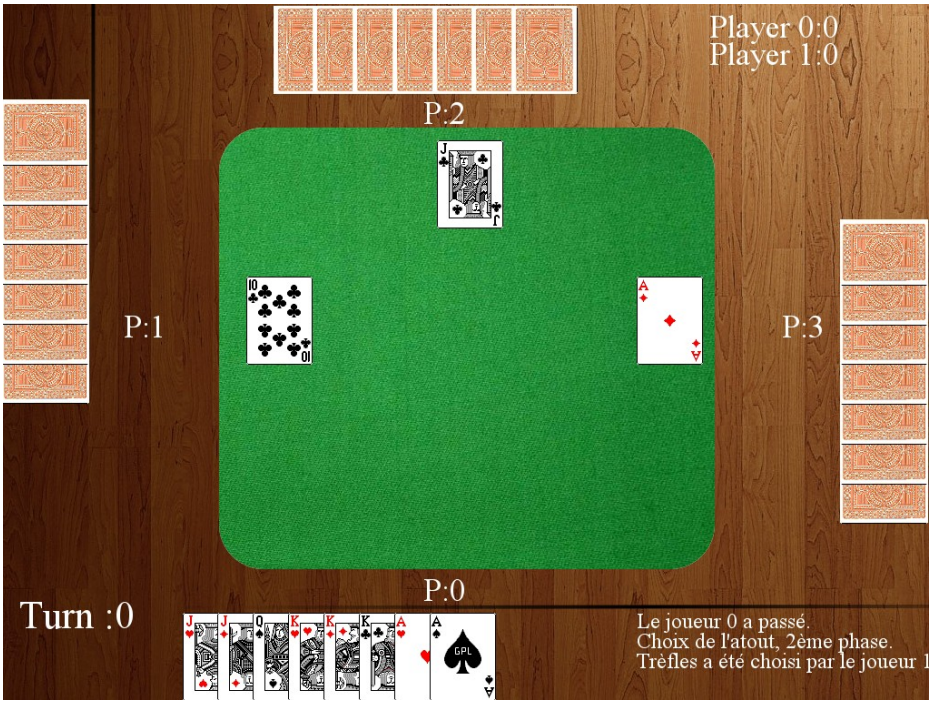
Le 24 mai 2015

Rapport PSTL

PyCE – Python CardEngine[1]

Un moteur de jeux de cartes modulaire.



Giuseppe Antonio Federico - *giuseppe.federico@etu.upmc.fr*

Aurélien Deharbe et Matthieu Dien (*Encadrants*)

1 Disponible sur : *https://github.com/digitex3d/pyCE*

Table des matières

1 Introduction

Julien est un informaticien qui adore les jeux de cartes traditionnels.

Pour satisfaire son besoin de jouer aux cartes, il a donc installé plusieurs jeux de ce type sur son pc : la belote, la dame de pique, le poker, le poker chinois etc …

Julien aime tellement les jeux de cartes qu'il décide un jour de concevoir son propre jeu de cartes avec ses propres règles ; il est aussi très paresseux et il n'a pas envie de recoder un client en entier.

Il va alors rapidement consulter internet pour rechercher un moteur de jeu de cartes qui puisse l'aider à concevoir rapidement son jeu à partir de ses règles.

Malheureusement, suite à sa recherche, il se rend compte que ce principe n' existe que pour la catégorie spécifique des jeux de cartes à collectionner et non pas pour les cartes traditionnelles.[2]

De plus, ces moteurs de jeux ne vérifient pas les règles durant les parties mais laissent le contrôle complet aux joueurs. L'objectif de Julien est également de pouvoir d'arbitrer les parties.

Ce projet propose une solution au problème de Julien: il s'agit de concevoir un logiciel de jeu de cartes traditionnel qui peut être étendu à l'aide de plugins.

Les plugins vont définir les règles du jeu et le comportement d'un adversaire, le moteur sera aussi capable d'arbitrer la partie à l'aide de ces règles.

Pour pouvoir concevoir un plugin adapté à notre moteur, une connaissance basique du langage *Python* est suffisante.

Par la suite, le lecteur sera introduit aux notion basiques de moteur de jeu et de plugin.

Successivement, seront présentés : les objectifs, les enjeux et difficultés de la réalisation, l'architecture du logiciel et pour terminer un bref tutoriel pour écrire un simple plugin d'un jeu de « Bataille »[3].

2 Quelque exemple de moteur pour les cartes à collectionner déjà existant: http://www.ccgmaker.com/; http://www.lackeyccg.com

3 Le principe sera illustré dans la partie dédié au tutoriel.

1.1 Qu'est-ce qu'un moteur de jeu?

Si vous suivez l'industrie des jeux vidéo , vous-allez beaucoup entendre parler de « moteur de jeu ».

On peut imaginer un moteur de jeu comme le moteur d'une voiture: c'est tout simplement l'outil principal qui fait que la voiture avance.
Il s'agit donc du composant le plus important de l'application future (même si c'est en fait une entité autonome).

Prenons en considération un jeu simple tel que tic-tac-toe. Tant que personne ne gagne/perd une partie, on se met en attente d'une action de l'un de deux joueurs. Une fois que cette action vient enregistré par l'interface graphique puis envoyé au moteur de jeu, ce dernier calcule un nouveau état de jeu (X ou O est ajouté).

En résumé, c'est tout ce qui est indispensable à la gestion d'un jeu vidéo, sans être dépendant du contexte du jeu même.

Le moteur de jeu regroupe donc tout ce qui concerne le fonctionnement globale du jeu (fonctions principales de base), comme par exemple:

- La gestion des entrées / sorties (clavier, souris, haut parleur, etc.)
- La physique
- L'intelligence artificielle

Ainsi , tout ce qui se rapporte à la gestion du scenario du jeu ne fera pas partie du moteur , puisqu'il s'agit d'une partie totalement en rapport avec le contexte du jeu.

1.1.1 Les avantages

Les avantages d'écrire un moteur de jeu sont différents :

- Réutilisation du code
- Modularité
- Commercialisation
- Maintenance du code

Souvent, le même moteur de jeu est réutilise pour des jeux de même nature.

De plus, étant un composant autonome, il peut donc être développé et maintenu indépendamment du jeu et être vendu à part.

1.2 Qu'est-ce qu'un plugin?

On parle fréquemment de plug-in dans le domaine des jeux vidéo, browsers, éditeurs de texte etc...

La traduction du mot anglais plugin - «Capable d'être branché» - définit suffisamment bien ce qu'un plugin représente en informatique .

En fait, un plugin est un petit bout de code qui va venir s'ajouter à un programme existant, en étendant ses fonctionnalités.

Ce qui est intéressant quand on parle de plugins n'est pas seulement le fait de pouvoir étendre les fonctionnalités d'un logiciel, mais notamment le fait que l'on puisse le faire pendant son exécution ; donc, pas la peine de recompiler.

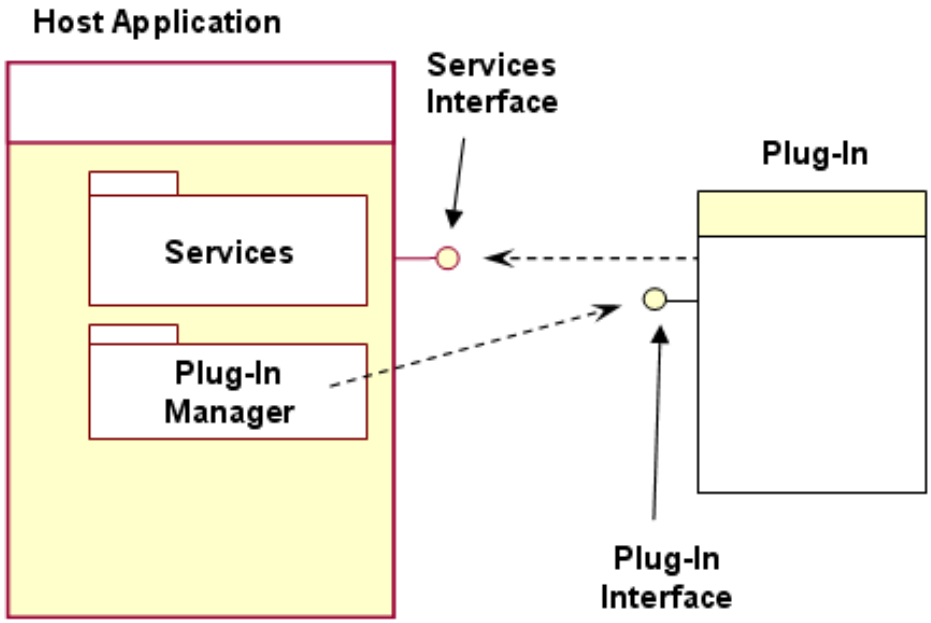


Diagramme 1: Contexte d'un plugin

Ci-dessus on peut visualiser la manière dont un plugin est branché à l'application principale.

Souvent, et c'est le cas dans ce projet, les plugins peuvent être produit par les utilisateur mêmes, même s' il ne connaissent pas l'architecture du logiciel.

Le but du développeur étant dans ce cas celui de fournir des fonctions simples et intuitives pour pouvoir permettre au développeur du plugin, d'interagir avec le moteur de jeu de manière transparente (avec une couche d'abstraction par exemple).

1.3 Développement du PSTL

Le défi fondamental sera donc celui de concevoir un moteur de jeu de cartes qui peut être étendu à l'aide de plugins créés par l'utilisateur.

Ceci sera découpé en différents phase de développement:

- La conception d'un moteur générique de Jeu de cartes
- La conception d'un client graphique pour interagir avec ce moteur
- La conception d'une couche d'abstraction capable, en même temps, d'isoler et de servir d'intermédiaire entre le plugin et le moteur de jeu.
- Des plugins décrivant les règles d'un jeu de cartes et le comportement des adversaires

Les difficultés principales seront les suivantes :

- Réussir à isoler le moteur de jeu et le plugin avec une couche d'abstraction
- Fournir des fonction et une structure simple et intuitive au développeur du plugin.

Il faut notamment s'assurer que l'on fournisse des fonctions qui recouvrent un nombre optimal de règles et caractéristiques des jeux de carte traditionnels.

On va permettre ainsi au développeur d'implémenter sans contraintes une grande variété de jeux de cartes.

2 Organisation du logiciel

2.1 Introduction

Dans cette partie je vais illustrer, à l'aide des diagrammes UML[4] , l'architecture du projet en détaillant les composants clés.

Le langage utilisé pour l'implémentation est Python 3.

Ce dernier offre des fonctionnalités intéressantes pour le développement de la partie plugin, la possibilité de « décorer » une fonction par exemple.[5]

Ci-dessous une vue d'ensemble des composants du projet .

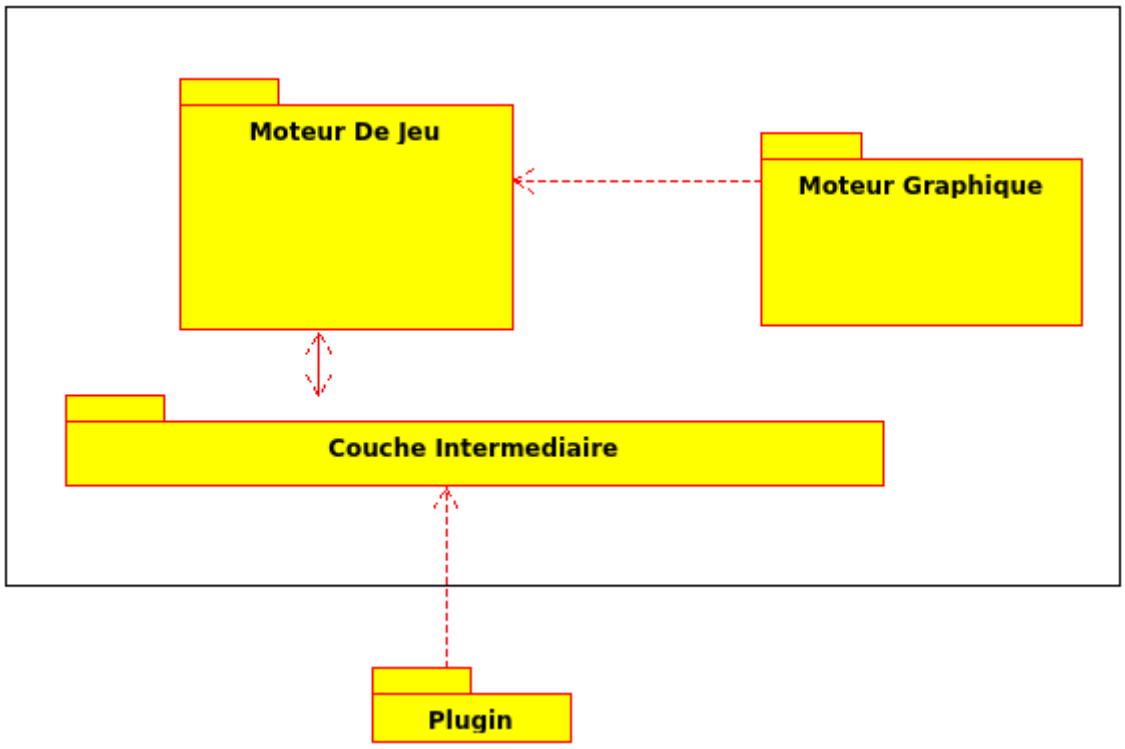


Diagramme 2: Vue d'ensemble du projet

4 Undefined Modeling Language
5 Le decorateur de Python 3 - https://www.python.org/dev/peps/pep-0318/

On remarque donc que le code du projet est organisé en 4 parties principales :

- Le moteur de jeu
- L'interface Graphique (Pyglet)
- La couche intermédiaire (Moteur / Plugin)
- Un plugin

2.2 La structure du moteur de jeu

2.2.1 Introduction

Comme il à été précédemment indiqué, le but du moteur de jeu est de gérer les événements qui proviennent de l'interface graphique et de calculer un nouvel état de jeu en fonction de ces événements.

Dans le projet, le moteur de jeu est composé par une liste de joueurs, des observateurs (l'interface graphique) , un état de jeu et une pile d'événements.

Voici ci-dessous les classes qui font partie du moteur de jeu :

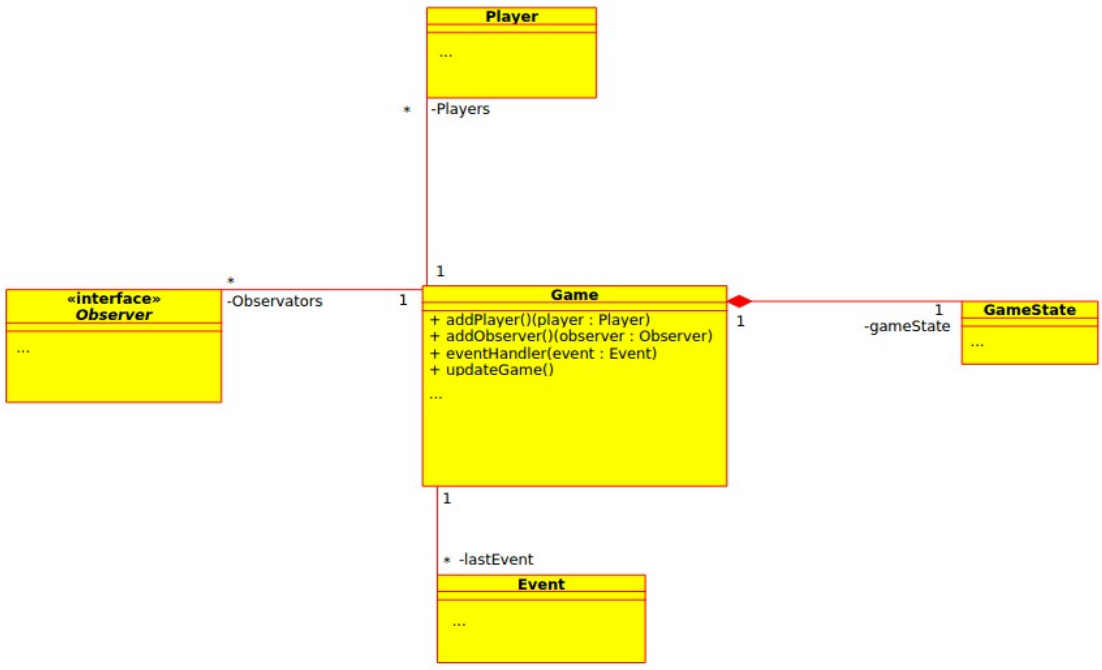


Diagramme 3: La structure du moteur de jeu

2.2.2 La classe Game

La classe game est le cœur du moteur. On y retrouve notamment la méthode updateGame qui met à jour l'état de jeu en fonction d'une queue événements générés par l'interface graphique.

Un *événement* contient tous les informations dérivant d'un clic (coordonnées, objet cliqué etc...).

Plus précisément, la fonction *updateGame* appelle la fonction *nextSate* du *gameState* et met à jour le jeu.

La fonction `updateGamestate` est appelée

dans la boucle principale de la fenêtre, voici son pseudo-code:

Pseudocode

```
def updateGamestate() :
     # Le tour courant
      turn = id_joueur_humain:
     if( lastEvent est vide):
          action = none
     else :
         action = l'action du joueur courant défini dans le plugin
          # Effectue l'action et met à jour l'état du jeu
          game_state = nextState(action)
          # On met à jour tous les observateurs du jeu
          updateObservers()
```

2.2.3 La classe Joueur

Un joueur est une classe munie d'une méthode *getAction* qui étant donné un état de jeu nous renvoie une action qui va se traduire ensuite en un changement de l'état du jeu.

Le comportement d'un adversaire est défini dans le plugin, une fonction *getPluginAction*() est définie dans le *pluginManager* pour définir le comportement d'un opposant.

La classe GameState

Le *gameState* est le composant qui contient les informations sur l'état de jeu courant.

Ce composant permet d'ailleurs d'accéder facilement à la table où se trouvent tous les objets qui font partie du contexte de jeu.

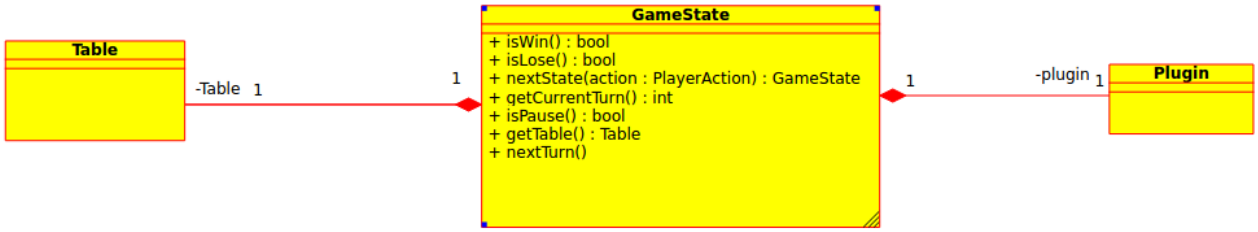


Diagramme 4: La classe GameState

On y retrouve notamment la fonction *nextState*, qui calcule le prochain état de jeu à partir de l'action d'un joueur.

Cette fonction est particulièrement intéressante car c'est grâce à elle que l'on peut accéder aux fonctions du plugin.

Ci-dessous le pseudocode de la fonction *nextState*, on peut remarquer que le *nextState* est calculé à l'intérieur du plugin.

Pseudocode

```
def nextState(action) :
     if( action legale ):
          return plugin.nextState( action )
     else:
          print('ILLEGAL MOVE !' )
     return self
```

2.2.4 La classe table

La classe table contient un « terrain de jeu », ou plus précisément, les états des joueurs (représentés avec la classe *PlayerState)* , les mains, les cartes etc ...

Un *PlayerState* contient les informations d'un joueur (l'id, le score, sa main etc...).

C'est grâce cette classe que l'on accède aux informations du joueur pendant le déroulement du jeu.

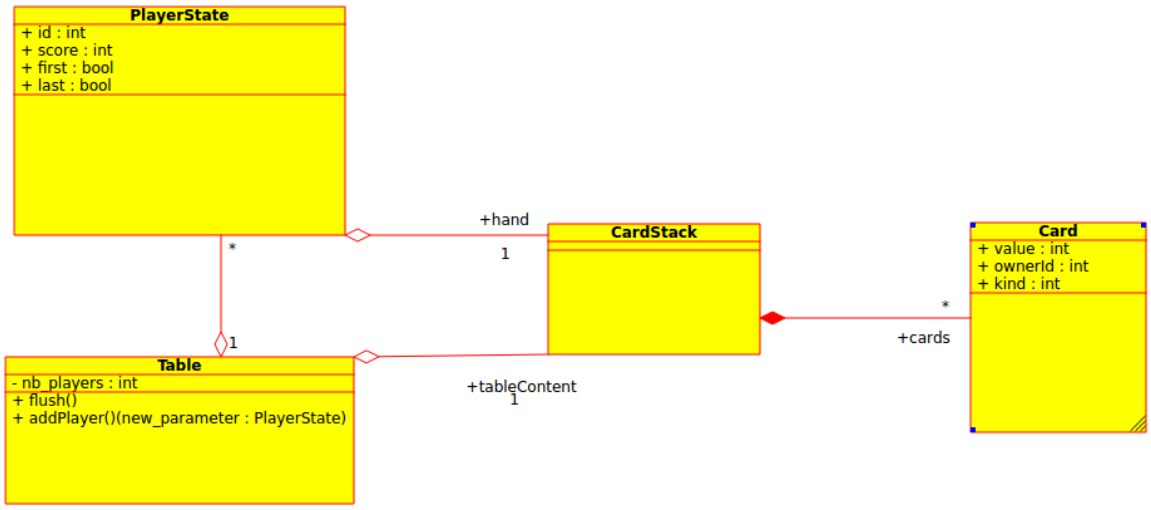


Diagramme 5: Les classes Table,PlayerState,CardStack et Card

2.3 L'interface graphique

2.3.1 Introduction

Une interface graphique (anglais GUI pour graphical user interface) est un dispositif de dialogue homme-machine dans lequel les objets à manipuler sont dessinés à l'écran.

Le monde Open Source offre plusieurs bibliothèques graphiques pour pouvoir implémenter une interface graphique en *Python*.

J'ai choisi d'utiliser un bibliothèque spécifique pour le développement des jeux 2D.

Au début j'étais attiré par *PyGame*[6], une des bibliothèques plus utilisée pour développer des jeux 2D en *Python*, mais malheureusement elle ne supportait pas *Python 3* au moment de l'écriture du logiciel.

J'ai finalement décidé d'exploiter la bibliothèque ***PyGlet***[7], pas très connue mais très intéressante pour les raisons que je vais détailler par la suite.

6 Site officiel PyGame- http://www.pygame.org/
7 Site officiel PyGlet - http://www.pyglet.org/

2.3.2 La bibliothèque Pyglet

J'ai décidé d'utiliser *Pyglet* pour les raisons suivantes :

- Simplicité
- Compatibilité avec Python 3
- Très bonne documentation.

En effet, le critère prépondérant est sa simplicité ; de plus la bibliothèque dispose d'un excellent tutoriel qui fourni de nombreux exemples.

Un autre aspect intéressant de *Pyglet* est sa puissance ; le fait qu'il soit basé sur *OpenGL* donne la possibilité d'utiliser tous ses fonctions.

La qualité et la simplicité de Pyglet m'ont permis de concevoir rapidement une interface graphique pour mon projet sans devoir forcer aucune structure particulière.

Je vais présenter les fonctionnalités de cette bibliothèque en parallèle avec la présentation de l'interface graphique.

2.3.3 Les conteneurs

Pour l'interface graphique j'ai défini trois conteneurs graphiques :

- Les Composants
- Les Drawables
- Les Sprites

Concrètement ce sont des classes abstraites qui implémentent une interface *ConteneurGraphique* qui contient les fonctions *isClicked(), draw(), update()* et les fonctions pour obtenir le contenu ou en ajouter.

Ci-dessus un diagramme pour mieux visualiser cette structure.

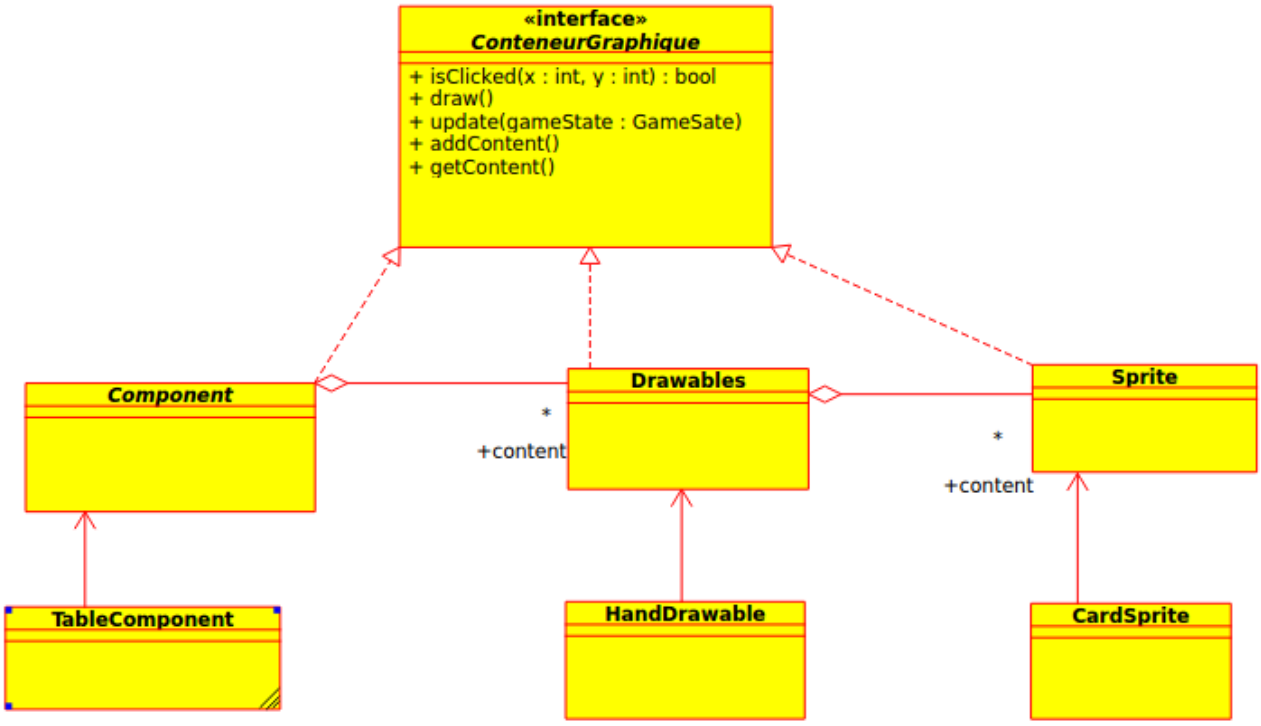


Diagramme 6: Les conteneurs graphiques

Tout conteneur graphique fait partie de la fenêtre principale, les conteneurs à l'intérieur de la fenêtre sont distribués ainsi :

Dans une fenêtre on peut dessiner (avec la fonction draw de Pyglet) un ensemble **Component**.

Un **Component** est un ensemble de **Drawable** et un *Drawable* est un ensemble de **Sprite**.

Un **Sprite** est défini par sa dimension et contient un Sprite Pyglet (qui peut être dessiné dans la fenêtre avec la fonction draw).

Par exemple, la table est un *Component* qui contient les main des joueurs, les cartes etc …

Le *Component* HUD[8] au contraire, contient seulement des labels, pour afficher le score par exemple.

Les mains des joueurs sont des *Drawable* qui contiennent des cartes (des sprites) et ainsi de suite.

De cette manière, quand on va produire un clic, le moteur de jeu reçois un objet de type *Event* qui contient une référence sur le composant cliqué, le *Drawable* cliqué et le *Sprite* cliqué.

De plus, pour pouvoir afficher tous les composants graphiques il suffit d'appeler la fonction *draw* sur l'ensemble des Composant.

Voici une capture d'écran qui illustre cette distribution.

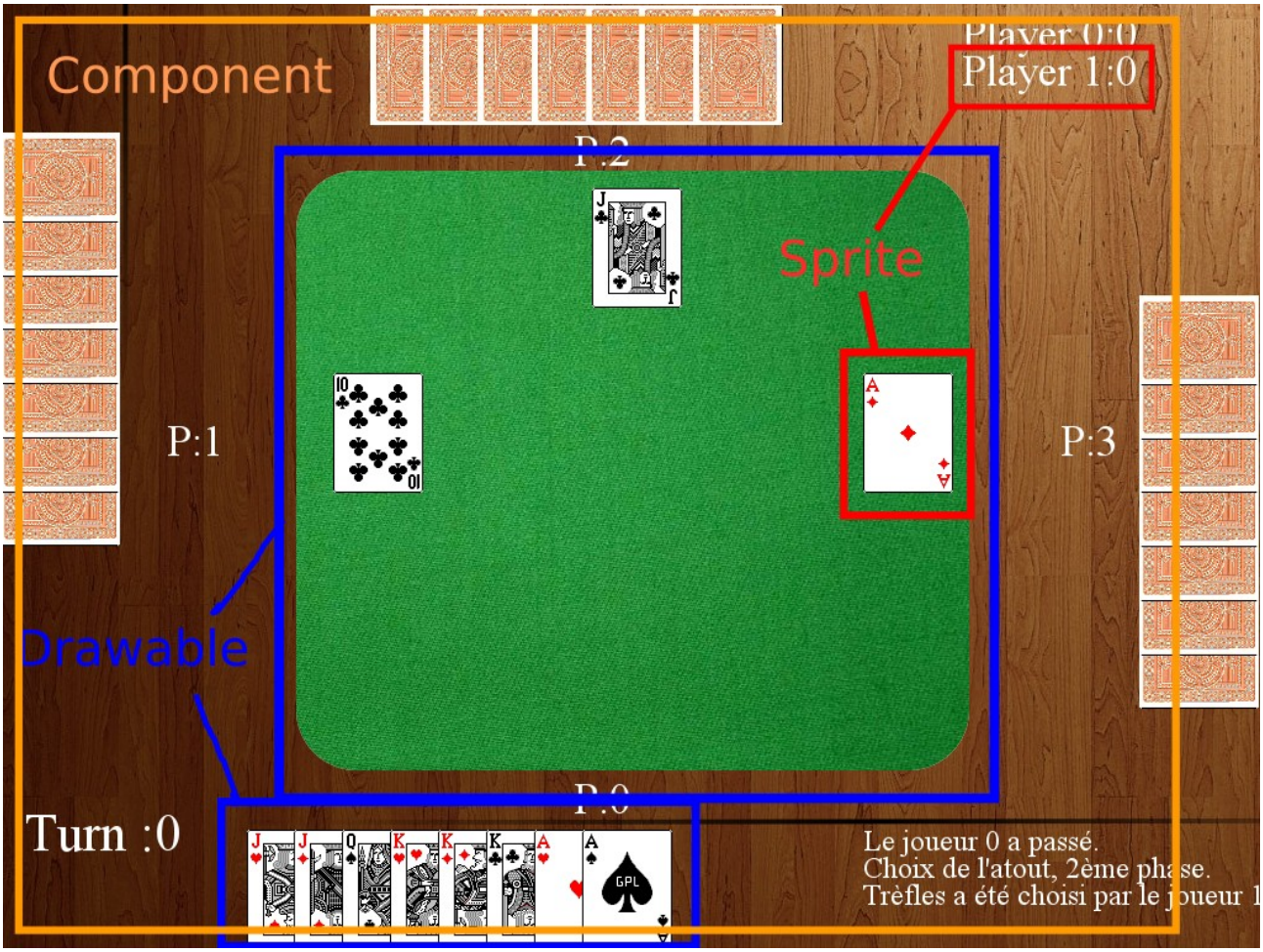


Diagramme 7: Les conteneurs

8 HUD : heads-up display

2.3.4 La fenêtre de jeu

La fenêtre principale du jeu se charge de l' interception et de la transmission des événements au moteur de jeu.

C'est ce composant qui contient la boucle principale qui met à jour le jeu au fur et à mesure que le temps découle.

Voici le pseudo-code de la boucle principale:

Pseudocode

```
def main_loop(self):
  """ La boucle principale du jeu
  """

  while not (self.has_exit):
    updateGame()
    draw()
```

On remarque que c'est à l'intérieur de cette boucle qu'on appelle la fonction updateGame() qui met à jour le jeu.

2.3.5 La Capture des événements

Pyglet offre un gestionnaire d'événements très simple et efficace .

Dans la fenêtre principale les entrées externes sont gérés par des fonctions *handlers* qui son exécutées à la suite d' un événement, le clic de la souris par exemple.

De cette manière, gérer les événements devient trivial, il suffit de remplir la queue des événements du moteur de jeu puis de les traiter au cours de l'exécution.

Voici le code de la fonction on_*mouse_press*, déclenché par l' « *Event Dispatcher* » de *Pyglet* :

Code

```
def on_mouse_press(self, x, y, dx, dy):
  # Last event est un objet de type événement qui sera passé au
  # moteur de jeu parmi la fonction updateGame
 last_event = Event("mouse_click")
 last_event.add_mouse_coords(x,y)
 for component in components:
     if component.isClicked(x,y):
        last_event.componentClicked = component
      for drawable in component.getDrawables():
         if drawable.isClicked(x,y):
            self.last_event.drawableClicked = drawable
          for sprite in drawable.getSprites():
             if sprite.isClicked(x,y)):
                self.last_event.spriteClicked = sprite
  info('Mouse press: X:%s Y:%s',x ,y )
```

Appuyer sur un bouton de la souris génère un objet *Event* qui contient :

- Les coordonnées du clic
- Le composant cliqué
- Le drawable cliqué
- Le sprite cliqué

L'Event est ensuite placé dans la queue des événements du moteur de jeu.

2.4 La structure du plugin

2.4.1 Introduction

Le plugin définit les règles d'un jeu de cartes et le comportement d'un adversaire.

Le but c'était celui de rendre l'écriture d'un plugin simple, intuitive et transparente.

2.4.2 Importer le plugin

Un plugin est importé au runtime par la classe main du programme et passé en paramètre au moteur de jeu.

La fonction *_import_()* de Python offre la possibilité d'importer un module ; le but ici est d'importer le module du plugin pendant l'exécution du programme, cette fonction est donc très utile.

Voici le code de son utilisation :

Code

```
# On importe le plugin passé en argument, de cette façon
# on pourra importer le plugin à tout moment au runtime
# avec la fonction __import__

def my_import(name):
  mod = __import__(name)
  components = name.split('.')
  for comp in components[1:]:
    mod = getattr(mod, comp)
  return mod

mod = my_import("plugins.%s" % plugin_name +".PluginInit")
plugin = mod.PluginInit()

# On initialise le jeu avec le plugin passé en paramètre
game_state = GameState(plugin.initGameState(), plugin)
game = Game(game_state)
```

2.4.3 La classe pluginManager

La classe pluginManager fais office de couche d'abstraction entre le moteur de jeu et le plugin.

On y retrouve toutes les fonctions utiles pour définir les règles du jeu à l'intérieur du plugin.

Voici un diagramme qui illustre son comportement :

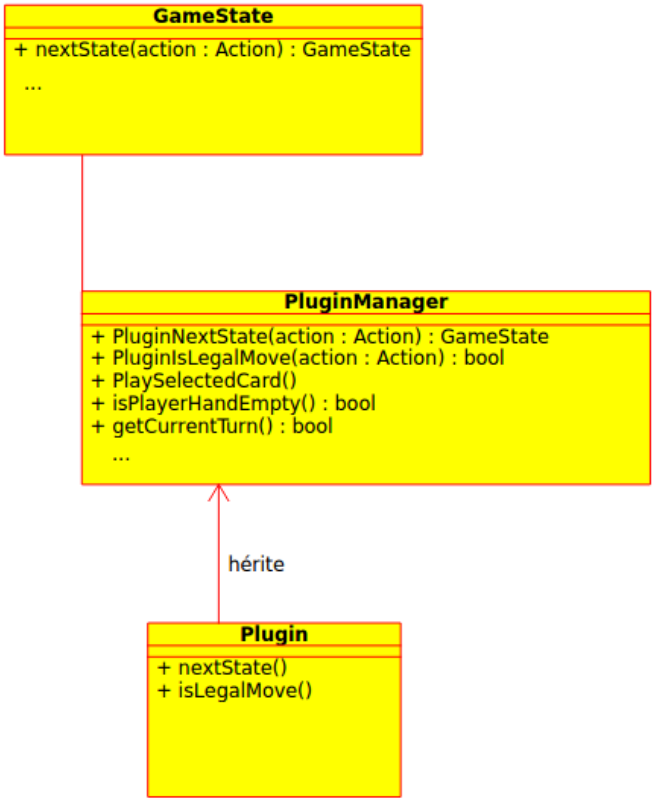


Diagramme 8: Le plugin

La fonction *nextState* du plugin est appelée à l'intérieur de la fonction *pluginNextState* tout comme la fonction *isLegalMove* qui est la condition pour pouvoir appeler *nextState*.

De plus, l'utilisateur accède au fonctions de pluginManager de manière transparente

sans pouvoir modifier directement le GameState.

Par exemple, si pendant une phase de jeu particulière l'on veut distribuer des cartes, on va procéder ainsi : dans la fonction nextState du plugin, on appelle tout simplement la fonction *dealCards* (définie dans PluginManager et disponible par héritage) et le jeu est fait ![9]

Voici le pseudo-code de la fonction PluginNextState et PluginisLegalMove :

```
def pluginNextState(action):
  # Cette fonction apelle la fonction nextState du plugin
  # isLost et isWin son redéfini dans le plugin
  if( not gagné and not perdu ):
    nextState() #  Défini dans le plugin !!
    sortAllHandsByValue() # Fonction commun à tout jeu de carte
  else:
    if(isWin()):
      print("Vous avez gangé.")
    else:
      print("Vous avez perdu.")
  return gameState

def PluginisLegalMove(action):
     return isLegalMove() # Défini dans le plugin !!
```

On peut remarquer dans ce pseudo-code que le *Decorator* de Python 3 est utilisé :

On ajoute des opérations avant et après l'appel de nextState, on a ainsi décoré la fonction du plugin.

Avec cette structure, le but du développeur devient donc celui de redéfinir les deux fonctions *nextState* et *isLegalMove* hérités du pluginManager avec les fonctions fournies par cette même classe.

Le plugin hérite donc toutes les fonctions nécessaires pour modifier et accéder a un état de jeu de manière transparente.

9 La documentation de la classe *PluginManager* peut être consulté en Annexe.

3 Tutoriel

On va a présent fournir un guide pour le développement d' un plugin qui implémente le jeu de la « bataille ».

La bataille est un jeu de cartes qui se joue habituellement à deux et qui est d'une grande simplicité puisque sous la conduite exclusive du hasard.

On distribue l'ensemble d'un jeu de cartes aux joueurs, qui n'en prennent pas connaissance. À chaque tour, chaque joueur retourne la carte du haut de sa main . Celui qui a la carte de la plus haute valeur gagne.

En cas d'égalité de valeurs les joueurs commencent par placer une seconde carte jusqu'à ce que les cartes jouées soient différents, la procédure est répétée sinon.

3.1 Implémentation

Un plugin doit impérativement être placé dans un répertoire nommé comme le plugin.

À l'intérieur de ce répertoire il faudra créer un module « *Plugin.py* » qui définit deux classes,

- Une classe « Pugin » pour définir les règles de jeu.
- Une classe IABataille pour définir le comportement du joueur adversaire.

Voici le début de notre fichier « Plugin.py » placé dans le répertoire bataille .

```
from game.PluginManager import PluginManager, IAPlugin
class Plugin(PluginManager):
  """ Cette classe représente un PluginBataille
  """
  def __init__(self):
    PluginManager.__init__(self, "La Bataille")
```

Il faut maintenant donner les informations pour l'initialisation du jeu, il faut connaître le nombre des joueurs et ajouter les opposants (la classe IABataille qu'on va définir à la fin)

```
def pluginInit(self):
    """ Cette fonction initialise l'état intitial du jeu
      La table et les joueurs.
    """
    self.initPlayers(2)
    self.opponents.append(IABataille())
```

3.1.1 Les phases de jeu

On suppose qu'un jeu est défini par plusieurs phases.

Pour pouvoir implémenter notre plugin, il faudra d'abord définir les phases du jeu, tout jeu de carte doit commencer par une phase « Start »

Une bataille est composé par le phases suivantes :

- Initiale
- Jeu
- Finale

Dans la phase initiale on va effectuer les action suivants

- Distribuer une carte à chaque joueur
- Déterminer le premier et dernier joueur
- Déterminer la prochaine phase de jeu, la phase de jeu dans ce cas

```
def initialPhase(self):
    # Nettoyage de la table
    self.flushTable()
    # Distribution des cartes aux joueurs
    self.dealCards(1)
    # On affecte le premier joueur
    self.setFirstPlayer(0)
    # On affecte le dernier joueur
    self.setLastPlayer(1)
    # On change la phase de jeu courante
    self.setCurrentPhase("Play")
```

Dans la phase de jeu :

- Jouer la seule carte qu'on a dans la main

- Si on est le dernier passer à la phase , passer le tour sinon

```
  def playPhase(self):
    # La fonction agentAction récupère l'action éfféctué par le
    # joueur
    if(self.agentAction.type == "move"):
     # Joueur la carte sélectionne
      self.playSelectedCard()

     # Si je suis le dernier saute directement à la phase suivante
      if(self.iAmLastPlayerToPlay()):
        self.endTurnPhase()
        return
      self.next_turn()
```

Dans la phase Finale on va déterminer le gagnant si (y en a un) et calculer les points, sinon, on distribue une carte et on retourne à la phase de jeu.

```
def endTurnPhase(self):
  # Les deux dernières  cartes sur la table
  lastCards = self.getLastCards(2)
  carte1Val = self.getValeurCarte(lastCards[0])
  carte2Val = self.getValeurCarte(lastCards[1])

  if((carte1Val != carte2Val)):
    score = self.getTableSumCardScore()
    winner = 0 if (carte1Val > carte2Val) else 1
    self.addPlayerScore(winner, score)
    self.appendLogInfoMessage("Player " + str(winner) + " win
turn. Points:" + str(score))
    self.setCurrentPhase("Start")

  else:
    self.appendLogInfoMessage("Draw")
    self.dealCards(1)
    self.setCurrentPhase("Play")
```

Chaque phase de jeu peut faire appel à la fonction *agentAction* qui renvoie l'action du joueur courant, en fonction de cette action on peut donc produire un effet sur l'état de jeu, dans ce cas, jouer une carte.

3.1.2 La fonction nextState du plugin

Dans la fonction du nextState on va tout simplement appeler la fonction

correspondante à la phase de jeu courante.

```
def nextState(self):
  """  Renvoie le prochain etat du jeu etant donnée une action
    getAction() pour récuperer une action
  """

  # On récupere la phase de jeu actuelle
  phase = self.getCurrentPhase()

  # Le jeu commence toujours par la phase Start
  if( phase == "Start"):    self.initialPhase()
  if( phase == "Play"):    self.playPhase()
  if( phase == "EndTurn"):    self.endTurnPhase()
```

3.1.3 Les fonctions isWin() et is Lost()

Dans tous les jeux de cartes, il faut impérativement redéfinir ces deux fonctions, dans le cas de la bataille, une partie est gagnée s'il n y a plus de carte et notre score est le meilleur.

Ces fonctions sont vérifiées à chaque tour.

```
def isWin(self):
    return( self.isDeckEmpty() and self.IHaveBestScore())

def isLost(self):
  return( self.isDeckEmpty() and not self.IHaveBestScore()
```

3.1.4 La fonction isLegalMove

Dans cette fonction on va vérifier si l'action passée en paramètre de la fonction nextState est une action valide, dans le jeu de la bataille on a pas vraiment beaucoup de cas d'action illégales, on pourrait juste essayer de jouer la carte de l'adversaire par example.

Voici son implémentation

```
def isLegalMove(self):
    """
    Renvoie True si l'action est légale à l'état courant
    :return:
    """
    if(self.agentAction.type == "move"):
          if(!self.isMine(self.selectedCard()):
               return false

    return True
```

3.2 L'opposant

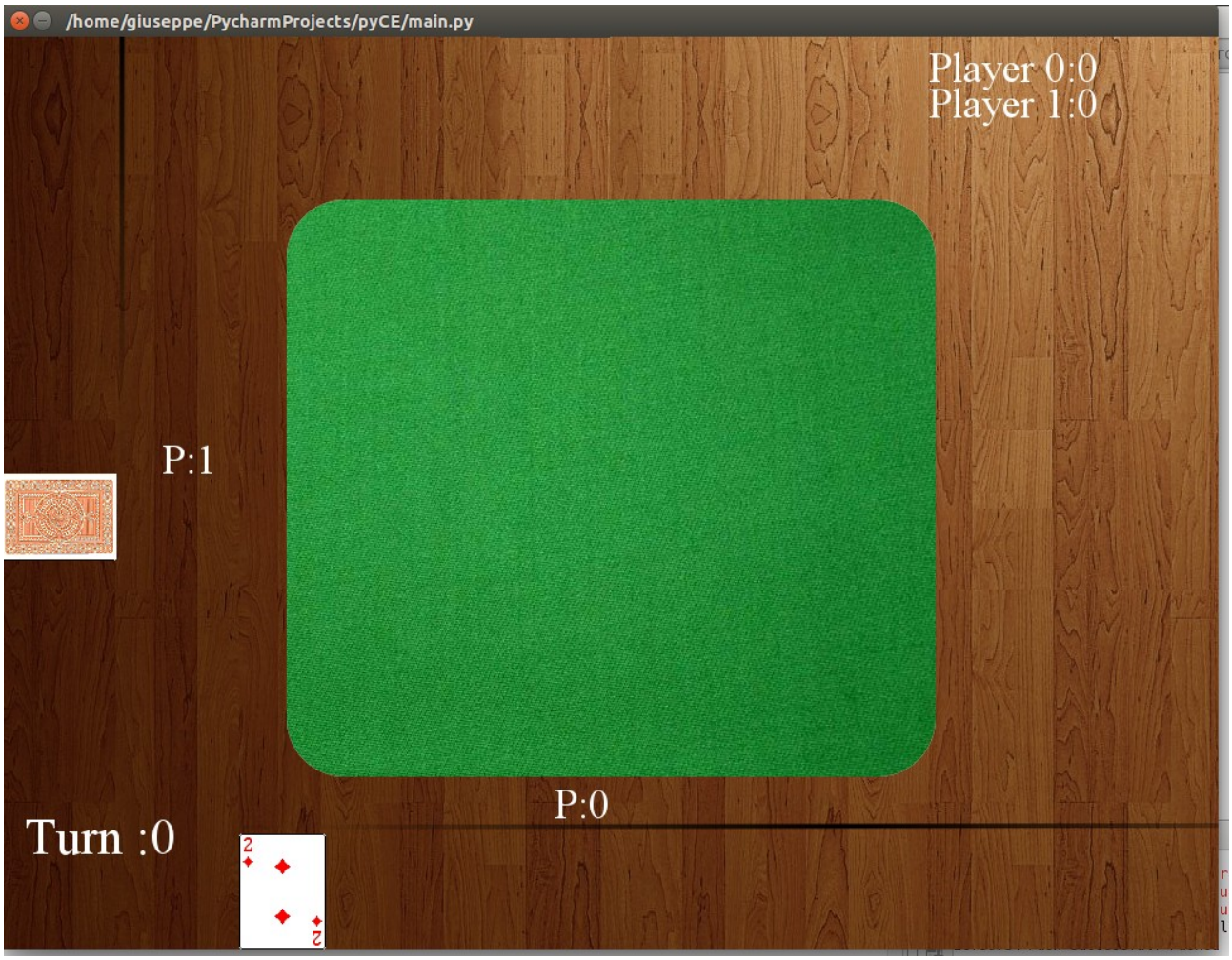
Pour définir le comportement de l'adversaire il faudra définir une classe qui contient la fonction getAction.

Dans le cas de la bataille, notre adversaire va tout simplement jouer la seule carte qu'il a en main, voici l'implémentation de cette classe.

```
class IABataille(IAPlugin):
  def __init__(self):
    IAPlugin.__init__(self, 1)
  def getAction(self):
    card = getCurrentPlayerFirstCard()
    move = defAgentAction("move", card)
    return move
```

Pour définir une action on utilise la fonction defAgentAction qui prend en paramètre le nom de l'action et un argument, dans ce cas, la carte à jouer, puis on renvoie l'action.

Ceci suffit pour définir le jeu de la bataille dont voici un screenshot.[10]



ScreenShot 1: La bataille

10 Vous pouvez consulter le code de la bataille dans le répertoire plugin/bataille du projet

4 Journal de bord

Voici les étapes du développement d'après le journal de bord.

1. Analyse du projet, choix des outils, premiers diagrammes structurels
2. Conception du squelette de l'interface graphique, premier rencontre avec Pyglet
3. Conception du squelette du moteur de jeu : boucle principale et premieres « actions »
4. Implémentation d'un jeu de bataille sans plugin
5. Ajout du PluginManager, déplacement des fonctions du GameState au PluginManager, premier plugin fonctionnant mais peu de fonctions.
6. Implémentation de la belote, ajout de fonctions pour les plugins.
7. Révision de la documentation.
8. Écriture rapport.

5 Conclusion

Mon objectif étant celui de travailler dans le domaine des jeux vidéos, ce projet m'a beaucoup aidé a comprendre le fonctionnement d'un moteur de jeu, d'une interface graphique et d'un plugin.

De plus, j'ai appris de nouvelles notions de Python 3, le *Decorator* par exemple, et j'ai acquis une bonne connaissance de Pyglet .

C' était agréable de travailler avec cette bibliothèque, simple et conceptuellement propre, surtout la gestion des événements. Je n'ai pas du forcer une structure de conception particulière pour m'adapter à Pyglet, je suis très satisfait de mon choix.

Bien qu' initialement ce projet était prévu pour deux personnes, j'ai quand-même pu avancer à la fois dans l'implémentation graphique et du moteur, j'aurais souhaité disposer d'avantage de temps pour pouvoir créer d' autres exemples de plugins et ajouter des fonctions supplémentaires, mais le temps était malheureusement limité.

Au-delà des buts pédagogiques, on a réussi dans la mission de concevoir un moteur de jeux de cartes traditionnels avec arbitrage.

On est maintenant capable de concevoir d'une manière simple et intuitive, une quantité satisfaisante de jeux de cartes sous forme de plugins écrits en *Python.*

On peut imaginer le futur de cette application employée à des fins pédagogiques en tant qu' outil pour apprendre les notions de base du langage *Python ;* ou tout simplement dans le but d'aider Julien[11] à écrire son propre jeu de cartes.

11 Le personnage fantasieux présenté en introduction.

6 Annexe

Les fonctions utiles pour définir les règles d'un jeu de cartes.

La documentation de la classe PluginManager

class **PluginManager**

```
La classe pluginManager fais office de couche d'abstraction entre
le moteur de jeu et le plugin.
On y retrouve toutes les fonctions utiles pour définir les
réglés du jeu à l'intérieur du plugin.
```

GNextState(self, gameState, agent_action)

```
Cette fonction apelle la fonction nextState du plugin
    actuel.

:param gameState (GameState): game state courant
:param agent_action ( AgentAction): Action courante
:return: Le nextState calculé par le plugin
```

GisLegalMove(self, gameState, agent_action)

```
Dans cette fonction on va vérifier si l'action
passé en paramètre est une action valide.

:param gameState: GameState
:param agent_action: AgentAction
:return: boolean
```

6.1 Les méthodes pour gérer le score

IHaveBestScore(self)

```
Renvoi true si le joueur courant a le meilleur score,
 false sinon.
:return: boolean
```

addPlayerScore(self, pid, score)

```
Ajoute le score 'score' au joueur 'pid'
:param (int) pid:
:param (int) score:
:return: None
```

getCurrentPlayerScore(self)

```
Renvoie le score du joueur
:return (int):
```

getPlayerScore(self, pid)

```
Le score du joueur de pid 'pid'
```

```
    :param (int) pid:
     :return:
```

getTableSumCardScore(self, inf=0, sup=None)

```
    Renvoie la somme des valeurs des cartes sur la table.
    On peux spécifier les indexes.

    :return (int):
```

hasBestScore(self, pid)

```
    Renvoie true si 'pid' a le meilleure score, false sinon.
    :param pid:
    :return:
```

6.2 Les méthodes pour gérer la main

getCurrentPlayerFirstCard(self)

```
    Retourne la première carte de la main du joueur courant si elle
    existe.

    :return (Card): Une Carte ou None
```

allHandsEmpty(self)

```
    Renvoie true si tous les main des adversaires sont vides,
     false sinon

    :return (boolean):
```

appendCardToHand(self, pid, card)

```
    Ajoute card à la main du joueur pid.
     :param pid (int):
    :param card( Card):
    :return:
```

appendCardToMyHand(self, card)

```
    Ajoute card à la main du joueur courant
     :param pid (int):
    :param card( Card):
    :return:
```

getCurrentPlayerHand(self)

```
    Retourne la main du joueur courant
     :return: un CardStack
```

getHandBestCard(self, kind=None)

```
    Renvoie la meilleure carte en main.
    Si kind est specifié, renvoie la meilleure carte d'un type en part

    :return (Card):
```

getHandCard(self, value, kind=None)

```
    Renvoie la carte de valeur 'value' ( de type kind si spécifié,
    une au hazard sinon )
```

```
    :param value (int):
    :param kind (String):
    :return: (Card)
```

getPlayerHand(self, pid)

```
     Cette fonction renvoie ma main du joueur à partir d'un pid
    :param pid: int
    :return: CardStack
```

handGotKind(self, pid, kind)

```
    Cette fonction renvoie true si la main du joueur 'pid' contient

    une carte de type 'kind'.

    :param pid (int):
    :param kind (String):
    :return: ( boolean )
```

handHasCardValue(self, cardValue)

```
    Renvoie true si la main du joueur courant contient la carte

    de valeur 'cardValue', false sinon,

    :param cardValue( int ):
    :return:
```

isPlayerHandEmpty(self, pid)

```
    True si la main du joueur pid est vide, false sinon
    :param pid:
    :return:
```

sortAllHandsByValue(self)

```
    Trie tous les main par valeur en ordre croissant.
    :return:
```

sortHandByValue(self, pid)

```
    Trie la main de 'pid' par valeur en ordre croissant.
```

6.3 Les méthodes pour gérer la distribution

choseRandomDealer(self)

```
    Fonction qui choisi un dealer au hazard.
    :return: None
```

dealCards(self, nb_cards)

```
    Distribution de nb_cards en sens antihoraire
    :param nb_cards:
    :param deck_pid:
```

dealTo(self, pid, nbCards)

```
    Ditribue nbCards au joueur (pid)
    :param pid (int):
    :param nbCards (int):
    :return: None
```

dealToTable(self, nbCards)

```
    Déplace nbCards cartes du jeu de carte principale à la table
    :return: None
```

getDealerPID(self)

```
    Retourne le pid du distributeur des cartes.

    :return (int): -1 si le dealer n'est pas defini.
```

setDealer(self, pid)

```
    Designe le distributeur des cartes (pid).
    :param pid (int):
    :return: None.
```

6.4 Les méthodes pour gérer la table

appendCardToTable(self, card)

```
    Ajoute la carte card à la table.
    :param card (Card):
    :return: None
```

flushTable(self)

```
    Efface le contenu de la table
```

getCardFromTable(self, index)

```
     Récupère la carte 'index' de la talbe
    :param (int) index: L'index de la table
    :return:
```

getLastCards(self, nb)

```
    Renvoie un tableau avec les nb dérnières cartes.
     :param nb (int):
    :return: (CardStack)
```

getNbTableCards(self)

```
    Renvoie le nombre de cartes sur la table.

    :return (int):
```

getTable(self)

```
    Cette fonction renvoie les cartes de la table
    :return: CardStack
```

getTableBestCardOwner(self)

```
    Retourne le pid du joueur avec la carte de valeur plus haute
    présent sur la table.
    :return: pid (int):
```

getTableDeck(self)

```
    Retour le jeu de cartes principal
    :return:
```

getTurnTableKind(self)

```
    Renvoie le type de tour courant
     :return (String):
```

popCardFromTable(self, ind=0)

```
    Prend la carte d'index ind de la table.
```

```
     :param index (int): ( default 0)
    :return: (Card)
```

resetTable(self)

```
    Fonction qui reset la table.

    :return: None
```

6.5 Les méthodes pour gérer le tours

setFirstPlayer(self, pid)

```
    Affecte le premier joueur au jouer pid
    :param pid:
    :return:
```

setLastPlayer(self, pid)

```
     Affecte le dernier joueur au joueur pid
    :param pid:
    :return:
```

next_turn(self)

```
    Fonction qui passe le tourne
```

isPlayerTurn(self)

```
    True  si c'est le tour du joueur
    :return: None
```

iAmLastPlayerToPlay(self)

```
    Renvoie true si le joueur courant est le dérnier joueur du tour.
    :return:
```

currentTurn(self)

```
    Renvoie le tour courrant
    :return:
```

getCurrentPhase(self)

```
    Retourne la phase de jeu courante.
     :return (String):
```

setCurrentPhase(self, phaseName)

```
    Affecte la phase de jeu courante.
     :return: None
```

setCurrentTurn(self, turn)

```
    Change current turn
    :param turn (int):
    :return: None
```

setFirstAndLastPlayers(self, first, last=None)

```
    Affecte le premier et le dernier joueur.
```

appendLogInfoMessage(self, message)

```
    Ajoute un message au log en bas à droite.
    :param message: texte du message à afficher
    :return:
```

6.6 Les méthodes pour gérer une action

defAgentAction(self, type, originSprite=None, originDrawable=None)

```
    Cette fonction définit une action de l'opposant.

    :param type (String): Le type de l'action.
    :param originSprite (ClickableSprite): Un sprite d'origine.
    :param originDrawable (Drawable): Un drawable.
    :return: (Action): Une action.
```

getAction(self)

```
    récoupère l'action courante

    :return:
```

6.7 Les méthodes pour gérer les cartes

isMine(self, card)

```
     Renvoie true si une carte appartient à la main du joueur
     courant, false sinon.
     :param (Card) card: la carte en question
     :return: boolean
```

defCard(self, value, kind)

```
    Définit une carte manuellement
    :param value: la valeur de la carte
    :param kind: le type de la carte
    :return:
```

getCardsPoints(self, pid, cards)

```
    Affecte au joueur pid la valeur de la liste des cartes
    passées en argument
    :param (int):
    :param List:
    :return:
```

getCardsValueSum(self, cards)

```
    Renvoie la somme de la valeur cartes dans la liste
    :param cards (list):
    :return:
```

getSelectedCard(self)

```
    Renvoie la carte selectionné
    :return: (Card)
```

getValeurCarte(self, carte)

```
    Renvoie la valeur de la carte selon les régles du jeu.

    :param carte (Card):
    :return: (int)
```

setCardValues(self, cardValues)
```
 Set the value of every card
:param cardValues:
:return:
```
playSelectedCard(self)
```
Joue la carte selectionné.
:return:
```
playCard(self, card)
```
Joue la carte 'card'

:param int card:
:return:
```

6.8 Les méthodes pour gérer les joueurs

getCurrentPlayer(self)
```
Renvoie le joueur courant
:return:
```
getCurrentPlayerDeck(self)
```
Renvoie le jeu de cartes du joueur
:return (int):
```

getCurrentPlayerPID(self)
```
Renvoie le PID du joueur courant.
:return:
```
getFirstPlayer(self)
```
Renvoie le pid du premier joueur à jouer

:return:
```
getLeftPlayerOf(self, pid)
```
Renvoie le pid du joueur à la gauche de pid
:param pid (int):
:return:
```
getPlayer(self, pid)
```
Retourne un le PlayerState du joueur (pid)

:param pid (int): Le pid du joueur que l'on veut obtenir
:return: un PlayerState
```
getRightPlayerOf(self, pid)
```
Renvoie le pid du joueur à la droite de pid
:param pid (int):
:return:
```
getnbPlayers(self)
```
 Renvoie le nombre de joueurs.
```

6.9 Les méthodes pour gérer les fenêtres d'info

showBlockingDialogMessage(self, message, title='Info', buttonText='Ok')

```
    Affiche un dialog message qui met le jeu en pause.
    :param message: Le message
    :param title: Le titre
    :param buttonText: le texte du bouton
    :return: None
```

showDialogAction(self, title, message, buttonText, action)

```
    Affiche un dialog message enregistre une action.
```

hideDialogMessage(self)

```
    Cache le dernier dialog message.
    :return:
```

initDialog(self, title, msg, tbutton)

```
    Initialise le dialog.
```

6.10 Les méthodes pour gérer le jeu de cartes

pickCardFrom(self, pid)

```
    Déplace une carte du jeu de carte  pid à sa main

    :param pid (int): Le pid du joueur résponsable de l'action
    :return:
```

isDeckEmpty(self)

```
    Renvoie false si le jeu de cartes est non vide, true sinon
    :return:
```

removeRangeOfCards(self, cardStack, inf=0, sup=13, kinds=[])

```
    Supprime les cartes de 'cardStack' qui ont une valeur comprise ent
    [inf, sup] et qui font partie de un des types de la liste 'kind' p
     h, c, s et d.

    :param inf (int): borne inferieure (comprise) (default = 13).
    :param sup (int): borne superieure (comprise) (default = 0).
    :param cardStack (CardStack):
    :param kind (list of String): une liste de types parmi h, c, s et
    :return: Un CardStack modifié.
```

6.11 Les méthodes pour gérer le jeu

initGameState(self)

```
    Initialise le jeu de cartes et appele la fonction
        pluginInit redéfinie dans le plugin.

    :return: InitState
```

initPlayers(self, nbPlayers)

```
    Initialise le numero de joueurs.

    :param nbPlayers (int) : Le numero de joueurs
    :return: None
```

isLost(self)
```
    Renvoie true si la partie est perdue, false sinon
    :return:
```
isWin(self)
```
    Renvoie true si la partie est gagné, false sinon
    :return:
```
kindToStr(self, kind)
```
    Traduit un type de carte en String,
    :param kind:
    :return:
```
lose(self)
```
    True si la partie est perdue, false sinon.
    :return:
```
nextState(self, gameState, agent_action)
```
    Fonction nextState abstraite à implémenter dans le plugin.
    :return:
```
pause(self)
```
    Met le jeu en pause.
```
pid(self, object)
```
    Renvoie le pid d'un objet
    :param object:
    :return: (int) pid
```

pluginInit(self)
```
    Fonction init abstraite à implémenter dans le plugin.
    :return:
```
restartGame(self)
```
    Réinitialise le jeu.
    :return:
```
unPause(self)
```
    Remet le jeu en éxécution.
```
win(self)
```
    La partie est gangé
    :return:
```

6.12 Installation

Prérequis :

- pyglet
- pip3
- git
- Python3

Assurez vous d'avoir installé Python3 sur votre machine.

Vous pouvez en suite installer pyglet avec l'outil *pip3* à partir d'un terminal avec la commande suivante :

```
$ pip3 install pyglet
```

Téléchargez maintenant le code source du projet avec le gestionnaire git :

```
$ git clone https://github.com/digitex3d/pyCE.git
```

Déplacez vous dans le répertoire pyCE et exécutez le moteur avec un plugin au choix parmi « belote » ou « bataille » ainsi :

```
$ ./pyce belote
```