Nom : Culié Florian
Patelli Alexandre
Niveau : *ISI01*

Livrable 3 LO02 :

Modélisation UML finale du jeu menhir et état de l'application.

Résumé :

Dans ce rapport se trouve une modélisation UML finale (via trois diagrammes, de cas d'utilisation, de classes et de séquences) du jeu menhir, de la manière que nous pensons la plus judicieuse de le programmer avec JAVA et l'IDE Eclipse et enfin une description de l'état actuel de l'application. Dans cette dernière partie, nous détaillerons les différences notables entre cette dernière et le cahier des charges, ainsi que les fonctionnalités implantées ou pas, et les bugs restants.

UV : LO02
Livrable 1
Responsable : G. DOYEN
Semestre : *Automne 2015*

Florian Culié
Alexandre Patelli

Table des matières

Introduction 3
Diagramme de cas d'utilisation 4
Diagramme de classe 6
Diagramme de séquence 10
Conclusion 15

Florian Culié
Alexandre Patelli

Introduction

Dans le cadre de l'UV LO02, nous devons réaliser une application en langage JAVA permettant de jouer au jeu de carte menhir.

Pour ce faire, nous devons, dans un premier temps, réaliser trois diagrammes UML (diagramme de cas d'utilisation, diagramme de classe, diagramme de séquence) que nous allons présenter dans ce premier rapport. Ces diagrammes ont été réalisés via le plug-in Omondo sous Eclipse, puis réécrit via le logiciel Visual Paradigm pour pouvoir être exportés et intégrés plus facilement au rapport.

Ensuite, nous allons devoir développer l'application, via l'IDE Eclipse, logiciel qui nous a été imposé dans le cadre de l'UV. Grâce au plug-in Omondo et les trois diagrammes que nous avons réalisé, une partie du code sera déjà générée (le squelette des classes entre autre, via le diagramme de classe).

Dans l'idéal, le jeu devra ressembler le plus possible à la version réelle, avec la présence, graphiquement, des dos de cartes si une carte est retournée (par exemple dans la main de l'adversaire), des images recto tirées du PDF de carte et des graines et menhirs possédés par les différents joueurs. Outre cet aspect graphique, le programme devra gérer la succession des saisons et les différentes valeurs des cartes associées à celles-ci. De plus, nous allons devoir différencier deux types de partie, celles dites "Rapides", et celles dites "Avancées", et leurs lots de spécificités.

Diagramme de cas d'utilisation

Nous avons, pour mener le projet à bien, réalisé le premier diagramme UML, celui traitant des cas d'utilisation, présenté et expliqué ci-dessous.

En premier lieu, nous avons un utilisateur de l'application. Il devra créer un avatar et renseigner son âge et son genre, pour permettre de déterminer quel joueur commencera la partie. Ensuite, il pourra choisir le nombre d'adversaires virtuels qu'il souhaite affronter puis sélectionner, pour chacun d'eux la difficulté qui leur sera associée. Il aura ensuite la possibilité de jouer une partie, en sélectionnant son type, à savoir Rapide ou Avancée, et donc verra ses possibilités en jeu modifiées.

La partie rapide permet au joueur de piocher quatre cartes "Ingrédient" et d'obtenir deux graines avant le début de la manche. Il devra alors jouer une manche dite "Rapide", avec son déroulement par tour, expliqué dans les règles du jeu. Pour ce faire, il ne peut jouer que des cartes "Ingrédient", et choisir leurs actions parmi trois choix possibles :

L'engrais qui permettra de faire pousser X graines de menhir, X étant le nombre de graine de menhir que possède le joueur, sans excéder la valeur de la carte à la saison correspondant au tour,

Le géant qui permettra de gagner X graines de menhir, X désignant la valeur inscrite sur la carte, à la saison correspondant au tour,

Les farfadets qui permettront de voler X graines à un joueur que l'utilisateur ciblera une fois la carte jouée. X désigne le nombre de graines volées au joueur cible, sans excéder la valeur de la carte à la saison correspondant au tour, et sans excéder le nombre de graine du joueur victime de l'action.

La partie Avancée permet, outre le fait de jouer des cartes "Ingrédient" comme décrit au-dessus, d'autres choix. En effet, la partie avancée se déroulant en plusieurs manches (autant que le nombre de joueur), l'utilisateur :

Choisit à chaque début de manche de piocher une carte "Allié" ou de recevoir 2 graines ;

Reçoit quatre cartes "Ingrédient" à chaque début de manche.

En plus de cela, il possède la capacité de jouer, à n'importe quel moment, une carte "Allié", s'il en a pioché une au début de la manche. Ces cartes sont de deux types différents :

Soit c'est une carte "Taupe", qui permettra au joueur de détruire X menhirs parmi ceux ayant déjà poussé chez un autre joueur que l'utilisateur ciblera après avoir joué la carte, X étant le nombre indiqué par la carte à la saison correspondante, sans excéder le nombre de menhirs poussés que le joueur possède.

Soit c'est une carte "Chien", qui permettra au joueur de protéger X de ses graines contre une attaque de farfadets adverse, X désignant le nombre de graines que la carte chiens protège, et rend "insensible" à l'attaque des farfadets.

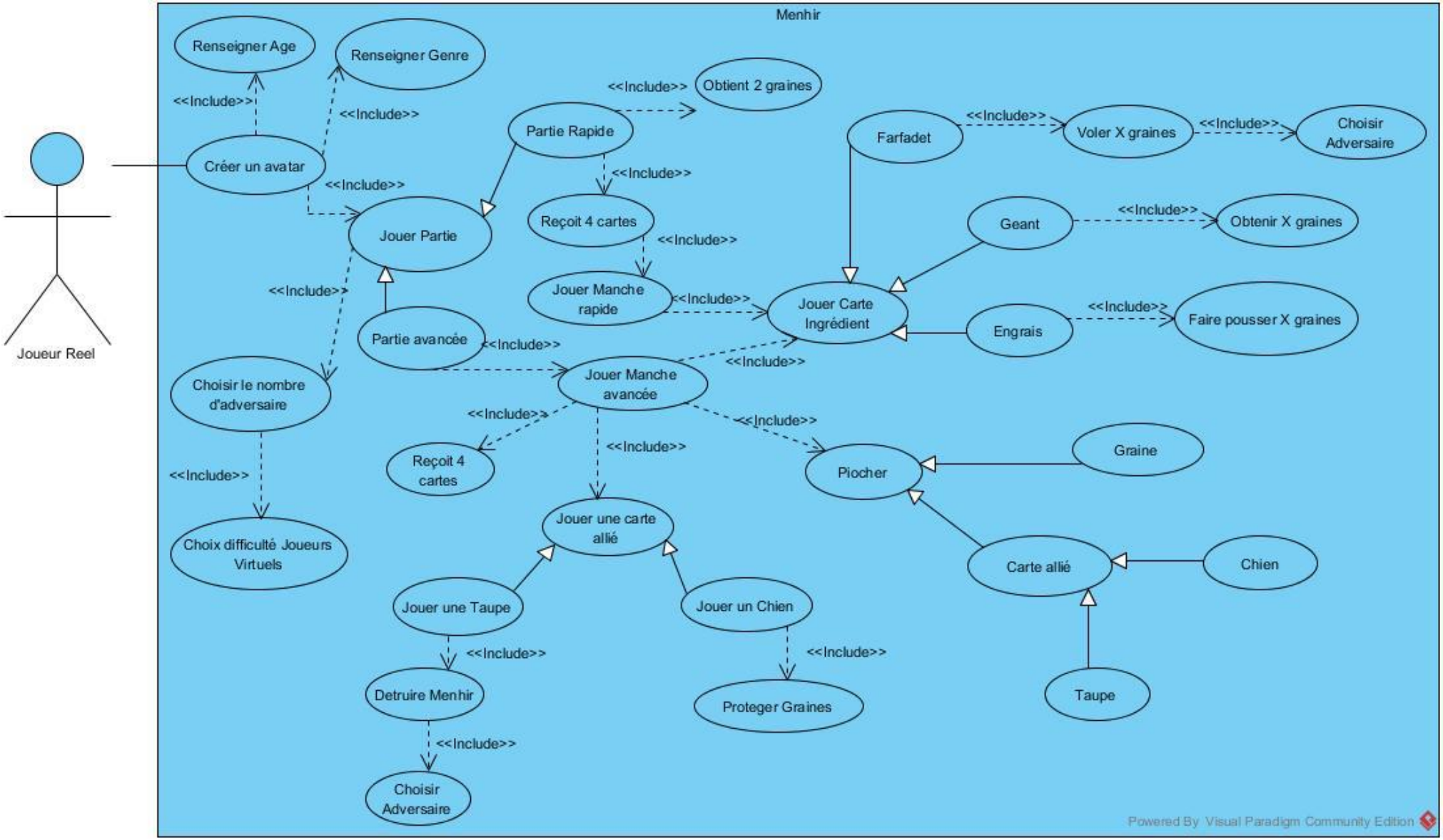
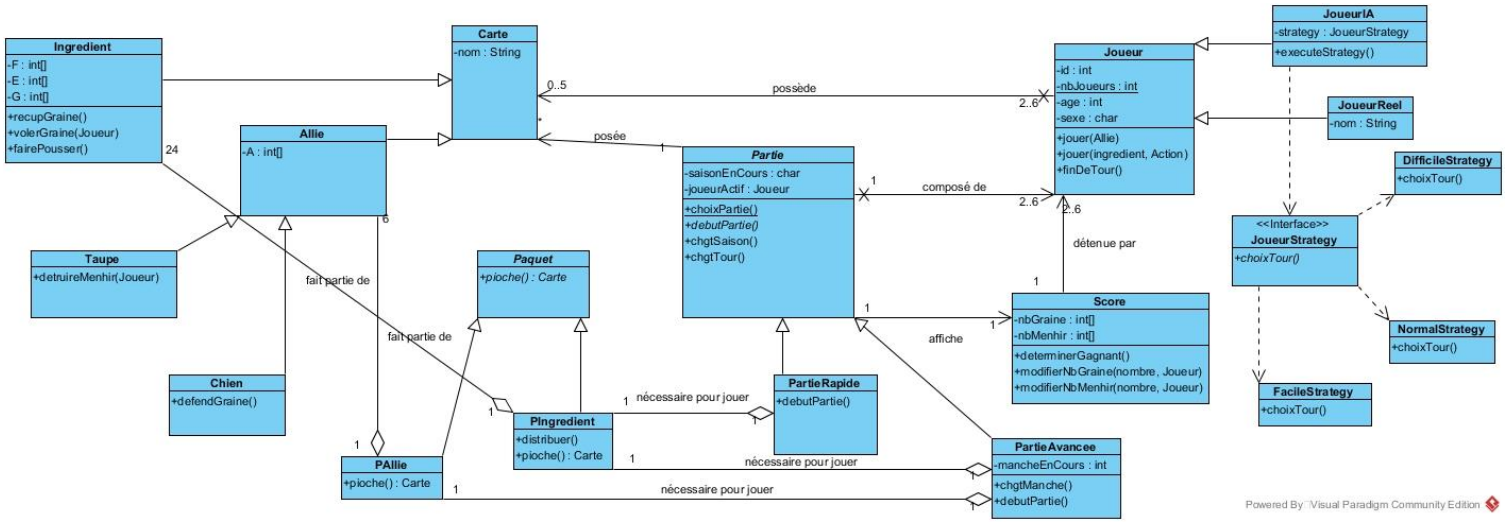

Menhir
Joueur Reel
Renseigner Age
Renseigner Genre
Créer un avatar
Jouer Partie
Partie Rapide
Obtient 2 graines
Reçoit 4 cartes
Jouer Manche rapide
Partie avancée
Jouer Manche avancée
Choisir le nombre d'adversaire
Choix difficulté Joueurs Virtuels
Reçoit 4 cartes
Jouer une carte allié
Jouer une Taupe
Jouer un Chien
Detruire Menhir
Choisir Adversaire
Proteger Graines
Jouer Carte Ingrédient
Farfadet
Voler X graines
Choisir Adversaire
Geant
Obtenir X graines
Engrais
Faire pousser X graines
Piocher
Graine
Carte allié
Chien
Taupe
<<Include>>
Powered By Visual Paradigm Community Edition

Diagramme de classe



Les classes Parties :

Avant de commencer à jouer, il faut que l'utilisateur puisse renseigner le nombre de joueurs, la difficulté (IA) de chacun des joueurs adverses ainsi que le type de partie auquel il souhaite jouer. Pour cela nous avons créé la classe abstraite "Partie". Elle possède deux classes filles qui sont "PartieRapide" et "PartieAvancee". Cette classe "Partie" sera donc le cœur central de l'application, c'est elle qui va instaurer le déroulement de la partie et interagir avec les autres objets.

On peut noter la méthode statique "choixPartie" dans "Partie", cette méthode nous permet de demander à l'utilisateur les différentes informations qu'il a choisi pour sa partie et ensuite d'instancier un objet de type "PartieRapide" ou "PartieAvancee" en fonction de son choix.

Une fois la partie initialisée, on va avoir différentes méthodes qui vont permettre le bon déroulement de la partie. Dans un premier temps la méthode "debutPartie" va permettre d'initialiser les différentes variables de la partie comme le score ou les cartes que nous verrons ensuite. On peut noter que la méthode "debutPartie" est abstraite car le début de partie est différent en fonction de la partie. Il faudra donc la définir différemment dans les deux classes filles de "Partie".

Enfin on va avoir plusieurs classes qui vont permettre de faire évoluer la partie, tout cela décomposé en manches, saisons et tours grâce aux méthodes "chgtManche", chgtSaison" et "chgtTour". A noter que la méthode "chgtManche" n'est définie que pour "PartieAvancee" car il n'y a pas de manche en partie rapide.

La classe "Partie" possède deux attributs qui sont "saisonEnCours" et "joueurActif" et qui permette de connaître à chaque instant l'état de la partie.

Les classes Joueurs :

On peut différencier deux types de joueurs, le joueur réel et le joueur simulé par une IA. C'est la raison pour laquelle la classe "Joueur" possède 2 classes filles qui sont "JoueurIA" et "JoueurReel". Chaque joueur est défini par un id (entier), un âge, un sexe et possède différentes méthodes qui lui permette de jouer ("jouer") ou de mettre fin à son tour ("finDeTour"). On peut noter que la méthode "jouer" est surchargée, en effet le joueur peut jouer 2 types de cartes différentes, il y a donc 2 façons de jouer.
La classe possède un attribut statique qui est "nbJoueurs" afin de compter le nombre de joueurs.

La classe Joueur entretient une association avec la classe Partie, en effet différents joueurs vont être associés à une partie.

Les classes Cartes :

On peut différencier deux types de cartes, les cartes "Ingredient" et les cartes "Allié". C'est la raison pour laquelle la classe "Carte" possède 2 classes filles qui sont "Ingredient" et "Allie". Chaque carte est définie par son nom ("nom"), ensuite on va avoir différentes informations en fonction du type de carte. Un ingrédient peut effectuer 3 actions différentes, avec un coefficient différent pour chacune des 4 saisons. La classe "Ingredient" contient donc trois tableaux d'entiers ("F", "E", "G") qui vont contenir chacun 4 éléments. Elle contient donc aussi 3 méthodes "recupGraine", "volerGraine(Joueur)" et "fairePousser" qui sont les 3 actions différentes possibles de la carte.

On peut différencier 2 différents types de carte Allié, la classe "Allié" possède donc deux classes filles qui sont "Taupe" et "Chien". Chaque carte a un coefficient en fonction de la saison, on a donc mis un attribut "A" qui est un tableau de 4 éléments. On a mis cet attribut dans la classe mère "Allie" car les deux classes filles ont besoin de ce tableau. On a ensuite une méthode différente pour chacune des deux classes filles qui sont "detruireMenhir(Joueur)" et "defendGraine" et qui sont les actions des 2 cartes.

La classe "Carte" est associée à la classe "Joueur" car un joueur va posséder des cartes. Elle est aussi associée à la classe "Partie" car des cartes vont être jouées dans la partie.

Les classes Paquets :

Il nous a paru utile de créer une classe abstraite "Paquet" qui va contenir des cartes. En effet, dans une partie de jeu de cartes, le joueur est amené à distribuer ou piocher des cartes. On peut piocher deux différents types de cartes à différents moments d'une partie, il parait

donc utile de créer deux classes filles de “Paquet” qui sont “PIngredient” qui va contenir les cartes Ingrédient et “PAllie” qui va contenir les cartes Allié.

La méthode abstraite “piocher” permet au joueur de piocher, elle devra être définie pour chacun des deux paquets de carte. On peut noter que la classe “PIngredient” possède une méthode “distribuer”, en effet au début de partie on doit distribuer aux joueurs un nombre d’ingrédients définis.

Les classes “PIngredient” et “PAllie” sont respectivement des agrégations des classes “Allie” et “Ingredient”, en effet ces cartes constituent les paquets.
On peut aussi noter que les classes filles de “Partie” sont des agrégations des classes filles de “Paquet”, en effet elles sont nécessaire pour jouer une partie et sont définies pour une partie.

La classe Score :

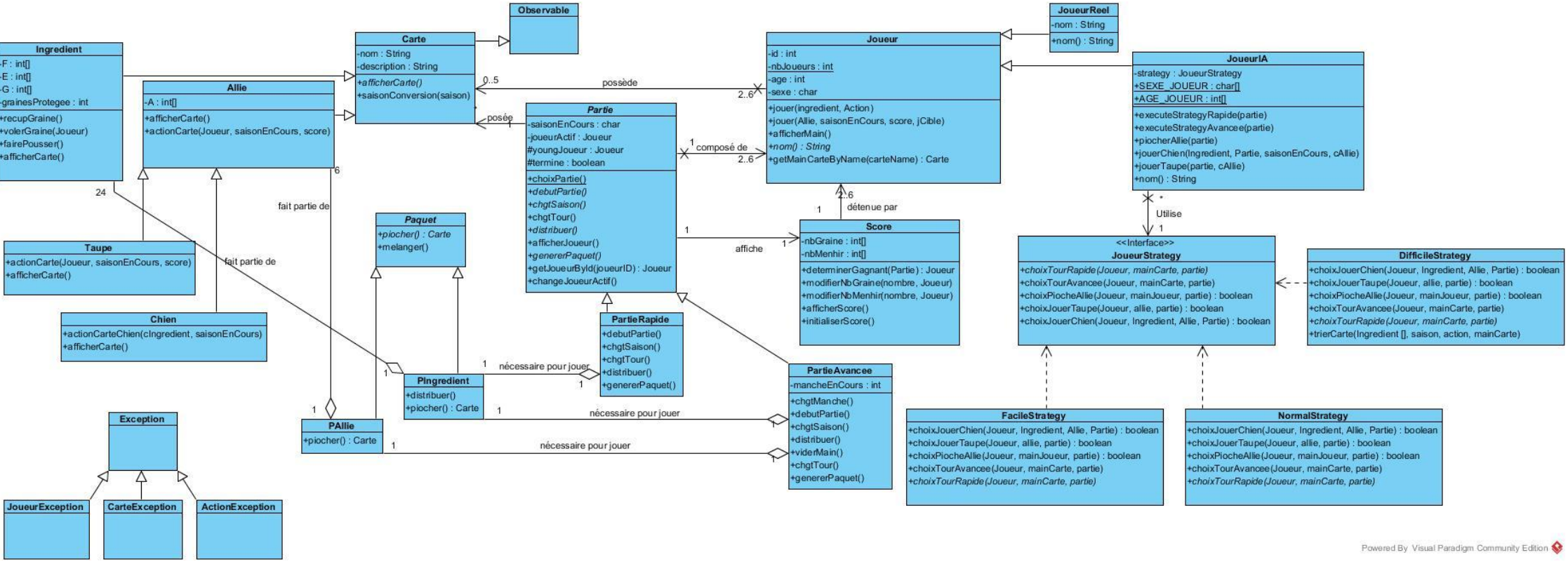
Il est important dans ce jeu de carte de connaître et de noter le score de chaque joueur. Pour cela nous avons fait le choix de créer une classe “Score”. Une instance de cette classe contiendra tous les différents scores des joueurs d’une partie. Les scores seront donc stockés dans des tableaux (“nbGraine”, “nbMenhir”) avec un élément de tableau pour chaque joueur. On aura une méthode, “determinerGagnant” qui permettra à la fin de la partie de déterminer le gagnant. Les méthodes “modifierNbGraine” et “modifierNbMenhir” permettent quant à elles d’ajouter ou de supprimer un certains nombres de graines ou menhirs à un joueur.

Un score est associé à plusieurs joueurs, ainsi qu’à une partie.

Les classe Strategy :

Comme demandé dans le sujet du projet nous avons utilisé le patron de conception “Strategy” pour concevoir l’IA de notre jeu. Le point clé est la création d’une interface “JoueurStrategy” qui va implémenter et nous assurer que les 3 différents algorithmes de difficulté qui sont “FacileStrategy”, “NormalStrategy” et “DifficileStrategy” respectent la même structure et peuvent être utilisé tous de la même façon. Il suffira ensuite à la création d’une instance de “JoueurIA” d’indiquer quel algorithme de difficulté il faut utiliser pour ce joueur puis la méthode “executeStrategy” de “JoueurIA” utilisera l’algorithme sélectionné.

Diagramme de classe Final



Modifications par rapport au diagramme Initial

Pour permettre l'implantation du patron MVC, nous avons dû faire hériter la classe Carte de la classe Observable.

Nous avons également créé des classes d'exception, pour pouvoir les exceptions qui pouvaient avoir lieu. Ainsi, trois classes ont étés créées, la classe JoueurException, qui est levée si un joueur est inconnue, la classe CarteException, qui est levée si une carte introuvable dans la main du joueur, ou inconnue et enfin ActionException, qui est levée si l'action liée à une carte est inconnue.

Concernant les classes de cartes, la méthode afficherCarte est désormais inutilisable, bien que présente dans le code. En effet, elle permettait un affichage console de la carte, fonction désormais effectuée par l'interface graphique. Cette méthode est indiquée comme dépréciée dans la javadoc du projet. Les méthodes des classes alliées ont été changées, dans leurs noms et leurs paramètres.

Les classes de paquets disposent également d'une méthode pour mélanger ces derniers.

La classe partie possède également trois nouveaux attributs, joueurActif, qui stocke le joueur en train de jouer, youngJoueur, qui permet de stocker le joueur qui commence à chaque manche, et termine, qui permet de vérifier si la partie est terminée ou non. Nous avons également créé d'autres méthodes. Distribuer permet de distribuer les cartes aux joueurs, et genererPaquet pour créer les paquets de la partie. La méthode getJoueurById permet de renvoyer une instance du joueur à partir de son ID. Enfin, la méthode changeJoueurActif permet de changer le joueur qui est en train de jouer.

Les classes PartieRapide et PartieAvancee possède également des méthodes héritées et surchargées de sa super classe Partie, car le traitement n'est pas le même pour une partie Avancée. PartieAvancee possède une méthode supplémentaire, viderMain, qui permet de vider la main des joueurs à la fin de chaque manche.

La classe score possède deux nouvelles méthodes, afficherScore et initialiserScore. afficherScore permet d'afficher le score de la partie dans la console. Cette méthode est indiquée comme dépréciée dans la javadoc, car elle est remplacée par l'interface graphique. initialiserScore permet, à chaque début de partie, d'initialiser les score conformément aux règles du jeu.

Les classes du patron Strategy ont également plus de méthode que ce que nous avions prévu. En effet, une méthode est utilisée pour chaque type de partie. De plus, pour la partie avancée, trois méthodes sont utilisée. choixJouerChien, qui permet à l'IA de prendre la décision, ou non, de jouer sa carte chien. La méthode choixJouerTaupe permet la même chose pour une carte Taupe. Enfin, la méthode choixPiocheAllie permet, en début de manche, à l'IA de prendre la décision de piocher une carte allié ou de récupérer deux graines.

La classe Joueur possède également des méthodes supplémentaires. getMainCarteByName permet de renvoyer une instance de carte à partir de son nom. La méthode nom permet de renvoyer le nom du joueur. Enfin, afficherMain permet d'afficher la main du joueur dans la console. Cette méthode est indiquée comme dépréciée dans la javadoc. L'interface graphique prend le relai de cette méthode.

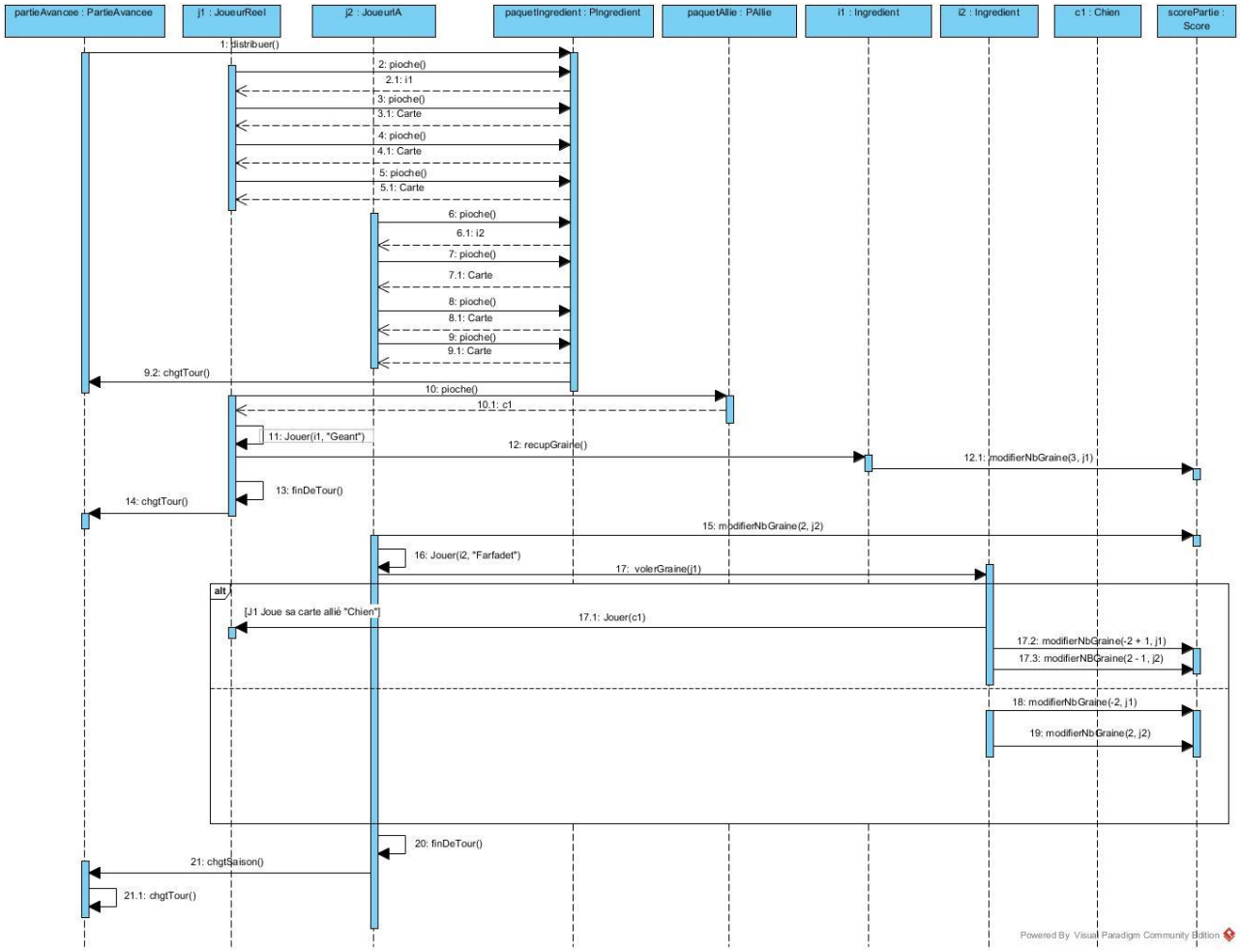
La classe JoueurIA possède deux attributs statiques qui n'étaient pas présent dans la modélisation initiale. Ces attributs sont des tableaux d'entier et de caractères. Ils nous permettent de générer aléatoirement, parmi les valeurs, le sexe et l'âge des joueurs IA de la partie. Cette classe possède de nombreuses méthodes qui n'étaient pas présente dans le premier diagramme :

-Les méthodes executeStrategyRapide et executeStrategyAvancee permettent d'exécuter la stratégie du joueur IA stockée dans son attribut de classe.

-Les méthodes de choix d'action. Ces méthodes permettent à l'IA d'appeler les méthodes de décisions pour jouer ou non une carte alliée, ou piocher une carte allié, ou récupérer deux graines.

Diagramme de séquence

Nous avons modélisé dans ce diagramme de séquence un tour de jeu, et plus précisément le premier tour d'une partie. On prend l'hypothèse que le joueur réel est le plus jeune et de sexe féminin pour cette simulation. De ce fait, il commencera la partie et donc le tour.



Dans un premier temps nous pouvons voir que "partieAvancee" appelle la méthode "distribuer" de "paquetIngredient" (1), ce qui va ensuite faire piocher à chaque joueur 4 cartes par le biais de la méthode "pioche" qui retourne la carte piochée.

Ensuite la méthode chgtTour (9.2) est appelée, déclenchant ici le début du premier tour. C'est donc J1 (le joueur réel) qui commence et qui choisit de piocher une carte dans le paquetAllie (10).

Il joue ensuite une de ses cartes, la carte "i1" et il choisit d'utiliser l'action "Géant". (11)
Le joueur appelle donc la méthode "recupGraine" de la carte "Ingrédient". Puis cette même carte appelle la méthode "modifierNbGraine" de "scorePartie" afin de modifier son nombre de graines et de lui en ajouter 3. (12)

Après cela, le joueur indique qu'il a fini son tour par le biais de la méthode "finDeTour" (13), ce qui entraîne l'appel de la fonction "chgtTour" (14) et ce qui permet de passer au tour du Joueur 2.

Le Joueur 2 ne choisit pas de piocher une carte « Allié » mais plutôt de récupérer 2 graines. On appelle donc la méthode "modifierNbGraine" afin de lui ajouter 2 graines au score.

Le Joueur 2 joue ensuite sa carte "i2" et utilise l'action "Farfadet". Il indique ensuite qu'il souhaite les voler au Joueur 1, le joueur appelle donc la méthode "volerGraine" de la carte afin de voler un certain nombre de graines au Joueur 1.

A ce moment précis, le Joueur 1 peut choisir d'utiliser sa carte « Allié » "Chien de Garde", "c1" qu'il a pioché au début de la partie afin de ne pas se faire voler une partie des graines. On a donc deux cas possible dans notre diagramme, soit J1 joue sa carte "Chien de Garde" en appelant la méthode "jouer" du J1 ce qui entraîne ensuite une modification du score par "modifierNbGraines" pour J1 ainsi que pour J2 qui va prendre en compte la carte "Chien de Garde".

Soit le J1 ne décide de rien faire un donc on va directement appeler la méthode "modifierNbGraines" de "scorepartie" afin de modifier le nombre de graines du J1 et du J2.

Ensuite le J2 informe qu'il a terminé son tour par la méthode "finDeTour" puis c'est la méthode "chgtSaison" qui va être appelée et enfin "chgtTour" afin de commencer la deuxième saison de la 1ère manche.

Etat actuel de l'application

En l'état actuel, l'application est fonctionnelle, et répond à la majorité des critères du sujet qui nous a été transmis en début de semestre. Toutes les spécificités du cahier des charges ont été implantées, et nous n'avons pas pris de liberté par rapport aux règles, bien que cela fût possible grâce à la liberté que laissaient les droits d'auteurs du jeu.

Au niveau des illustrations, vu que nous n'avions pas la possibilité d'exploiter les illustrations fournies avec le jeu et pour restituer la même expérience de jeu que la version papier, nous avons-nous même créé nos propres illustrations, à partir d'images libres de droits (obtenues grâce au site pixabay.com), ou créées de nos mains.

L'interface graphique est fonctionnelle, et permet de renseigner les paramètres nécessaires à la création de la partie (nombre de joueurs, nom, sexe et âge du joueur réel, type de partie). Elle permet également de consulter les règles du jeu (issue du document de règles disponibles en ligne), et permet de jouer aux deux types de parties.

La partie rapide permet aux joueurs de jouer pendant les quatre saisons. Le joueur le plus jeune commence la partie, comme indiqué dans le cahier des charges.

La partie avancée permet aux joueurs de jouer autant de manches qu'il y a de joueur. Ces manches sont composées de quatre saisons, et au début de chacune, une fenêtre « pop-up » modale est affichée pour inviter le joueur à choisir entre récupérer deux graines ou piocher une carte allié. Les joueurs peuvent jouer durant leur tour, une carte allié s'il en possède une, et une carte Ingrédient. En l'état actuel de l'application, il n'est pas possible de jouer une carte chien si l'on n'est pas ciblé par une carte farfadet. Le cas échéant, une fenêtre pop-up demande à l'utilisateur s'il souhaite la jouer. Il n'est possible de jouer une carte taupe qu'au début de son tour.

A la fin des deux types de parties, l'utilisateur est informé du résultat de cette dernière, par une fenêtre « pop-up » qui lui annonce le gagnant. Il est ensuite ramené à l'écran d'accueil ou il pourra quitter l'application ou relancer une partie.

Pour les deux types de parties, nous avons, grâce au patron de conception Strategy, créé trois types d'intelligences artificielles (Facile, Normal, et Difficile). Ces dernières sont fonctionnelles pour les deux types de parties, et son basés sur les mêmes algorithmes. La seule différence consiste en l'utilisation de cartes alliées, qui est implémentée dans les versions utilisées pour la partie avancée.

L'affichage des différents organes du jeu est fonctionnel (affichage de la main, affichages des scores). Cependant, il réside quelque problèmes de rafraichissement, notamment une fois le tour de l'utilisateur fini, puisque la main n'est pas rafraichie. De même, il arrive parfois que les boutons associés aux trois actions des cartes Ingrédient (Géant, Engrais, Farfadets), restent affichés alors que la carte a été jouée.

Conclusion

Nous avons, dans ce premier rapport, réalisé trois diagrammes UML, pour représenter les cas d'utilisation, les différentes classes composant notre programme et le déroulement d'un tour, respectivement via un diagramme de cas d'utilisation, de classe et de séquence. Cependant, LO02 étant l'UV dans laquelle nous avons découvert la modélisation UML et la programmation orientée objet, ces diagrammes sont susceptibles d'évoluer au cours du développement de l'application.

Une des premières choses qui n'est pas fixée est le pattern stratégie et sa modélisation. En effet, étant donné que nous n'avons jamais pratiqué ce patron de conception et que nous n'avons effectué que des recherches sur ce sujet, il nous est difficile d'être sûr de notre modélisation.

Les différents paquets de cartes sont également incertains, car nous n'avons pas encore vu comment ils se représenteraient en JAVA. A priori, ce seraient des collections de type piles. Cependant, n'ayant jamais utilisé ce genre d'objets, nous ne sommes pas sûr de son efficacité, malgré que cela nous apparait être la meilleure option.

Nous avons également des doutes sur la présence de retour pour les méthodes. En effet, il nous parait peut évident de savoir si il est plus judicieux pour la méthode pioche d'un des paquets de renvoyer la carte, ou d'appeler en son sein une méthode qui l'ajoutera au tableau de carte du joueur. Nous avons dans ce rapport privilégié le premier choix, un peu par défaut, en l'attente de pouvoir coder et choisir la solution qui nous sied le plus.

Enfin, il aurait pu être intéressant de créer un singleton pour la classe partie et pour la classe score, de manière à éviter de pouvoir créer plusieurs parties, et donc éviter les cas absurdes.

Du côté des aspects sûrs de notre modélisation, il nous parait évident d'avoir une classe centrale "Partie", qui gérera les tours des joueurs, les saisons ainsi que les manches.

Les classes principales nous paraissent également essentielles (Carte, Joueur, Score) au développement du jeu.