INF7854 - Tp1 API

Mathieu Gravel GRAM02099206[1]*, Nicolas Reynaud REYN23119308[2]**

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*
[2] *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 Long = 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 : Long {*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.

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.

1.2 Perso

Description

La classe Perso, sous-classe de Carte, est une représentation d'une carte personnage du jeu, tel qu'un prêtre ou guerrier. La classe hérite de Carte et est associé à Arme puisqu'un perso doit équiper une arme dans une partie. Comparé aux classes d'enchantements, la classe perso peut être utilisé pour créer n'importe quel type de personnage d'une une partie, puisque la différence entre chaque personnages est minime.

Attributs

- hp : int
- maxHP : int
- mp : int
- armePerso : Arme
- typePerso : TypePerso

  Le nom du personnage. Utilisé afin de les différencier coté utilisateur.

Fonctions

+ forceAttaque() : int

  forceAttaque calcule le dégat causé par l'arme du personnage. La fonction appelle la fonction degats de son arme et retourne le résultat.

+ forceAttaque(Perso ennemi) : int

  forceAttaque calcule le dégat causé par l'arme du personnage. La fonction prend la valeur de forceAttaque(), lui ajoute le modificateur d'attaque de son arme contre celle de l'ennemi et retourne le dommage résultant.

+ prendreDommage(Int dmg) : AttackResult

  prendreDommage enlève les points de vies reçus en argument 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.)

+ ajouterEnchant(Enchant ench) : Result

  AjouterEnchant ajoute l'enchangement donné en argument à l'arme du Perso. Le résultat va être soit un EnchantResult si l'enchantement a marché ou RefusedResult si l'arme ne peut pas être enchanté.

+ placerArme(Arme arm) : Void

  PlacerArme 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.

Choix de conceptions

— Étant donné que les seuls différences entre chaque personnages sont leurs points de vie, leurs points de magie, leur noms et les types d'armes qu'ils peuvent utiliser (Qui ne peux pas elle-même être noté dans le personnage selon notre choix de conception pour les armes.), nous n'avons pas vu de besoin de créer des sous-classes pour chaque types de personnages.

— Puisque chacun des types de personnages suivent toujours les mêmes règles clairement définis, nous avons créé une classe,PersoBuilder, spécialement concus pour instancier les persos correctement. Ceçi nous permet de garder la classe Perso simple lors de son initialisation et utilisation. Ce choix de design nous permet aussi de pouvoir façilement ajouter d'autres types de personnages (Il faut seulement ajouter un nouveau nom à TypePerso

et un nouveau Builder avec les bons paramètres.) sans briser le reste du programme.

— Puisqu'une arme peut changer de métiers acceptables grâce à un enchantement, nous avons poussé toutes logique sur la vérification du métier pour une arme dans la classe Arme.

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 ;}

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.

listUtilisateurs : List<TypePerso>

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

- listUtilisateursOrg : List<TypePerso>

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(TypePerso personne) : bool

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

+ ajouterEnchant(Enchant ench) : Result

AjouterEnchant applique si possible l'effet de l'enchantement sur l'arme et l'ajoute dans sa liste d'enchantements et retourne un EnchantResult. Si l'arme est stased, alors AjouterEnchant retourne un RefusedResult.

- 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.)

- 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.)

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 le sapproches possibles (Ex : Cloner la carte et lui appliquer ensuite les enchantements, encapsuler l'arme dans ses encahntements etc...), nous considérons cette méthode comme la plus modulaire et efficiente.

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 à jouer.

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) : Result

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

+ toJSON : JSON

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

Choix de conceptions

Afin de pouvoir placer les enchantement sur les armes, nous nous sommes basés sur le patron de conception Visiteur. 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.

Cette méthode nous permet en plus de modifier directement les données de l'arme touché lors de l'application de l'enchantement, ce qui sauve considérablement de temps lors du calcul des dégats. Le seul point négatif à cette méthode est que nous devons ajouter un clone de toutes les variables qui peuvent être modifiés par un enchantement, afin de pouvoir les réinitialiser si une carte stase est joué.

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) : Result

Data : L'arme donc nous voulons appliquer l'enchantement.
Result : Result décrivant le résultat pour le controlleur.
begin
  if *!arm.stased* **then**
    arm.listUtilisateurs = La liste entière de TypePerso.;
    Result resultat = (...) `/* Instantiation des données du resultat. */`
  else
    Result resultat = new ResultatRefused();
  end
  return *resultat*;
end

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) : Result

begin
  if *!arm.stased* **then**
    arm.degat += 1;
    Result resultat = (...) `/* Instantiation des données du resultat. */`
  else
    Result resultat = new ResultatRefused();
  end
  return *resultat*;
end

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 pseudo-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) : Result

begin
  if *!arm.stased* **then**
    arm.stased = Vrai ;
    arm.listEnchants.vider();
    arm.reset();
    Result resultat = (...) `/* Instantiation des données du resultat. */`
  else
    Result resultat = new ResultatRefused();
  end
  return *resultat*;
end

2. Classes de joueur et Deck

2.1 PersoBuilder

Description

PersoBuilder est une implémentation du patron de conception Builder pour fins de créations des cartes personnages. PersoBuilder est constitué d'une classe abstraite implémenté par une sous-classe spécifique pour chaque type de personnage. Chacune de ces sous-classes est instantié dans la classe Deck d'un joueur et est ensuite utilisé pour batir les métiers correctement.

Attributs

- hp : Int
- mp : Int
- type : TypePerso

Fonctions

+ batir() : Perso Instancie une nouvelle carte personnage selon les valeurs donnés.
+ setHp(int vie) : Void Donne une valeur à la vie du personnage. Cette sera sera utilisé lors de la création pour hp et maxHP.

+ setMp(int magie) : Void Donne une valeur à la vie du personnage. Cette sera sera utilisé lors de la création pour mp.

Choix de conceptions

L'utilisation d'une classe de construction nous permet d'instancier aisément chacune des cartes personnages pour un nombre répété de fois, tout en permettant de limiter les connaissances internes de la classe Deck sur la manière de batir chacune des cartes de métiers.

Grâce à PersoBuilder et ses instances, il est possible d'instancier autant de cartes de personnages en début de partie rapidement de manière modulaire.

Nous avons décidé d'utiliser une classe abstraite plutôt qu'une interface puisque chacune des sous-classes peuvent utiliser la même fonction afin de batir leurs cartes.

Comparé aux cartes Armes et enchantements qui sont batis d'un seul coup par des classes "Factory", nous avons laissé la construction des cartes persos un peu plus manuelles puisque le nombres de cartes de chaque métiers diffère selon les règles du jeu.

GuerrierBuilder

Description

GuerrierBuilder est une extension de la classe PersoBuilder qui va lire les règles de la classe Règles et ensuite batir un personnage selon ces modalités.

Attributs

- hp : Int
  Instancié à hp = Règles.GUERRIERHP.
- mp : Int
  Instancié à mp = Règles.GUERRIERMP.
- type : TypePerso
  Instancié à type = TypePerso.Guerrier.

PretreBuilder

Description

GuerrierBuilder est une extension de la classe PersoBuilder qui va lire les attributs PretreHP et PretreMP de la classe Règles pour ensuite batîr un Perso avec les attributs HP et MP correspondants, ainsi que le typePerso == "Prêtre"

Attributs

- hp : Int
  Instancié à hp = Règles.PRETREHP.
- mp : Int
  Instancié à mp = Règles.PRETREMP.
- type : TypePerso
  Instancié à type = TypePerso.Pretre.

PaladinBuilder

Description

GuerrierBuilder est une extension de la classe PersoBuilder qui va lire les attributs PaladinHP et PaladinMP de la classe Règles pour ensuite batîr un Perso avec les attributs HP et MP correspondants, ainsi que le typePerso == "Paladin"

Attributs

- hp : Int
  Instancié à hp = Règles.PALADINHP.
- mp : Int
  Instancié à mp = Règles.PALADINMP.
- type : TypePerso
  Instancié à type = TypePerso.Paladin.

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.
+ 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.

Attributs

— 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>
  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.
+ defausserCartes(List<int> defausse) : Result
  DefausserCartes retire les cartes dites en argument et les met dans le cimetière.
+ piocher() : Result
  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.)
+ recoitAttaque(int degats) : Result
  RecoitAttaque applique les dégats reçu au joueur.
+ attaque(int attaqueur, Carte attaque) : Result
  Attaque joue le coup d'attaque demandé et retourne le résultat.
+ attaque(int attaqueur, Joueur attaque) : Result
  Attaque joue le coup d'attaque demandé et retourne le résultat.
+ ajouterEnchants(List<Pair<int,int» enchs, Joueur opposant) :List<Result>
  AjouterEnchants applique les enchantements choisis par le joueur au cartes décrites et retourne les résultats. Si une des cartes données n'est pas trouvable dans els cartes du joueur, la fonction va aller la chercher du coté du joueur opposé.
+ placerPerso(int personnage, int arm) : Result
  Déploie une carte personnage et arme de la main du joueur .
+ declarerForfait() : Result
  DéclarerForfait place toutes les cartes du joueur au cimetière pour marquer qu'il a perdu et retourne un Result ditant qu'il a perdu.
+ soignerPerso (int soigneur, int soignee) : Result
  SoignerPErso effectue l'action de soigner un personnage si possible et retourne un Result décrivant l'acte.
+ detientCarte(int idCarte) : Bool
  DetientCarte vérifie si l'identifiant donné est l'une des cartes du joueur, soit dans sa main, soit sur le jeu.
+ toJSON : JSON
  Instancie une représentation JSON du deck du joueur afin de pouvoir la transférer à l'interface de l'utilisateur.

Choix de conceptions

Chacune des fonctions appliquant un coup commençe premièrement par une vérification que le joueur ait le droit de l'appliquer sur les cartes sélectionnés. S'il n'a pas le droit, la fonction retourne un RefusedResult.

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 TypePerso

Description

TypePerso est un Enum représentant les métiers/types de personnages disponibles dans le jeu.

Valeurs

— Guerrier
— Prêtre
— Paladin

Choix de conceptions

L'utilisation d'un enum pour nommer chacun des types de personnages nous permet d'avoir un accès rapide au nom qui peut être utilisé dans différent cas, tel que l'initialisation des personnages, la création des armes et l'attribution des armes au personnages. Puisque nous avons plusieurs emplacement qui ont besoin de cette information, il est plus efficace de la mettre disponible dans un Enum que de la lier explicitement à la classe Perso elle-même.

3.3 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 desoign améliore l'évolutivité du modèle. Si à n'importe quel moment, les règles du jeu doivent être modifié afin de le rendre plus intéressant au joueur, on a seulement besoin d'ouvrir cette classe et d'effectuer les modifications désirés. Ceci permet au futurs développeurs de pouvoir modifier le jeu et ses règles sans devoir perdre du temps à chercher l'emplacement des attributs modifiables dans toutes les classes.

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 Result 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

- joueurA : Joueur
- joueurB : Joueur
- joueurTour : int
  Int représentant quel joueur est en train de jouer. Si joueurTout == 0, c'est le tour du joueurA, Sinon si joueurTout == 1, c'est le tour de joueurB.
- partieEnMarche : Bool

Fonctions

+ aQuiLeTour() : int
  Getter pour joueurTout. Permet au controlleurs de déterminer quel joueur peut jouer un coup actuellement.
+ demarrerPartie() : Void
  Si une partie n'Est pas deja en cours (déterminé par partieEnMarche), cette fonction démarre la partie, initialise les cartes des deux joueurs et permet au joueurA de placer un coup.
+ declarerForfait(int idJoueur) : Result
  Applique le coup de déclarer forfait sur le joueur donné. Pour fins de sécurité, la fonction marche seulement si idJoueur == joueurTour.
+ getEtatJeu() : JSON
  Retourne une représentation JSON du jeu au complet. Ceci inclut les mains des joueurs, les cartes déployés, le nombre de cartes dans le deck, les informations importantes sur chacune des cartes et les autres données importantes pour l'affichage du jeu.
+ partieFini() : Bool
  PartieFini retourne si un des deux joueurs a perdu la partie en déclarant forfait (manuellement ou automatiquement).
+ getJoueurGagnant() : int
  GetJoueurGagnant retourne, si la partie est terminé, l'identifiant du joueur qui l'a gagné.
+ piocherCartes(int idJoueur) : List<Result>
  Si idJoueur == joueurTour, PiocherCartes appelle la fonction PiocherCartes du joueur demandé et retourne les résultats de l'acte. Cette liste est normalement constitué de 0 à plusieurs piocheResult, décrivant les cartes pigés et les effets graphiques à afficher.
+ attaquePerso(int idJoueur, int idAttaqueur,int idReceveur) : Result
  Si idJoueur == joueurTour, AttaquePerso appelle la fonction Attaque du joueur demandé avec la carte à attaquer ainsi de la carte de l'attaqueur et retourne le résultat de l'acte. Ce résultat sera soit un AttaqueResult,soit un RefusedResult, selon si l'attaque a pu être éffectué.
+ attaqueJoueur(int idJoueur,int idAttaqueur) : Result
  Si idJoueur == joueurTour, AttaquePerso appelle la fonction Attaque du joueur demandé et retourne le résultat de l'attaque. Ce résultat sera soit un AttaqueResult,soit un RefusedResult, selon si l'attaque a pu être éffectué.
+ ajouterEnchantements(int idJoueur,int carteTouche, List<int> Enchant) : List<Result>
  Si idJoueur == joueurTour, AttaquePerso appelle la fonction AjouterEnchantements du joueur demandé et retourne le résultat des enchantements. Ce résultat sera une liste constitué soit de EnchantResult, soit de RefusedResult, dépendamant si les enchantements ont fonctionné.
+ placerPerso(int idJoueur,int personnage, int arm) : Result
  Si idJoueur == joueurTour, PlacerPerso appelle la fonction PlacerPerso du joueur demandé et retourne le résultat du déploiement.
+ defausserCartes(int idJoueur, List<int> defausse) : Result
  Si idJoueur == joueurTour, defausserCartes appelle la fonction defausserCartes du joueur demandé et retourne le résultat du défaussage.
+ soignerPerso (int idJoueur,int soigneur, int soignee) : Result
  Si idJoueur == joueurTour, SoignerPerso appelle la fonction SoignerPerso du joueur demandé et retourne le résultat du soin.
+ finPartie() : Void
  FinPartie remet l'état du jeu à sa forme initial après la fin d'une partie.

Choix de conceptions

Grâce à un API simple à l'utilisation qui retourne des résultats très expressifs, Jeux nous permet de limiter aux controlleurs l'accès à seulement les coups qu'il puisse effectuer lors d'une partie et l'extraction en forme JSON du "GameBoard", qui peut alors être utilisé pour des fins d'affichages.

4.2 Result – *Interface*

Description

De nature simple, l'interface Result et ses implémentations permettent au système de communiquer expressivement et clairement le resultats des coups joués par les utilisateurs. Grâce à son interface simple, Result 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...

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 AttackResult

Description

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

Attributs

+ dommageRecu : int
  Le points de dégats attribué par l'attaque
+ attaqueur : int
  L'identifiant unique de la carte qui a attaqué.
+ attaqueJoueur : bool
  Bool déterminant si l'attaque a été faite sur le joueur ou un perso.
+ idCarte : int
  L'identifiant unique de la carte attaqué. Si l'attaque était porté sur un joueur, l'id == -1.
+ attaqueATue : bool
  Bool déterminant si l'attaque a tué la cible.

Fonctions

+ getDmgEffectue() : int
+ attaqueTue() : bool

4.2.2 EnchantResult

Description

Implémentation de Result pour les enchantements effectué dans une partie.

Attributs

- Enchantement : int
  L'identifiant unique de la carte enchantement.
- CarteEnchant : int
  L'identifiant unique de la carte qui a été enchanté.

Fonctions

+ getCarteEnchantement() : int
+ getCarteEnchantee() : bool

4.2.3 PersoResult

Description

Implémentation de Result pour les déploiements.

Attributs

- persoID : int
  L'identifiant unique du personnage déployé.
- armeID : int
  L'identifiant unique de l'arme déployé.

Fonctions

+ getPersoDeploie() : int
+ getArmeDeploie() : bool

4.2.4 RefusedResult

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.

4.2.5 SoinResult

Description

Implémentation de Result pour les actions de soin.

Attributs

- persoID : int
  L'identifiant unique de la carte perso qui a utilisé un sort de soin.
- persoSoigneeId : int
  Identifiant unique du perso qui a été soigné.

Fonctions

+ getSoigneur() : int
+ getSoignee() : bool

4.2.6 PiocheResult

Description

Implémentation de Result pour les actions de pioche.

Attributs

- carteID : int
  Identifiant unique de la carte pioché.
- carteJSON : JSON
  Description en JSON de la carte afin de l'afficher convenablement .

Fonctions

+ getCarteID() : int
+ getCarteDescription() : JSON

4.2.7 DefausseResult

Description

Implémentation de Result pour les actions de défausse.

Attributs

+ carteID : int

  Identifiant unique de la carte défaussé.

Fonctions

+ getCarteID() : int

4.2.8 FinPartieResult

Description

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

Attributs

- idJoueurGagne : int

  Identifiant unique du joueur qui a gagné la partie.

Fonctions

+ getIdJoueurGagne() : int

5. Conclusion

Les principes demandés semblent être tous respecté, qui plus est l'évolutivité est facilitée notammement grâce à la classe Règle qui permet ainsi très rapidement de modifier les valeurs essentielles au jeu. 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. Diagramme

Voir la prochaine page.