INF7854 - Tp1 API

Mathieu Gravel GRAM02099206[1]*, Nicolas Reynaud REYN23119308[1]**

Résumé
Le document suivant va décrire nos classes, relations et notre API utilisé pour la réalisation du jeu décrit par le TP1.

Sujets
Jeu de Carte — API — Classes et propriétés — Patrons de conception

[1] *Département d'informatique, UQAM, Montréal, Canada*
***Courriel**: gravel.mathieu.3@courrier.uqam.com
****Courriel**: reynaud.nicolas@courrier.uqam.com

Table des matières

Introduction

Ce document à pour but de présenter les classes ainsi que les choix de concéption pour un jeu de carte. Ce dernier invoque ainsi les conceptes suivants :

— Des cartes personnages

— Des cartes armes, pouvant être utilisé sur un personnage sous certains restrictions

— Des cartes enchantements, utilisé sur les armes afin de leurs accorder des éffets positifs et/ou négatifs.

— Pour avoir la liste complète des conceptes, consulter le lien suivant : `http://info.uqam.ca/~privat/INF7845/TP1/`

Le document suivant présente alors les différentes classes permettant de répondre aux différentes problèmatique prosée par le jeu. Ces classes représentant donc le modèle ainsi que l'API permettant ainsi l'interfacage avec des contrôleurs.

Classes Internes

1. Cartes de jeu

1.1 Carte (*Abstract*)

Description

Classe abstraite héritée par la totalité de nos sous-classes de cartes de jeu spécialisés, Carte nous permet de relier chaque type de carte du jeu ensemble. Cette classe nous permet aussi de leur donner des valeurs communes, tel un identifiant unique ainsi qu'une manière de les représenter aisément pour n'importe quelle interface graphique connectée.

Attributs

— SSID : static Int = 0
SSID est une varaible statique utilisé par la totalité des cartes de jeu lors de leur création. Façilement modifiable pour marcher sur un réseau, elle permet d'assurer un id unique pour la totalité des cartes dans une partie, afin de leur démarquer et identifier rapidement.

— cardID : Int {*get*;}
cardID sert à stocker la valeur de l'identifiant unique de la carte attribué par SSID.

Fonctions

- createID : Void Fonction simple qui note le SSID actuel dans cardID et ensuite l'incrémente. Ceci nous permet de donner un Id unique pour toutes les cartes d'une partie.
- abstract toJSON() : JSON Instancie une représentation JSON de la carte afin de pouvoir la transférer à l'API.
- getCardID : Int

Choix de conceptions

L'utilisation d'une classe abstraite qui est hérité par chacune des types de cartes du jeu nous permet d'homogéiniser le deck de cartes afin de pouvoir tout stocker dans la même structure. Ceci simplifie grandement toutes actions reliés à l'utilisation des cartes, tel que la pige ou l'attaque.

Une autre raison derrière le découpage des cartes du jeu en une classe abstraite et plusieurs sous-classes, plutôt qu'une seule contenant toutes les attributs et fonctions possibles (Similaire à Forge Magic) est que cette méthode limite la taille et complexité de leurs instances.

Nous avons choisi de mettre Carte comme classe abstraite puisque nous ne voulons pas qu'elle puisse être instancié par elle-même. Pour ce qui est de la question classe abstraite versus interface, le second choix ne nous aurait pas permis l'utilisation de notre méthode d'identifiant unique.

Pour ce qui est de notre variable d'identification, elle nous permet de vérifier rapidement si deux variables Cartes sont pareilles et aussi de pouvoir retrouver nos cartes plus rapidement dans des Map. (Ex : Map<int,Carte> ou int = Carte.Id)

1.2 Perso

Description

La classe Perso, sous-classe de Carte, est une représentation d'une carte personnage générique du jeu. La classe hérite de Carte peut se voir attribuer une instance d'Arme afin de noter l'arme du perso après déploiment.

Comparé aux classes Enchanements et Arme, la classe Perso n'est jamais utilisé directement lors d'une partie. En effet, Perso va plutot être sous-classé par d'autres classes tel que Combattant, Guerrier, Prêtre ou Paladin, qui eux seront réelement utilisé dans une partie. La raison derrière ce choix est qu'il permet de découpler de Perso la logique d'instanciation des différent métiers, ainsi que celle d'attaquer. Ce choix nous permet alors d'ajouter aisément de nouveaux métiers qui peuvent être des attaquant ou du support. (Ex : un classe troubadour sans arme.)

Attributs

- hp : int {*get*;}
- maxHP : int {*get*;}
- mp : int {*get*;}
- maxMp : int {*get*;}
- armePerso : Arme {*get*;}
- armeUtilisables : List<TypeArme> {*get*;}
Les armes dont le personage peut équiper.

Fonctions

\# utiliserMagie() : Void
Enlève un point de magie du personnage.

\+ recoitAttaque(Combattant attaqueur) : AttaquePersoResult
recoitAttaque enlève les points de vies de la forcce d'attaque de l'attaqueur au Perso et retourne un AttackResult qui en décrit le résultat (Le nombre de points de vies perdus, si le personnage est encore en vie et n'importe quel autre infos pertinentes pour le GUI.)

\+ estMort() : Boolean
Retourne si le personnage a encore des points de vie.

\+ equiperArme(Arme arm) : Void
equiperArme ajoute l'arme donnée en argument au personnage.

\+ soigner(Perso allie) : Result
soigner vérifie si le joueur peut actuellement utiliser un sort de soin et si oui, soigne l'allié donné et enlève le point de magie utilisé.

\# recevoirSoins() : Void
recevoirSoins remet les points de vie du perso à leur état initial.

\+ estMort() : bool Retourne si le personnage a encore des points de vie.

\# libererCarte() : List<Carte>
Enlève la carte de l'arme et ses cartes d'enchantements. Cette fonction sert seulement pour ajouter les cartes associés à l'arme au cimetière lors que le personnage meurt.

\+ getTypeArme() : TypeArme
Retourne le type d'arme utilisé par le personnage. Fonction utile lors du calcul du modificateurs d'attaque.

\+ toJSON() : JSON
Instancie une représentation JSON de la carte ainsi que son arme afin de pouvoir la transférer à l'interface du joueur.

\+ peutEtreAttaquee() : Boolean
Override de la fonction abstraite de cible. peutEtreAttaquee retourne vrai si le perso est encore en vie.

Choix de conceptions

— Notre version initiale de Perso était originellement utilisé pour instancier n'importe quel type de personnage grâce à différents Builders, ce qui limitait notre nombre de classes. Malheureusement, cette méthode de conception nous limitait quant à l'évolutivité individuelle des métiers (Aucun métier ne pouvait avoir des fonctions et attributs

uniques, tel l'ajout de sorts de magie, de forces spéciales etc...).

Afin de rendre les personnages plus modulables, nous avons décidé de modifier leur conception pour découper leurs métiers en classes individuelles et, en même temps, découpler leur logique de combat (Par exemple si nous voudrions ajouter un métier barde sans arme ou Mage qui utilise de la magie.).

Pour ce faire, nous nous sommes inspiré des travaux de Philippe Petitclerc et Zerrouk Radhia. Les interfaces Attaquant, Cible et Soigneur de M. Petitclerc, ainsi que la classe Combattant de M. Zerrouk, nous as enmmenés à défaire la logique d'attaque, de défense et de soins de la classe Perso en 2 Interfaces (Cible et Soigneur) et une classe abstraite (Combattant). (Plus de détails de conceptions sont disponible dans leurs sections.)

Ce découpage de responsabilités nous as permis d'avoir un code plus modulaire (Un perso peut maintenant ne pas pouvoir attaquer s'il extend de Perso et non d'attaquant) et plus générique (Cible nous permet de généraliser les appels de prendreDommage et la vérification de la survie du per/et ou joueur lors d'une attaque. L'utilisation de Soigneur tant qu'à elle nous permet de cacher les fonctions de soins aux persos ne connaissant pas ce sortilège.)

1.3 Arme

Description

La classe Perso, sous-classe de Carte, est la représentation des cartes d'armes du jeu. La classe hérite de Carte et est associé à Enchantement et ses implémentations puisqu'un perso doit équiper une arme dans une partie.

Attributs

type : TypeArme {get ;}

estStase : bool {get ;}

degat : int {get ;}

armeUtilise : bool {getl}

estFacile : bool {get ;}

listEnchant : List<Enchant>

Liste des enchantements placés sur la carte.Puisque les enchantements sont appliqués directement à l'arme, cette liste sert principalement pour fins de visualisation des cartes aux utilisateurs.Afin de pouvoir montrer en tout temps les enchantements placés, même si l'arme est stased, cette liste peut seulement être vidé lorsque nous envoyons l'arme au cimetière.

Liste des métiers pouvant utiliser l'arme.

- degatOrg : int

Valeur originale des dégats de l'arme. Utilisé avec typeOrg et listUtilisateursOrg pour réinitialiser la carte si l'enchantement stase est appliqué.

- typeOrg : TypeArme

Fonctions

+ forceAttaque(TypeArme arm = Neutre) : int

Retourne la force de l'attaque de l'arme. Ce chiffre est calculé selon les dégats basiques + les valeurs ajoutés/enlevés par les enchantements et finalement en ajoutant le modificateur d'attaque de l'arme contre l'arme ennemi donné en argument. (Si aucune arme est donné, la fonction utilise par défaut une arme neutre sans modificateurs.)

+ peutUtiliserArme(Perso p) : bool

Fonction qui retourne si le personnage donné peut équipper l'arme.

+ ajouterEnchant(Enchant ench) : void

AjouterEnchant applique si possible l'effet de l'enchantement sur l'arme et l'ajoute dans sa liste d'enchantements .

- reset() : Void

Reset remet les valeurs de l'arme à leur état initial. Cette fonction est seulement utilisé quand l'arme est mis en stase. (degat, listUtilisateur et type sont réinitialisé, mais listEnchants est laissé tranquille afin de garder concordance lors de l'affichage des cartes.)

- peutUtiliserArme() : Bool

- deployerArme() : Void

- peutAjouterEnchantement() : Bool

- toJSON() : JSON

Instancie une représentation JSON de la carte ainsi que ses enchantements afin de pouvoir la transférer à l'interface du joueur.

Choix de conceptions

Dans le but d'appliquer les modifications faites par les enchantements, les classes Armes et Enchantements utilisent le patron de conception Visiteur dans le but de modifier directement l'arme. (Décrit en détails dans Enchant.) La raison pourquoi le code fonctionne selon le patron visiteur est que les Enchantements ne changent pas l'état de l'arme. Elle vont plutôt lui accorder une nouvelle fonction dynamiquement (l'effet de l'enchantement.)

Puisque les enchantements modifie directements les armes, il est donc essentiel de stocker les valeurs initiales des attributs modifiables afin de pouvoir les réinitialiser si nécessaire. De toutes les approches possibles (Ex : Cloner la carte et lui appliquer ensuite les enchantements, encapsuler l'arme dans ses enchantements etc...), nous considérons cette méthode comme la plus modulaire et efficiente.

Nous pensons que cette méthode en vaut la peine parce que toutes les autres méthodes nous aurait nécessité à un moment de cloner l'arme, que ce soit pour appliquer les enchantements lors du calculs des dégats s'ils ne modifient pas réelement l'arme où lors de la réinitialisation de l'arme si les enchantements sont des décorateurs. Cette méthode nous permet dont : De modifier directement Arme par Enchant sans devoir modifier la classe Arme elle-même, d'ajouter des Enchants façilement, d'éviter de devoir recalculer plusieurs

fois les enchantements et de limiter la mémoire utilis par une arme et ses enchantements.

1.4 Enchant

Description

Classe abtraite, Enchant est l'implémentation basique des cartes enchantements dans le système. Utilisé principale sur les armes, Enchant est implémenté différement par chacune de ces sous-classes selon le type d'enchetement joué.

Attributs

— description : String

Description de l'enchantement. Cet attribut est utilisé pour détailler la différence entre chaque carte dans sa forme JSON.

Fonctions

\# abstract placerEnchant(Arme arm) : Void

Fonction abstraite, placerEnchant applique l'enchantement de la carte sur l'arme et retourne un EnchantResult.

\+ toJSON : JSON

Choix de conceptions

Afin de pouvoir placer les enchantement sur les armes, nous nous sommes basés sur le patron de conception Visiteur. (Plus précisément, sur l'exemple suivant : `https://sourcemaking.com/design_patterns/visitor/java/1`)

Lorsqu'une arme reçoit un nouvel enchantement, elle l'ajoute à sa liste d'enchantements actuels et ensuite appelle la fonction placerEnchant avec elle-même en argument. À ce stade, placerEnchant va dynamiquement modifier l'arme afin d'appliquer l'enchantement désiré. (Les divers effets seront expliqués dans leurs propres sections.)

L'utilisation de cette méthode nous permet de découper entièrement la logique des enchantements de la classe Arme, ce qui facilite l'ajout de nouvelles cartes d'enchantements si nécessaire. Comme ceci, nous pouvons ajouter n'importe quel enchantement qui touchent les variables actuelles des armes sans devoir modifier la classe Arme en aucune manière.

EnchantNeutre

Description

EnchantNeutre est une extension de Enchant dont la fonction de visite change la liste des utilisateurs de l'arme afin que tout le monde puisse l'utiliser.

Fonctions

\# placerEnchant(Arme arm) : Void

placerEnchant change le type de l'arme à Neutre.

EnchantDegatPlus

Description

EnchantDegatPlus est une extension de Enchant dont la fonction de visite augmente les dégats de l'arme de 1.

Fonctions

\# placerEnchant(Arme arm) : Void

placerEnchant augmente les dégats de l'arme de 1.

EnchantDegatMoins

Description

EnchantDegatMoins est une extension de Enchant dont la fonction de visite diminue les dégats de l'arme de 1.

Fonctions

placerEnchant(Arme arm) : Result *Même code que EnchantDegatPlus avec += transformé en -=.*

EnchantStase

Description

EnchantStase est une extension de Enchant dont la fonction de visite remet l'arme à son état initial et la place sous stase pour qu'aucun enchantement puisse être placé au futur.

Fonctions

\# placerEnchant(Arme arm) : Void

placerEnchant met l'arme sous l'effet de stase et la réinitialise.

EnchantFacile

Description

EnchantStase est une extension de Enchant dont la fonction de visite rend l'arme façile à utiliser pour tout les perso.

Fonctions

\# placerEnchant(Arme arm) : Void

placerEnchant met l'effet façile sur l'arme.

1.5 Cible

Description

L'interface Cible est une représentation des fonctions communes aux classes pouvant être attaqué lors d'une partie. Pour l'instant cette liste de classes se limite à Perso et Joueur.

L'utilisation de l'interface nous permet alors de gérer l'attaque de ces deux types de comportements sans devoir doubler la logique d'appel.

Fonctions

\+ abstract peutEtreAttaque() : Boolean

Retourne si la cible peut actuellement être attaqué.

\+ abstract recoitAttaque(Combattant attaque) : Resultat

Traite le nombre de dégats pouvant petre infligé par l'attaqueur (hp pour un perso, cartes pour un joueur) et retourne un Résultat (Soit un AttaquePersoResult, soit un AttaquePlayerResult.)

\+ abstract estMort() : Boolean

Retourne si la cible est encore en vie.

Choix de conceptions

Idée inspiré en majeur partie de l'API de Philippe Pépos Petitclerc et Mehdi Ait Younes, Cible nous permet de rendre la logique de pouvoir être attaqué générique entre les classes. À la place d'avoir plusieurs fonctions qui traite séparément si l'attaqueur attaque le joueur ou un personnage (comme dans notre version initiale de l'API), nous pouvons maintenant lier l'entière logique du ciblage à une interface directe.

1.6 Soigneur

Description

L'interface Soigneur regroupe toutes les fonctions associés au sortilège de Soins dans le Jeu. Toutes classes l'implémentant pourra alors vérifier s'il peux utiliser actuellement le sortilège et, si oui, le déployer pour réinitialiser la vie de son allié.

Fonctions

+ abstract soigner(Perso allie) : SoinsResult soigner confirme que le soigneur a encore assez de magie pour le sort et ensuite soigne l'allié. Il renvoie alors un SoinsResult décrivant le résultat.
+ abstract peutSoigner(Perso p) : boolean peutSoigner confirme que le soigneur est capable actuellement d'utiliser le sort de soins (Voir s'il a encore assez de magie.)

Choix de conceptions

Dans la version initiale de l'API, la logique des soins était directement dans Perso. Cette méthode fonctionnait correctement, puisque le métier qui ne peut utiliser le sortilège n'a pas de MP.

Nous avons décidé de découper la logique de cette magie en interface après avoir vu le travail de Philippe Pépos Petitclerc et Mehdi Ait Younes. En effet, cette méthode nous permet de s'assurer le contrôle sur le type de classe pouvant utiliser le sortilège, ce qui nous permet d'ajouter aux jeu des classes tel que des sorciers, qui utiliserait leur magie non pour soigner mais plutôt pour attaquer (Sinon cette classe pourrait aussio soigner, ce qui irait à l'encontre des habiletés qu'elle devrait avoir.).

Ce choix de design nous permettrait aussi d'ajouter plusieurs autres types de sortilèges dans le jeu, sans toutefois devoir les rendre accèssible à chaque personnage.

1.7 Combattant

Description

Extension abstraite de Perso, Combattant regroupe la logique d'attaquer une Cible dans le jeu.

Cette classe est ensuite étendu par Guerrier, Pretre et Paladin pour instancier nos personnages avec la logique d'attaque.

Fonctions

+ forceAttaque(TypeArme ta) : int forceAttaque retourne le nombre de dégats éffectué par ce combattant contre quelqu'un utilisant le type d'arme donné en argument.
+ Attaque(Cible c) : Resultat Attaque appelle la fonction recoitAttaque de Cible et retourne son résultat. (Il recoit soit un AttaquePlayerResult, soit un AttaquePersoResult qu'il met en forme Resultat pour fins de communication.) (Puisque dans la version actuelle des vue + controlleurs utilisé, seul la description unique du résultat est utilisé, nous n'avons pas besoin de retourner la forme spécifique.)

Choix de conceptions

Originellement, nous avions l'entière logique de Combattant à l'intérieur de Perso. Toutefois, après avoir vu l'API dees équipes de Philippe Pépos Petitclerc et Zerrouk Radhia qui nous avait inspiré à découpler la logique de Ciblage et de sortilèges des Persos, nous avons décidé de libérer aussi la logique d'attaque. (Afin qu'on puisse aouter au futur si nécessaire des classes telles que des Bardes qui font seulement des effets de status ou quelque chose de similaire.)

Notre seconde version était une interface, similaire à Soigneur, mais nous avions décidé qu'il serait plus sage de la transformer en classe abstraite puisque ces fonctions utilise le même code pour tout les métiers actuelles. (Nous n'aimions pas avoir 3 fonctions dans 3 classes qui utilise le même code.)

1.8 Guerrier

Description

Extension de Combattant, Guerrier nous permet d'instancier un Personnage pouvant attaquer avec les valeurs d'HP, de MP et d'armes utilisables lié à un Guerrier dans Regle.

Choix de conceptions

Nous avons fait que Guerrier soit une extension de Combattant sans implémenter Soigneur puisque ceci nous permet de lui attribuer la logique d'une Carte, d'un Perso, d'un combattant et d'un Guerrier en même temps.

1.9 Pretre

Description

Extension de Combattant, Pretre nous permet d'instancier un Personnage pouvant attaquer avec les valeurs d'HP, de MP et d'armes utilisables lié à un Pretre dans Regle.

Fonctions

+ soigner(Perso allie) : SoinsResult
+ peutSoigner(Perso p) : boolean

Choix de conceptions

Nous avons fait que Pretre soit une extension de Combattant qui implémentee Soigneur puisque ceci nous permet de lui attribuer la logique d'une Carte, d'un Perso, d'un combattant, d'un soigneur et d'un Pretre en même temps.

1.10 Paladin

Description

Extension de Combattant, Paladin nous permet d'instancier un Personnage pouvant attaquer avec les valeurs d'HP, de MP et d'armes utilisables lié à un Paladin dans Regle.

Fonctions

+ soigner(Perso allie) : SoinsResult
+ peutSoigner(Perso p) : boolean

Choix de conceptions

Nous avons fait que Pretre soit une extension de Combattant qui implémentee Soigneur puisque ceci nous permet de lui attribuer la logique d'une Carte, d'un Perso, d'un combattant, d'un soigneur et d'un Pretre en même temps.

2. Classes de joueur et Deck

2.1 PersoFactory

Description

PersoFactory est une classe basé sur le concept de Factory qui permet d'instancier aisément les personnages nécessaires pour une partie. Cette classe est instancié dans Deck au début d'une partie afin de contruire les cartes plus rapidement.

Fonctions

— creetSetGuerriers(int nbCopies) : List<Guerrier> Instancie une liste de cartes Guerriers de nbcopies.

— creetSetPretre(int nbCopies) : List<Pretre> Instancie une liste de cartes Pretres de nbcopies.

— creetSetPaladins(int nbCopies) : List<Paladin> Instancie une liste de cartes Paladins de nbcopies.

Choix de conceptions

PersoFactory nous permet de défaire la logique de créations des cartes personnage de la classe Deck, qui doit pouvoir en instancier au début d'une partie. En effet, puisque que le Deck n'a pas besoin de savoir comment chacune des cartes est réellement créé, PersoFactory permet de généraliser leur constructions dans une fonction hors de créerDeck() et d'encapsuler la logique d'instanciation afin de simplifier le modèle et de limiter les connaissances inter-classes inutiles.

Nous avons décidé de transformer nos PersoBuilders en un seul PersoFactory pour fins de simplification du code et de la logique d'instantiation.

2.2 ArmeFactory

Description

ArmeFactory est une classe basé sur le concept de Factory qui permet d'instancier aisément les armes nécessaires pour une partie. Cette classe est instancié dans Deck au début d'une partie afin de contruire les cartes plus rapidement.

Fonctions

— creerSetArmes(int nbCopies,int degats) : List<Arme> Instancie une liste de cartes d'armes de chaque type avec nb copies fesant chacun un nombre spécifié de points de dommages.

Choix de conceptions

ArmeFactory nous permet de défaire la logique de créations des cartes d'armes de la classe Deck, qui doit pouvoir en instancier au début d'une partie. En effet, puisque que le Deck n'a pas besoin de savoir comment chacune des cartes est réellement créé, ArmeFactory permet de généraliser leur constructions dans une fonction hors de créerDeck() et d'encapsuler la logique d'instanciation afin de simplifier le modèle et de limiter les connaissances inter-classes inutiles.

2.3 EnchantFactory

Description

EnchantFactory est une classe basé sur le concept de Factory qui permet d'instancier aisément les enchantements nécessaires pour une partie. Cette classe est instancié dans Deck au début d'une partie afin de contruire les cartes plus rapidement.

Fonctions

— creerSetEnchants(int nbCopies) : List<Enchant> Instancie une liste de cartes d'enchantements de chaque type avec nb copies.

Choix de conceptions

EnchantFactory nous permet de défaire la logique de créations des cartes enchantements de la classe Deck, qui as besoin de les instancier au début d'une partie. En effet, puisque que le Deck n'a pas besoin de savoir comment chacune des cartes est réellement créé, EnchantFactory permet de généraliser leur constructions dans une fonction hors de créerDeck() et d'encapsuler la logique d'instanciation afin de simplifier le modèle et de limiter les connaissances inter-classes inutiles.

2.4 Deck

Description

La classe Deck est une représentation du deck de cartes d'un joueur au courant d'une partie. Il contient toutes les informations pertinentes relié au cartes pas encore pigés du joueur tel que leur instanciation, leur piochages etc...

Attributs

- cartespioches : List<Card>
  Liste des cartes pas encore piochés du joueur.

Fonctions

- initialiserDeck() : Void
  Appelé par le constructeur,InitialiserDeck crée une instance de ArmeFactory,EnchantFactory et chacun des builders de personnages et leur demandes le nombres de cartes définis dans la classe Règles pour chaque type qu'il ajoute ensuite à cartespioches.

+ piocherCarte(int nbCartes) : List<Card>
  PiocherCarte prend le nombre minimum entre la taille de cartespioches, 5 (la taille minimale d'une main) et nbCartes, retire ensuite ce chiffre de cartespioches et retourne les cartes enlevés.

+ dommageJoueur(int nbCartes) : List<Card>
  DommageJoueur prend le nombre minimum entre la taille de cartespioches et nbCartes, retire ensuite ce chiffre de cartespioches et retourne les cartes enlevés qui seront envoyés au cimetière du joueur.

+ carteRestantes() : int
  Retourne le nombre de cartes restantes à piocher.

+ viderDeck() : Void
+ deckEstVide() : boolean
+ toJSON : JSON

  Instancie une représentation JSON du deck du joueur afin de pouvoir la transférer à l'interface de l'utilisateur.

Choix de conceptions

Le découpage des fonctions associés au deck d'un utilisateur hors de sa classe Joueur nous permet de simplifier les contrôles et de limiter les actions possibles d'un joueur envers son deck.

Afin de garder les connaissances sur la manière de créer les cartes le plus encapsulés possibles, Deck utilise des instances de ArmeFactory, EnchantFactory et des implémentation de PersoBuilder pour construire le deck. Ce découpage de la logique d'initialisation en différentes classes permet ainsi de garder le code de Deck simple, modulaire et façilement modifiable.

2.5 Joueur

Description

La classe joueur est une représentation de toutes les cartes et actions disponibles à un utilisateur au courant d'une partie. Cette classe, servant de proxy pour tout les appels de l'API Jeux, permet d'effectuer les coups désirés par le joueur et ses résultats tout en masquant leur implémentation.

En autre termes, Joueur nous permet d'appliquer réelement les actions demandés par l'API Jeux, sans lui donner accès à des choses qu'il ne devrait pas avoir. (Ceci nous permet de limiter le spossibilités de triches par des joueurs malins.)

Attributs

— idJoueur : int {get ;}
— cardDeck : Deck

  Les cartes restantes à piocher du joueur.

— main : Map<int,Card> {get ;}
— cartesEnJeu : Map<int,Card> {get ;}

  Les cartes persos déployés dans le jeu selon leur identifiant unique.

— cimetiere : Map<int,Card> {get ;}

  Les cartes mortes du joueur.

Fonctions

+ aPerdu() : bool

  Retourne vrai si le joueur a au moins une carte soit dans son deck, soit dans sa main ou soit déployé.

+ MAJPlancher() : Void

  Enlève les perso mort du terrain et les place au cimetière.

+ carteDansJeu(Carte c) : boolean

  Retourne si la carte est actuellement dans le jeu déployé du joueur.

+ carteDansMain(Carte c) : boolean

  Retourne si la carte est actuellement dans la main du joueur.

+ peutEtreAttaque() : Boolean

  Retourne vrai si le joueur a aucune carte sur le terrain.

+ recoitAttaque(Combattant attaquant) : Resultat

  recoitAttaque enlève de la pioche le nb de cartes égale aux dégats fait par le combattant.

+ estMort() : Boolean

  Retourne vrai si le joueur n'a plus aucune cartes.

+ defausserCartes(List<Carte> defausse) : Resultat

  DefausserCartes retire les cartes dites en argument et les met dans le cimetière.

+ piocher() : Resultat

  Piocher appelle la fonction PiocherCarte de Deck avec en argument le maximum entre 0 et 5 - le nombre de cartes dans la main. (Puisque peut piocher jusqu'à qu'il a 5 cartes dans sa main selon l'énoncé.)

+ attaque(Combattant attaqueur, Cible attaque) : Resultat

  Attaque joue le coup d'attaque demandé et retourne le résultat.

  Cette fonction est appelé pour attaquer un Perso ou un joueur.

+ ajouterEnchants(List<Enchants> enchs, Carte carteTouchee) :Resultat

  AjouterEnchants applique les enchantements choisis par le joueur à la carte donné et retourne le résultat.

+ placerPerso(Perso personnage, Arme arm, List<Enchant> ench) : Resultat

  Déploie une carte personnage, arme et enchantements si donné de la main du joueur et retourne un résultat décrivant l'acte.

  Puisque l'API à deja confirmé que le déploiement est correct (arme utilisé + enchantements), les enchantements sont appliqués directement sur l'arme ainsi que l'arme sur le Perso.

+ declarerForfait() : Resultat

  DéclarerForfait place toutes les cartes du joueur au cimetière pour marquer qu'il a perdu et retourne un Resultat dictant qu'il a perdu.

+ soignerPerso (Soigneur soigneur, Carte soignee) : Resultat

  SoignerPerso effectue l'action de soigner un personnage si possible et retourne un Resultat décrivant l'acte.

+ toJSON : JSON

  Instancie une représentation JSON du deck du joueur afin de pouvoir la transférer à l'interface de l'utilisateur.

Choix de conceptions

Comparé à la version initiale de joueur décrit dans le livrable 1, notre nouvelle version fonctionne avec l'utilisation des instances des cartes plutôt que leur identifiant unique. Ceci est dû à un changement de vision apporté à la logique des retours du modèle après avoir reçu la correction du livrable 1.

Originellement, l'API ne retournait jamais une instance d'une carte, seulement son JSON. Ce choix de design était

basé sur le fait que l'API doit marcher avec des joueurs à distance selon l'énoncé donné, ce qui nécessite d'avoir un moyen de décrire une carte à un joueur distant. Toutefois, après avori discuté avec M. Privat, nous avons décidé d'ajouter des getters de cartes dans l'API afin de retourner les cartes elle-mêmes. Ce choix de design permet de simplifier leur utilisation dans une vue tel un GUI. Pour le cas des joueurs distants, il est encore possible au contrôlleur d'avoir le format JSon de la carte si nécessaire.

Puisque nous avons laissé la possibilité à la vue de recevoir les cartes, nous avons du alors changé la structure du joueur et de Jeu. Maintenant, seul Joueur est capable d'accéder aux fonctions des cartes qui permet de modifier l'état. Ceci nous permet de s'assurer qu'un utilisateur malin ne se connecte pas à l'API directement afin de tricher en modifiant les cartes directement.

3. Classes Règles du jeu

3.1 Enum TypeArme

Description

TypeArme est un enum représentant chacune des armes existantes dans le jeu, ainsi que leurs forces et faiblesses. Cet enum nous permet ainsi de vérifier aisément le triangle d'attaque ainsi que la neutralité de l'arme sans devoir noter la logique directement dans les cartes. Voici les valeurs possibles de notre enum (Les arguments représentent respectivement le nom de l'arme ainsi que sa force et sa faiblesse.) :

- Contondant("Contondant","Perforant ","Tranchant")
- Perforant("Perforant ","Tranchant","Contondant")
- Tranchant("Tranchant ","Contondant","Perforant")
- Neutre("Neutre","","")

Attributs

- nom : String
- force : String
- faiblesse : String

Fonctions

+ calculModificateur(TypeArme armeEnnemi) : int Compare le type d'arme de l'arme contre celle donné en argument et retourne la valeur do modificateur d'attaque selon la formule suivante :
  - Si this.nom == armeEnnemi.faiblesse, alors 1.
  - Sinon si this.faiblesse == armeEnnemi.nom, alors -1.
  - Sinon 0.

Choix de conceptions

Raison derrière la classe :

Selon les règles du jeu, le système d'Armes fonctionne selon trois types clairement définis. Étant donné que chaque arme du jeu fonctionne selon ces règles pour le type et triangle d'attaque, il est plus efficiant de séparer les règles des type d'armes de leurs implémentations.

Ce découpage permet ainsi d'éviter de devoir étendre notre classe d'arme en une sous-classe pour chaque type d'arme différent. En plus de limiter la redondance de classes et de code, l'utilisation de l'enum ArmeType nous donne un emplacement parfait pour noter notre fonction de vérification de forces et de faiblesses afin de la retrouver plus façilement en cas de besoin.

3.2 Règles

Description

Classe statique constituée seulement d'attributs statiques finaux, Règles sert à contenir les valeurs paramétrisables du jeu dans un endroit façile d'accès pour n'importe quelle classe.

Attributs

+ GUERRIERHP : int = 5
+ GUERRIERMP : int = 0
+ PRETREHP : int = 3
+ PRETREMP : int = 3
+ PALADINHP : int = 4
+ PALADINMP : int = 1
+ CARTEGUERRIER : int = 4
+ CARTEPRETRE : int = 4
+ CARTEPALADIN : int = 2
+ CARTEARMEUN : int = 2
+ CARTEARMEDEUX : int = 2
+ CARTEENCHANTEMENT : int = 2
+ CARTEMAIN : int = 5

Choix de conceptions

Raison derrière la classe :

Théoriquement, nous n'aurions pas besoin de cette classe pour implémenter le jeu de cartes. En effet, chacuns de ces attributs sont utilisés à un seul endroit spécifique et pourrait dont être retranscrit directement à ces emplacements. La raison pourquoi ils ont plutôt été regroupé à cet emplacement est que ce design améliore l'évolutivité du modèle. Il nous serait même possible avec ce design de créer une fenêtre d'options dans le jeu avec ces valeurs afin de pouvoir les modifier entre 2 parties par le joueur.

Statique :

Puisque la classe détient seulement des attributs non-modifiables qui sont seulement lues, il n'y a aucune raison pourquoi elle aurait besoin d'être instancié dans le système.

4. API

4.1 Jeux

Description

Jeux est la classe principal de notre modèle du système. Mis avec la classe Resultat et ses implémentations, il rempli la totalité des fonctions de l'API utilisé pour les controlleurs du jeu (Qu'ils soivent humain ou IA).

Attributs

- joueurList : List<Joueur>

- partieEnMarche : Boolean
- joueurTour : int

  Int représentant le joueur actif dans la partie.

Fonctions

Dû aux grand nombres de fonctions de l'API, nous avons décidé de ne pas retranscrire les déclarations du code source dans le document.

La raison derrière un si grand nombre de fonctions est que nous voulions avoir une fonction pour chaque action possible, une fonction pour valider chacune des actions demandés, des fonctions pour démarrer/arrêté une partie ainsi que savoir le joueurAct et finalement des fonctions pour retourner l'état du jeu aux controlleurs (Une pour des controlleurs d'un serveurs Web en JSON et une pour un controlleur graphique locale)

Choix de conceptions

Utilisé comme facade pour les contrôlleurs, Jeux nous permet de centraliser tout les morceaux de l'API publics aux joueurs en un seul endroit façile d'accès.

Puisque nous voulions que l'API puisse être utilisé par n'importe quel types de joueurs (joueurs locaux, joueurs distants qui communique par web, AI), nous avons créé Jeux pour être capable de communique avec n'importe quel type de contrôlleur possibles. (Controlleurs qui fonctionne par Command pattern, Controlleur Web, controlleur avec le patron de conception stratégie etc...).

Pour ce faire, nous avons implémenté dans Jeux une fonction pour chaque action possible (piocher,déployer etc...) ainsi qu'une fonction pour confirmer que le coup demandé par le contrôleur est réalisable.

Nous avons aussi mis des getters de Cartes pour les main et terrains des joueurs ainsi qu'un getter du nombre de pioche restante. Ces getters, mis avec les getters Json du design original, nous permet d'envoyer une représentation d el'était du jeu utilisable pour tout types de contrôlleurs.

La raison pourquoi chacune des actions jouables dans Result retourne un Resultat et non un AttaquePlayerResult,SoinsResult etc... est pour traiter les cas de triches. En effet, même si nous avons des fonctions pur valider si un coup est correct, il serait possible pour un joueur malin (par exemple qui essaie de faire un AI pour un TP mais ne veut pas mettre 100% d'effort) qui les saute, pour éviter cela, les actions analyse le coup pour s'assurer qu'il n'y a aucune triche et retourne soir le bon Resultat soir un refuseeResult. Puisque nous utilisons actuellement seulement la description de Resultat, nous n'avons pas besoin d'avoir sa forme plus détaillé.

Si, au futur, nous devrions placer l'API dans un système sans triche possible coté client, alor snous pourrions remplacer chaque Resultat avec sa forme plus complexe et ainsi utilise la totalité de sinfos pour l'expérience graphique. (Ex des animations, des effets 2d etc...)

4.2 Resultat – *Interface*

Description

De nature simple, l'interface Resultat et ses implémentations permettent au système de communiquer expressivement et clairement le resultats des coups joués par les utilisateurs. Grâce à son interface simple, Resultat permet de donner aux utilisateurs les données pertinentes aux résultats de leurs coups tel que sa description, ses effets graphiques, s'il a fonctionné etc...

Dans la version actuelle du code (Modèle + Vue + Contrôlleur montré lors de la présentation), seul la description est utilisé. Selon les besoins pour les vues futures, la classes pourrait recevoir plus de données afin de faires des animations plus intéressantes.

Fonctions

+ coupAMarche() : Bool

  Étant donné que certaines actions peuvent échouer (Ex : Une attaque à dégats négatifs), il est nécessaire pour chaque résultat de savoir si tout s'est bien déroulé.

+ getDescription() : JSON

  GetDescription donne une description détaillé du coup joué et de son résultat pour les interfaces textuelles.

+ coupJouePar() : int

  CoupJouePar permet de noter quel joueur a placé le coup. Cette variable peut être utilise pour des controllers avec fichiers logs.

Choix de conceptions

Grâce à l'utilisation d'une interface commune pour toutes les communications avec les controlleurs (si on ne compte pas GetEtatJeu), on peut s'assurer que les données tranmises aux contrôlleurs sont tous du même type et sont regroupés ensemble afin de pouvoir les modifier aisément afin d'ajouter/enlever des informations.

4.2.1 AttacquePersoResult

Description

Implémentation de Result pour les coups d'attaques dans une partie (Attaquer une carte.)

Attributs

+ dommageRecu : int {get ;}

  Le points de dégats attribué par l'attaque

+ attaqueur : int {get ;}

  L'identifiant unique du joueur qui a éffectué l'action.

+ idCarteAttack : int {get ;}

  L'identifiant unique de la cible

+ idCarte : int {get ;}

  L'identifiant unique de la carte attaqueur.

+ attaqueATue : bool {get ;}

  Bool déterminant si l'attaque a tué la cible.

+ description : String {get ;}

  String décrivant le résultat de l'aciton

4.2.2 AttacquePlayerResult
Description

Implémentation de Result pour les coups d'attaques dans une partie (Attaquer un joueur.)

Attributs

+ dommageRecu : int {get ;}
  Le points de dégats attribué par l'attaque
+ attaqueur : int {get ;}
  L'identifiant unique du joueur qui a éffectué l'action.
+ idAdversaire : int {get ;}
  L'identifiant unique de la cible
+ idCarte : int {get ;}
  L'identifiant unique de la carte attaqueur.
+ attaqueATue : bool {get ;}
  Bool déterminant si l'attaque a tué la cible.
+ description : String {get ;}
  String décrivant le résultat de l'aciton

4.2.3 EnchantResult
Description

Implémentation de Result pour les enchantements effectué dans une partie.

Attributs

- joueurId : int {get ;}
- CarteEnchant : Carte {get ;}
  La carte qui a été enchanté.
- coupAFonctionne : boolean {get ;}
- enchant : List<Enchant> {get ;}
  Liste des enchantements utilisé.
- description : String {get ;}

4.2.4 PersoDeploieResult
Description

Implémentation de Result pour les déploiements.

Attributs

- joueurId : int {get ;}
  L'identifiant unique du joueur qui a agi.
- coupAFonctionné : boolean {get ;}
- description : String {get ;}

Fonctions

+ Constructeur(int jId,boolean coupCorrect Carte Perso)
  Le constructeur note l'id du joueur, si le coup a marché et note dans la description les info de la carte et de son arme (et en même temps ses enchantements.)

4.2.5 RefuseResult
Description

Implémentation de Result pour les actions refusés. Que ce soit que le joueur n'ai pas de droit d'effectuer l'action ou qu'elle ne suit pas les règles du jeu, RefusedResult va permettre au joueur de savoir pourquoi.

4.2.6 SoinResult
Description

Implémentation de Result pour les actions de soin.

Attributs

- joueurId : int {get ;}
  L'identifiant unique du joueur qui a agi.
- coupAFonctionné : boolean {get ;}
- description : String {get ;}
- healerId : int {get ;}
  L'identifiant unique de soigneur
- persoSoigneeId : int {get ;}
  L'identifiant unique du joueur qui a été soignée.

4.2.7 PiocheResult
Description

Implémentation de Result pour les actions de pioche.

Attributs

- joueurId : int {get ;}
  L'identifiant unique du joueur qui a agi.
- coupAFonctionné : boolean {get ;}
- description : String {get ;}
- cartesId : int {get ;}
  L'identifiant unique des cartes piochés.

4.2.8 DefausseResult
Description

Implémentation de Result pour les actions de défausse.

Attributs

+ cartesId : List<Integer> {get ;}
  Identifiant unique des cartes défaussé.
+ joueurId : int{get ;}
+ description : string {get ;}
+ coupAFonctionne : bool{get ;}

4.2.9 FinPartieResult
Description

Implémentation de Result pour décrire au joueur qui a gagné la partie.

Attributs

- joueurId : int {get ;}
  L'identifiant unique du joueur qui a agi.
- coupAFonctionné : boolean {get ;}
- description : String {get ;}
- joueurGagne : int {get ;}
  L'identifiant unique du joueur qui a gagné.

4.2.10 ForfaitResult
Description

Implémentation de Result pour décrire au joueur qui vient de déclarer forfait dans la partie.

Attributs

- joueurId : int {get ;}
  L'identifiant unique du joueur qui a agi.
- coupAFonctionné : boolean {get ;}
- description : String {get ;}
- healderId : int {get ;}
  L'identifiant unique de soigneur
- joueurPerdu : int {get ;}
  L'identifiant unique du joueur qui a perdu.

5. Conclusion

Les principes demandés semblent être tous respecté, qui plus est l'évolutivité est facilitée notammement grâce aux plusieurs choix de conceptions tel que la classe Règle, les interfaces Soigneur et Cible, la classe abstraite Combattant etc... Qui plus est, nous avons voulu intégré directement dans ce document les évolutions que nous pourrions ajouter au jeu, avec par exemple le cimetière, qui permetterait ainsi d'ajouter différente stratégie de jeu sans pour autant en changer le concept de base.

En conclusion, nous pensons que notre système proposé fonctionne pour tout les use-cases décrits et est assez modulaire pour résister aux futurs changements.

6. Remerciments

— Jean Privat : Client principal pour le tp1 et aide majeur à la compréhension de la problématique

— Design Patterns : Elements of Resuable Object-Oriented Software : Source des design patterns utilisé dans l'API.

— Philippe Pépos Petitclerc et Mehdi Ait Younes : Leur API a été l'influence principale pour la création des classes Cible,Soigneur et Combattant. Leur classes Tour nous ont aussi influencé lors de la conception de notre controlleur utilisé pour la présentation.
`info.uqam.ca/~privat/INF7845/TP1/PY.pdf`

— Zerrouk Radhia et Belarbi Faten : Leur livrable 1 nous a grandement influencé pour notre création de la classe Combattant.
`info.uqam.ca/~privat/INF7845/TP1/PY.pdf`

— Abdelkarim Belkhir et Dylan Lebatteux : Leur livrable 1 nous a grandement influencé pour notre création de la classe Combattant.
`info.uqam.ca/~privat/INF7845/TP1/BL.pdf`

— Rubin Jehan et Haas Ellen : Leur livrable 1 nous a influcné à modifier l'API Jeux afin d'y ajouter les fonctions de validations.
`http://info.uqam.ca/~privat/INF7845/TP1/RH.pdf`

— Kaamelott : Source des petits gags visuels insérés dans nos présentations du projet.

7. Diagramme

Voir la prochaine page.