de gérer les modifications du déroulement de la partie induites par la pose de carte spéciales. Son fonctionnement est simple, on récupère la dernière carte jouée. Si celle-ci est spéciale, on exécute l'action correspondante.

Présentation des méthodes :

La méthode *appelBonneMethode()* permet de choisir l'action à effectuer par rapport à la carte spéciale qui vient d'être jouée.

2.9 Classe "Joueur"

Cette classe sert à définir l'entité du joueur, aussi bien dans sa version humaine que gérée par l'ordinateur(IA). Elle contient les actions basiques que peut effectuer le joueur.

Présentation des attributs :

Un joueur se définit par son nom, son numéro et les trois tas qui lui sont propres, à savoir les instances de *Main*, *TasVisible* et" *TasCache*.

Présentation des méthodes :

Le joueur peut changer ses cartes en début de partie dans la période d'initialisation mais aussi prendre des cartes de ses tas *TasCache* et *TasVisible* grâce à la méthode *changerCarte()*. C'est par appel de la méthode *jouer()* que le joueur peut poser de une à trois cartes. Cette méthode voit son implémentation varier en fonction du type de joueur. En effet, il est nécessaire de changer dynamiquement les algorithmes qui sont utilisés. On utilise pour cela le patron de conception Stratégie. Son analyse est présentée plus bas.

Selon les règles du jeu, si le joueur ne peut pas jouer lorsque c'est son tour, il "cocogne" en utilisant la méthode *cocogner()* qui met fin à la manche courante. Il peut également *etreDanish()*, c'est-à-dire n'avoir plus aucune carte à jouer dans sa *Main* ni dans ses *TasVisible* et *TasCachee* et par conséquent gagne la partie.

Description de l'héritage :

La classe *Joueur* a deux spécialisations : *Personne* et *IA* qui doivent gérer différement l'action de jouer. Les classes *IAOffensive* et *IAAleatoire* héritent de IA et redéfinissent la méthode jouer() pour une implémentation plus précise selon le type de joueur artificiel.

2.10 Etude du patron de conception Stratégie :

Il s'agit d'une application du principe de redéfinition. Le patron de conception se compose soit d'une interface *Strategie* qui contient des méthodes qui seront implémentées différement dans les classes qui vont implémenter cette interface, soit d'une classe abstraite *Strategie* dont les méthodes abstraites seront définies différement dans ses classes filles. On peut ainsi faire varier différment les algorithmes utilisées.

Dans le cadre de la Bataille Norvégienne nous avons choisi d'utiliser une interface *Strategie* qui contient une méthode *jouer()*. Cette méthode correspond à la façon dont le joueur pose les cartes. La classe *Joueur* implémente cette interface. La méthode *jouer()* est implémentée

dans les classes filles *Personne* et *IA*. Dans *Personne* elle fait appel à un choix humain, alors que dans les classes qui héritent de *IA*, elle correspond soit à un choix totalement aléatoire de l'ordinateur : la première carte posable étant posée, soit à un choix dit offensif : les cartes entraînant un passage de tour ou une cocogne étant privilégiées.

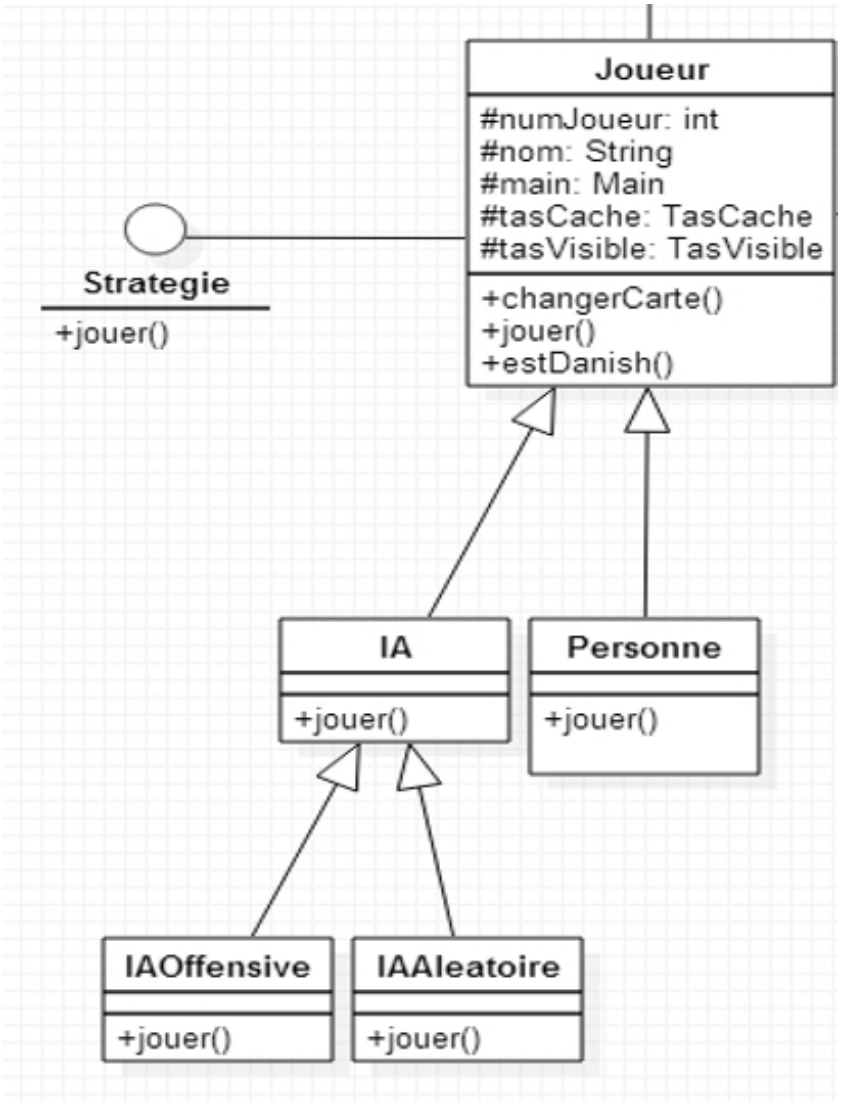

FIGURE 6 – Classe Joueur et implémentation de Stratégie

3 Diagramme de séquence

Cette dernière partie tend à résumer l'initialisation d'une partie puis le déroulement d'un tour au travers d'un diagramme de séquence. Pour des raisons de lisibilité et de compréhension, nous avons choisi de réduire au maximum le nombre d'instances présentes sur le diagramme. Ainsi les mains des joueurs ne sont pas représentées, ni les interactions avec leur tas cachés.

La première partie du diagramme se focalise sur l'initialisation de la partie et la mise en place du jeu. On y retrouve donc la création des différentes entités de tas de cartes, la distribution des cartes à chacun de joueur.

Le rectangle qui suit présente le cas d'un changement de cartes en début de partie, c'est-à-dire l'échange des cartes de la main d'un joueur avec celles qui composent son tas visible.

Le déroulement de la partie est modélisé par deux boucles imbriquées, qui symbolisent le déroulement d'une manche à l'intérieur de la partie. Effectivement, dans notre étude de la Bataille Norvégienne, nous avons considéré comme manche toute période de jeu séparant une cocogne, c'est-à-dire la suppression du tapis. La boucle interne du diagramme montre ici un exemple de ce fonctionnement ,en considérant une manche de jeu élémentaire. Le joueur humain joue puis pioche pour compléter sa main. Le joueur IA fait de même. C'est à nouveau au tour du joueur humain, il ne peut pas jouer, il cocogne : les cartes qui composent le tapis sont déplacées vers sa main. Le tapis est détruit. La méthode *lancerManche()*, est appeleé et on entre à nouveau dans la boucle de manche. La méthode *getInstance()* permet la création d'un nouveau tapis et le jeu peut reprendre.

Lorsqu'un joueur pose sa dernière carte et que le tas de pioche, ainsi que ses tas visibles et invisibles sont vides, il est danish. La manche se termine et la partie également. Ces conditions de sorties devront être expliciteés dans le code et se baseront les méthodes tests *estDanish()* et *estGagnee()*.

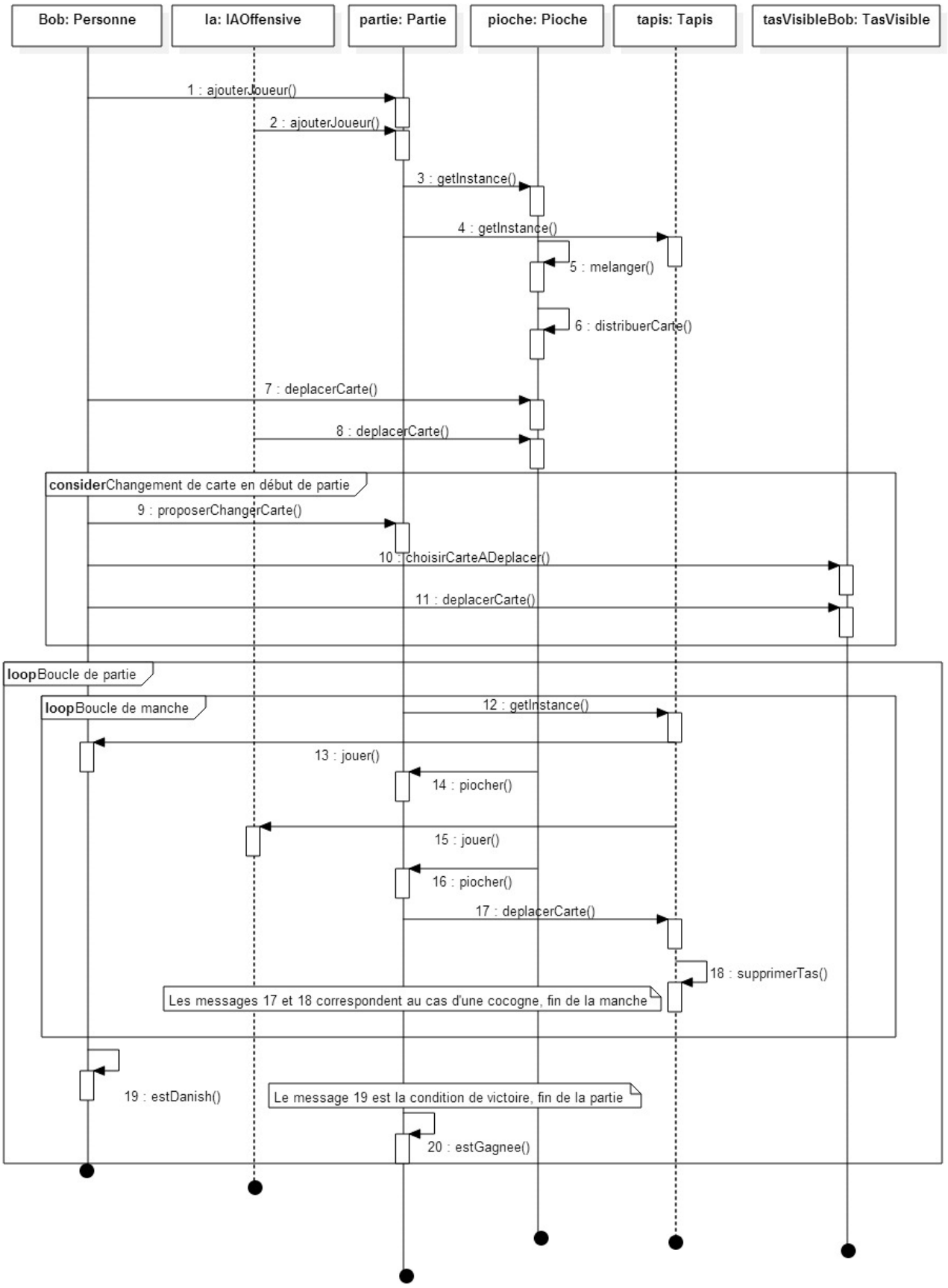

FIGURE 7 – Diagramme de séquence

4 Vue d'ensemble de la modélisation finale

L'implémentation du modèle n'a occasionné aucune modification majeur au niveau du diagramme de classes. Certaines méthode ont été changés.

Le projet à été scindé en 4 packages : carte, joueur, partie et vue.

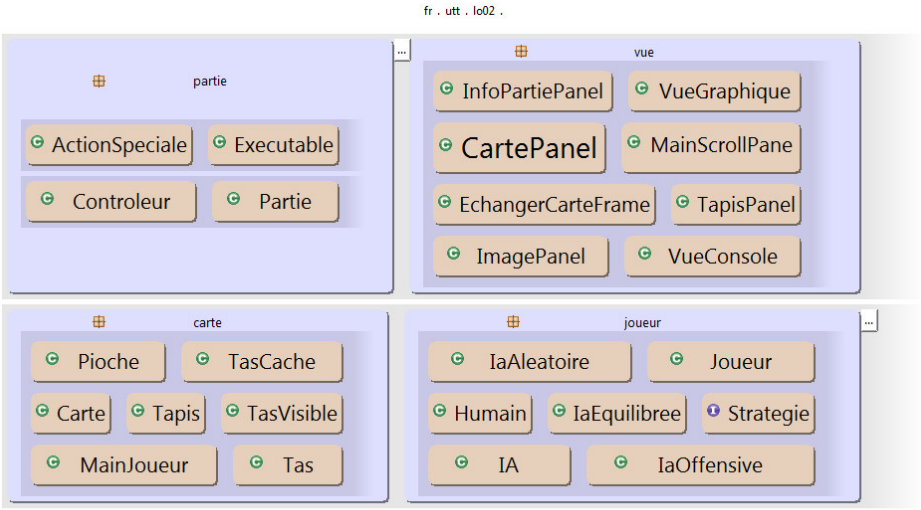

FIGURE 8 – Diagramme en couche du projet

Pour l'organisation de *MVC* (Modèle, Vue Contrôleur) nous avons crée un seul contrôleur qui faisait le pont entre les différentes classes qui composent la vue graphique et les modèles. Le modèlesont les différentes classes qui font partie des packages Carte, Joueur et Partie.

Le bouton *Envoyer* présent dans l'interface graphique actionne via un listener une méthode du contrôleur. Celui-ci se charge d'appeler les différentes méthodes permettant de faire jouer. Lorsque les modifications sont effectués le contrôleur fait rafraichir les données de la vue et ainsi de suite.

5 Etat actuel du projet

De manière générale, le projet nous semble fonctionnel. Les contraintes de fonctionnements explicitées dans le cahier des charges sont respectées :

— Il est possible de lancer une partie comprenant un joueur humain et plusieurs joueurs virtuels.
— Les joueurs virtuels disposent de trois comportement différents : Offensif, Aléatoire et Equilibrée qui influent sur leur technique de jeu.
— Les cartes spéciales sont gérées, elles impliquent des actions spéciales affectant directement la partie
— Une interface graphique utilisant utilisant *Swing* a été implémentée et est fonctionnelle

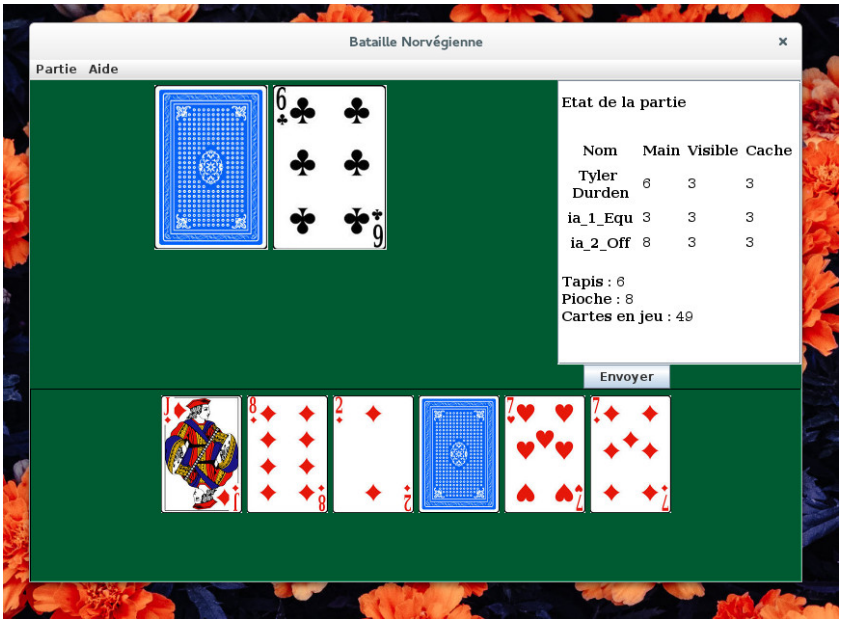

FIGURE 9 – Screenshot d'une partie

Améliorations possibles :

Il subsiste malgré tout quelques légers problèmes au niveau de l'interface graphique dont la résolution pourrait améliorer l'expérience de jeu :

— Les rafraîchissement multiples de l'interface entre chaque tour
— Un bug de scrolling du panel présentant la main du joueur qui ne s'adapte pas toujours correctement lors de l'ajout d'un nombre important de carte.
— L'estéthique générale, notamment celle du panel d'information rédigée entièrement en HTML qui est très sommaire.

D'autre part certaines améliorations du moteur de jeu pourrait s'avérer intéressantes comme :

— L'implémentation d'un système de sauvegarde de partie
— L'ajout de stratégies IA plus efficaces, et donc d'un niveau de difficulté supplémentaire
— L'implémentation des multiples variantes du jeu

Conclusion

En conclusion la réalisation de ce projet a constitué une expérience enrichissante. Tout d'abord du point de vue des compétences développées en programmation. En effet, le projet a permis de mettre en application les compétences acquises au cours de ce semestre avec en premier lieu la modélisation *UML*. Il a été nécessaire de réfléchir à une modélisation logique et cohérente du problème. La partie Java nous a permis d'appliquer les concepts principaux du cours tels les collections, la programmation par événements mais surtout les design patterns *MVC* et *stratégie*. La mise en place de ces derniers ainsi que Swing a necessité des approfondissements personnels, et donc le développement de compétences annexes.

Par ailleurs, cet exercice nous a aussi permis de nous confronter à la gestion d'un petit projet informatique, les différentes phases de son élaboration avec des deadlines fixées à l'avance. De plus, nous avons du réfléchir à une méthode pour travailler efficacement en équipe en se répartissant bien la charge de développement, mais également la gestion efficace des versions que nous avons réalisée avec l'outil *Git*.

Au final, nous sommes parvenu à un résultat qui bien que perfectible nous semble fonctionnel et agréable.

Table des figures