Adrien BERNARD
Naël MAVOUNGOU

MU4RBI01 : 1000 Bornes

Dans le cadre de l'ue de Programmation orientée objet en Python, nous avons dû élaborer un programme Python permettant de réaliser des parties du célèbre jeu du 1000 Bornes.

Objectif :

- Faire le diagramme UML
- Implémenter ce diagramme en Python
- Réaliser une des fonctionnalités proposées en option (dans notre cas, il s'agit d'une fonction sauvegarde)

Fonctionnement :
site des règles officielles :https://www.regledujeu.fr/regle-mille-bornes/
Notre jeu respecte les règles standard (point A dans la partie déroulement du site) de 2 à 4 joueurs. Nous n'avons cependant pas implémenté le coup fourré.

Diagramme UML

Avant de nous attaquer à la programmation du projet, nous avons effectué des recherches, celles-ci nous ont aidé à trouver le point de départ. Nous avons donc eu bien évidemment des inspirations qui nous ont servi à concevoir certaines de nos classes. Cependant nos choix n'ont cessés d'être guidés par nos réflexions personnelles. Différentes possibilités ont été envisagées, nous en avons étudié un bon nombre dans le temps qui nous était imparti. La durée totale de travail de ce projet (recherche, conception, implémentation, tests, finitions) aura été de un mois.

Classe Carte : Elle décrit et crée les cartes utilisées dans le jeu.
Remarque : On ne peut pas faire un jeu de cartes sans carte. Il nous fallait alors une classe de ce type. Cependant, nous avions plusieurs possibilités :

- Utiliser l'héritage en faisant un classe globale (classe "mère") et plusieurs sous classes héritant de celle ci (une classe carteAttaque, carteParade, carteBorne par exemple).
- Faire une seule et unique classe pouvant prendre "l'apparence" de n'importe quelle type selon les valeurs passées en paramètres du constructeur.

La première possibilité était alléchante,surtout qu'elle nous faisait utiliser un point important du cours qu'est l'héritage et par extension, la notion de polymorphisme. Mais l'une des utilités de l'héritage (pour ne pas dire la seule) est la factorisation de code en regroupant un maximum d'attribut et fonctions communes aux classes de la même lignée (ici les cartes). Nous n'avions pas trouvé assez d'éléments généralisant justifiant de faire une classe mère et plusieurs classes filles. La deuxième possibilité dont nous avions été inspiré, nous a paru plus logique. Elle utilise une variable commune à toutes les instances de classe Carte, celle ci répertorie tous les effets du jeu dans un tableau afin d'y avoir accès n'importe où dans le code même sans instancier une seule Carte. Le type et l'effet d'une carte est un entier passé en paramètre, cet entier représente la position de ces derniers dans le tableau d'effets.

Classe Pile : La Pile regroupera plusieurs cartes. Le constructeur ne prend rien en paramètre, de ce fait une Pile créée est vide par défaut. Un objet Pile utilisera par la suite

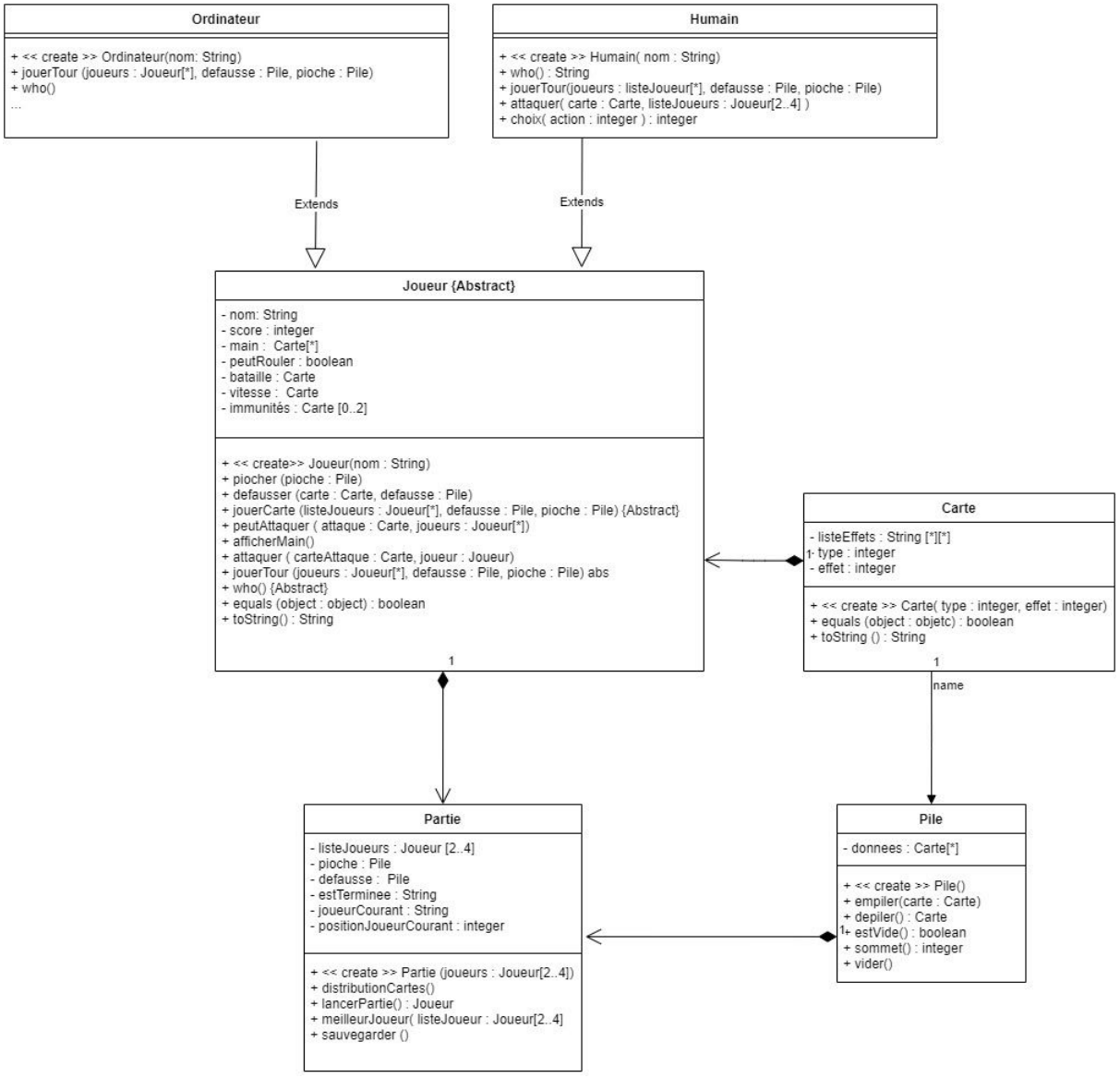
des objets Cartes déjà créés et les partagera avec d'autres classes notamment avec des instance de la classe Joueur il y aura donc une **relation d'association** entre la classe Pile et la classe Carte.

Classe Partie : La classe qui crée la partie. Elle est composée d'instances de classe Joueur et de classe Pile (de Cartes). Elle détient ces objets et ne les partage avec aucune autre classe, elle a donc une **relation de composition** avec la classe Pile et Joueur.

Classe Joueur : Définit un joueur, initialise tout ses attributs. Cette classe est abstraite et n'est donc jamais instanciée, elle sert à généraliser un "joueur".
Bien soit créé sans, un joueur possède des cartes soit dans sa main, ou dans son emplacement vitesse ou bataille, il ne les partage pas, il s'agit donc d'une **relation de composition**.

Classe Humain : Comme son l'indique, cette classe représente les joueurs qui sont des êtres humain, ce sont typiquement les joueurs qui devront interagir via la console. La classe Humain hérite de la classe Joueur. Contrairement à sa classe "mère", la classe Humain peut être instanciée et c'est tout l'intérêt car on créera forcément soit un joueur humain ou une IA (représentée par la classe Ordinateur).

Classe Ordinateur : La classe Ordinateur désigne un joueur qui sera entièrement géré par le programme (une IA). Cependant nous n'avons pas implémenté cette fonctionnalité.


Ordinateur
+ << create >> Ordinateur(nom: String)
+ jouerTour (joueurs : Joueur[*], defausse : Pile, pioche : Pile)
+ who()
...
Humain
+ << create >> Humain(nom : String)
+ who() : String
+ jouerTour(joueurs : listeJoueur[*], defausse : Pile, pioche : Pile)
+ attaquer(carte : Carte, listeJoueurs : Joueur[2..4])
+ choix(action : integer) : integer
Extends
Extends
Joueur {Abstract}
- nom: String
- score : integer
- main : Carte[*]
- peutRouler : boolean
- bataille : Carte
- vitesse : Carte
- immunités : Carte [0..2]
+ << create>> Joueur(nom : String)
+ piocher (pioche : Pile)
+ defausser (carte : Carte, defausse : Pile)
+ jouerCarte (listeJoueurs : Joueur[*], defausse : Pile, pioche : Pile) {Abstract}
+ peutAttaquer (attaque : Carte, joueurs : Joueur[*])
+ afficherMain()
+ attaquer (carteAttaque : Carte, joueur : Joueur)
+ jouerTour (joueurs : Joueur[*], defausse : Pile, pioche : Pile) abs
+ who() {Abstract}
+ equals (object : object) : boolean
+ toString() : String
Carte
- listeEffets : String [*][*]
- type : integer
- effet : integer
+ << create >> Carte(type : integer, effet : integer)
+ equals (object : objetc) : boolean
+ toString () : String
1
1
name
Partie
- listeJoueurs : Joueur [2..4]
- pioche : Pile
- defausse : Pile
- estTerminee : String
- joueurCourant : String
- positionJoueurCourant : integer
+ << create >> Partie (joueurs : Joueur[2..4])
+ distributionCartes()
+ lancerPartie() : Joueur
+ meilleurJoueur(listeJoueur : Joueur[2..4]
+ sauvegarder ()
Pile
- donnees : Carte[*]
+ << create >> Pile()
+ empiler(carte : Carte)
+ depiler() : Carte
+ estVide() : boolean
+ sommet() : integer
+ vider()