Pychess

I - Présentation du projet

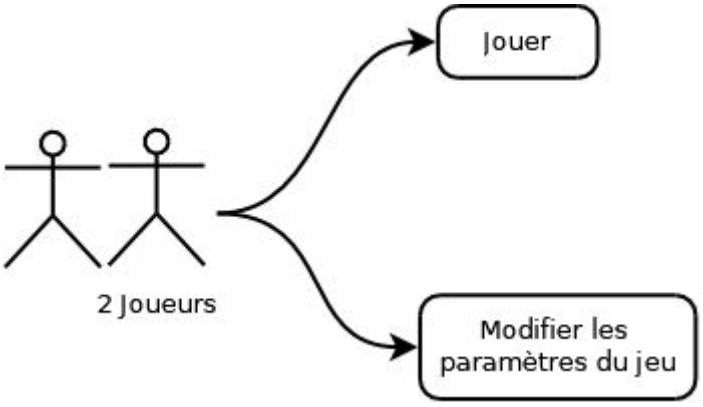
Pychess est un jeu d'échecs développé avec le langage Python 3.7 et la librairie Pygame. C'est un projet personnel que j'ai développé durant mon temps libre pour développer et améliorer mes compétences en programmation.
J'ai utilisé le langage Python car c'est un langage orienté objet que j'apprécie particulièrement.

II - Représentation du projet

a - Présentation pygame

Pour développer le projet, j'ai utilisé la librairie pygame (https://www.pygame.org/news) qui est une librairie open-source basée elle même sur la librairie SDL. Elle permet de faciliter l'accès et la gestion aux flux d'entrée et de sortie tels que les événements du clavier, l'affichage à l'écran etc … C'est pour ces raisons que j'ai décidé de l'utiliser.

b - Diagramme de cas d'utilisation



Ici on peut voir que le jeu ne peut pour l'instant que se jouer à deux joueurs devant l'ordinateur en un contre un, ils peuvent seulement jouer ou modifier les paramètres du jeu.

c - Représentation UML des classes du projet

d - Arborescence du projet

```
|  /
|  config.json
|  listing.txt
|  Main.py
|  README.md
+---BasicObjects
|  |  BaseCanvas.py
|  |  BaseObject.py
|  |  BaseState.py
|  |  BaseWidget.py
|  |  MyBaseProcess.py
+---canvas
|  |  GameCanvas.py
|  |  HomeCanvas.py
|  |  LogCanvas.py
|  |  SettingsCanvas.py
|  |  __init__.py
+---display
|  |  Button.py
|  |  ClickableImage.py
|  |  FlashMessage.py
|  |  GUI.py
|  |  TextToDisp.py
|  |  Window.py
|  |  __init__.py
+---game
|  |  Bishop.py
|  |  King.py
|  |  Knight.py
|  |  Pawn.py
|  |  Piece.py
|  |  Player.py
|  |  Queen.py
|  |  Rook.py
|  |  __init__.py
+---res
|  +---font
|  |      GOOD_DADDY.otf
|  |      terminal_font.TTF
|  |
|  \---img
|      |  chess_2.png
|      |  icon.png
|      |
|      +---buttons
|      |      back_rounded_corner.png
|      |
|      +---grids
|      |      chess_plate_1.png
|      |      chess_plate_2.png
|      |      chess_plate_3.png
|      |
```

```
|       \---pieces
|           +---black
|           |       bishop.png
|           |       king.png
|           |       knight.png
|           |       pawn.png
|           |       queen.png
|           |       rook.png
|           |
|           +---blue
|           |       bishop.png
|           |       king.png
|           |       knight.png
|           |       pawn.png
|           |       queen.png
|           |       rook.png
|           |
|           +---red
|           |       bishop.png
|           |       king.png
|           |       knight.png
|           |       pawn.png
|           |       queen.png
|           |       rook.png
|           |
|           \---white
|                   bishop.png
|                   king.png
|                   knight.png
|                   pawn.png
|                   queen.png
|                   rook.png
|
+---states
|   |   GameState.py
|   |   HomeState.py
|   |   SettingsState.py
```

Je n'ai pour le moment pas trouvé de meilleure façon de représenter le répertoire, ce texte a été généré avec la commande *tree* de l'invite de commande Windows.

III - Description technique

a - Fonctionnement

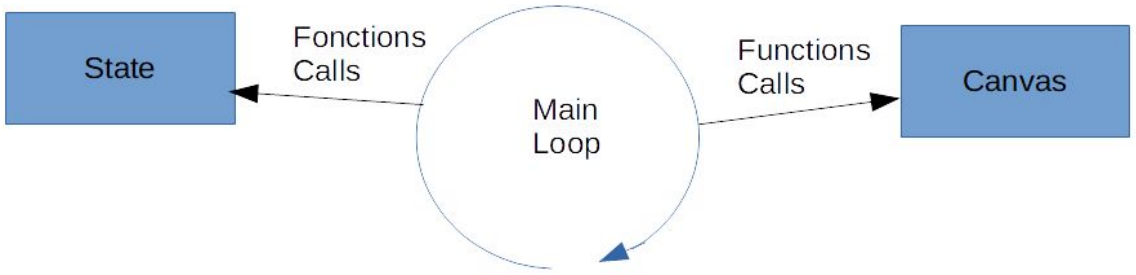
Pour que le programme puisse fonctionner et rester en fonctionnement, il lui faut une boucle qui n'est pas quittée tant que l'utilisateur n'a pas souhaité de quitter le programme (ou si un bug survient).

Dû à un manque de compétences au moment ou j'ai commencé le projet, la boucle principale du jeu se trouve dans la classe **GUI** car il y avait un problème de programmation concurrente notamment pour le partage de mémoire et les divers échanges entre les Threads/Processus. Cela vient du fait que la fenêtre que génère pygame est elle même un thread et je n'ai pas su le gérer au moment ou j'ai débuté. Cela reste un point à modifier dans le projet.

Le programme est divisé en 3 parties :

1. Le "main", c'est ce qui permet de faire la passerelle entre les deux autres parties et qui s'occupe de stocker et gérer des données liées au fonctionnement du programme comme la configuration. → Fichier **Main.py**
2. Le "state", c'est ce qui permet la gestion de la logique du programme pour un état donné. En effet le programme a divers état, par exemple "Home" ce qui correspond à l'accueil du programme et la classe "HomeState" permet de gérer par exemple les événements de clics sur les boutons. → Tous les fichiers du répertoire **states**.
3. Finalement, le "canvas" qui est utilisé pour dessiner dans la fenêtre. En effet, cette partie du programme va nous permettre de dessiner dans notre fenêtre tout ce que l'on souhaite, en l'état actuel du programme, il y a par exemple des boutons, des images cliquables, etc ...

Voici l'interaction représentée sous forme d'un diagramme simplifié :



Ce sont ces 3 parties qui constituent le programme et permettent son fonctionnement.
Nous allons maintenant nous intéresser aux divers états du programme et aux différentes parties qui les composent.

b - Les widgets

Nous commencerons par présenter les Widgets, qui seront nécessaires dans les prochaines explications du code.
Dans le programme, un Widget représente un objet graphique présenté à l'utilisateur, il peut ou pas être interactif (Texte à afficher / Bouton).

En l'état actuel du projet, on peut différencier deux types de Widgets :
Les widgets qui permettent l'interaction avec l'utilisateur et ceux qui ne le peux pas.

Les boutons :

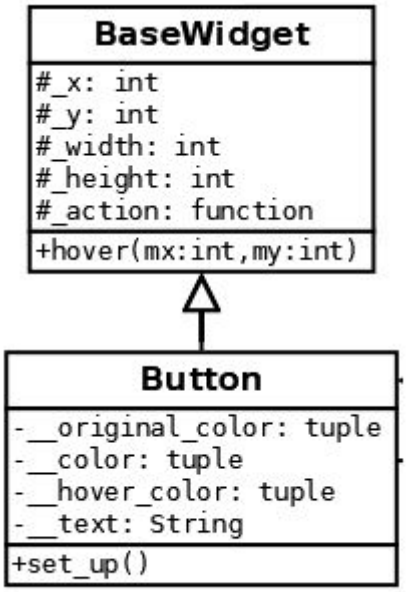
Les boutons sont très utilisés dans les interfaces graphique, c'est pour cela qu'il a été important de d'en développer dans le cadre de ce projet.

Ils sont caractérisés par :

1. Une position sur l'écran (x et y)
2. Une taille (width et height, pour largeur et hauteur)
3. Une couleur d'affichage et une couleur de superposition (quand l'utilisateur passe la souris dessus (elle est dite *hover_color*)
4. Un texte à écrire au milieu
5. Pour finir, le plus important, une *action*, elle correspond à l'action à performer lors du clique de l'utilisateur sur celui-ci.

Voici la classe **Button** extraite du diagramme UML :

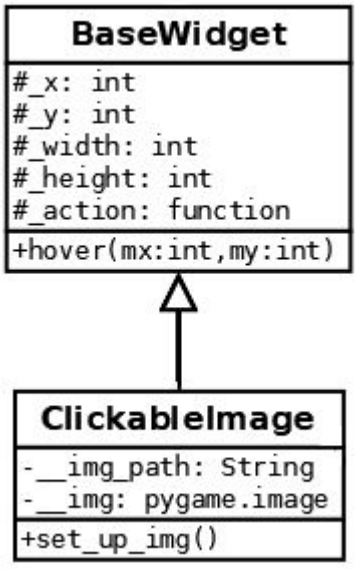
On peut y retrouver tous les attributs dans la classe mère ***BaseWidget*** ainsi que les attributs propres aux boutons.



<u>Les images cliquables :</u>

Les images cliquables ont été nécessaire lors de la création du menu des paramètres que nous verrons plus tard. En effet, cela permet d'afficher une image à l'utilisateur pour par exemple lui montrer un thème et si il souhaite le changer, il clique sur cette image.

Voici la partie du diagramme concernant les images cliquables, elles sont représentées dans le projet par la classe **ClickableImages**.



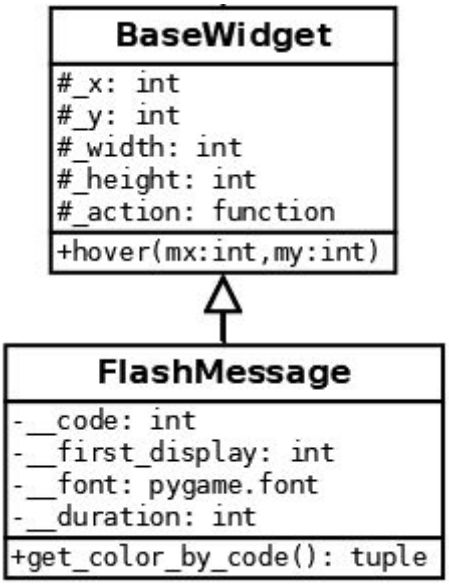
Nous allons maintenant nous intéresser aux widgets qui servent seulement à afficher des informations à l'utilisateur.

FlashMessage :

Les FlashMessage permettent d'afficher des messages dis Flash à l'utilisateur, c'est notamment utile pour lui indiquer lorsque des changements sont opérés ou des actions performées.
Voici à quoi ils ressemblent :

La couleur, le texte ainsi que la durée d'affichage sont modifiables.

Voici l'extrait du diagramme correspondant à la classe FlashMessage:



c - La classe GUI (Graphic User Interface)

Il y a dans le projet une classe destinée à la gestion de l'interface graphique, elle est nommée ***GUI***, c'est elle qui permet la gestion de l'affichage et comme il est dit plus haut, c'est ici que la boucle principale du programme se trouve (explications).

Voici à quoi ressemble la boucle principale :

```
while self._isRunning:
    # Draw everything on the GUI
    self.set_up_canvas()
    self.__canvas.draws()
    self.__window.blit(self.__canvas, (0, 0))
    pygame.display.flip()
    # Handling events
    super(GUI, self).handle_self_events()
    self.__main.handle_events(pygame.event.get())
    time.sleep(0.001)
```

On peut voir que beaucoup de méthodes sont appelées.

Premièrement, la méthode *set_up_canvas()*, cette méthode permet de d'effectuer un changement d'état. En effet, la classe ***GUI*** a un attribut ***__current_state_name*** qui permet de garder en mémoire le nom de l'état courant, par exemple "Home". Elle a également un attribut qui permet de garder en mémoire l'objet du canvas courant, il s'appelle ***__canvas***, grâce à ces deux attributs, nous pouvons détecter un changement d'état. C'est pour cela que cette méthode existe.

Voici le code de la méthode :

```
# There we check if the current state is not the same
if type(self.__canvas).__name__ != class_name:
    # Creating dynamically the state class
    try:
        # Import library
        module = import_module("canvas." + class_name)
        if class_name:
            state_class = getattr(module, class_name)
            # Setting the current state got in parameters as the new state
            self.__canvas = state_class(gui=self,
cfg=self._ownConfig["canvas"][current_state.lower()],
                                        master=self.__window,
width=self._ownConfig["def_w"],
                                        height=self._ownConfig["def_h"])
            self.__canvas.set_up()
            # self.__state.launch()
            self.__current_state_name = current_state
            # self.__gui.set_game_canvas()
    except Exception as exc:
        print(
            "Can't import : {lib}".format(lib=class_name))
        print(
            "Error message : {msg}".format(msg=exc.args))
```

On peut voir que dans un premier temps elle teste si le nom du Canvas courant et ***class_name*** (***__current_state_name + "Canvas"***) s'ils sont différents, alors elle opère un changement d'état si celui-ci est possible.

La méthode `self.__canvas.draws()` permet d'appeler la méthode du ***Canvas*** courant qui permet de dessiner sur la fenêtre.

Ensuite vient `self.__window.blit(self.__canvas, (0, 0))` et `pygame.display.flip()` qui sont deux méthodes de la librairie pygame qui respectivement, affiche un objet ***Surface*** (Objet de la libraire, ici le ***Canvas***) et mettent à jour l'affichage à l'écran.

Les deux dernières lignes :
```
super(GUI, self).handle_self_events()
self.__main.handle_events(pygame.event.get())
```
permettent la gestion des événements, nous verrons plus tard dans le rapport comment ils sont gérés.

Nous pouvons maintenant passer aux différents états du programme.

d - Les différents états

b.1 L'accueil

L'accueil est géré par 2 classes principales qui sont :

1. HomeState.py
2. HomeCanvas.py

Comme nous le verrons, seule les classes liées au "*state*" et au "*canvas*" seront affectées en fonction des états.

L'écran d'accueil du jeu

HomeCanvas.py

Nous allons donc maintenant nous intéresser à l'affichage de la page d'accueil.

Dans un premier temps, nous pouvons voir qu'il y a un Titre - qui correspond à l'attribut ***__title*** de la classe - ainsi que de boutons.
Les boutons se trouvent dans l'attribut hérité de la classe ***BaseCanvas*** "***_buttons***".

HomeState.py

La gestion de l'état de l'accueil est plutôt simple car il n'y a que deux boutons à gérer, c'est les seuls éléments avec lesquels l'utilisateur peut interagir.

Voici le code de sa méthode **handle_events(events)** :

```
# We go through all events
for e in events:
    if e.type == pygame.QUIT:
        self._main.stop_gui()
    elif e.type == pygame.MOUSEBUTTONUP:  # Event when the mouse button is released
        mx, my = pygame.mouse.get_pos()
        # We check if there is a button under the click
        for widget in self._main.get_buttons() + self._main.get_clickable_images():
            if mx > widget.get_x() and mx < widget.get_x() + widget.get_width() and
my > widget.get_y() and my < widget.get_y() + widget.get_height():
                widget.action()
    else:
        mx, my = pygame.mouse.get_pos()
        for widget in self._main.get_buttons():
            widget.hover(mx, my)
```

Nous pouvons voir que le premier événement traité est l'événement **pygame.QUIT**, il correspond au clic sur le bouton de fermeture de la fenêtre. Il est placé en premier car cela ne sert à rien de vérifier tout le reste si l'utilisateur souhaite quitter l'application.

Le second événement est **pygame.MOUSEBUTTONUP**, il correspond à l'event qui est déclenché lorsque le bouton gauche de la souris est relâché, cela permet de détecter un clique. La méthode regarde ensuite si il y a eu un clique sur un des widgets et fait appelle à l'action qui correspond au widget cliqué.

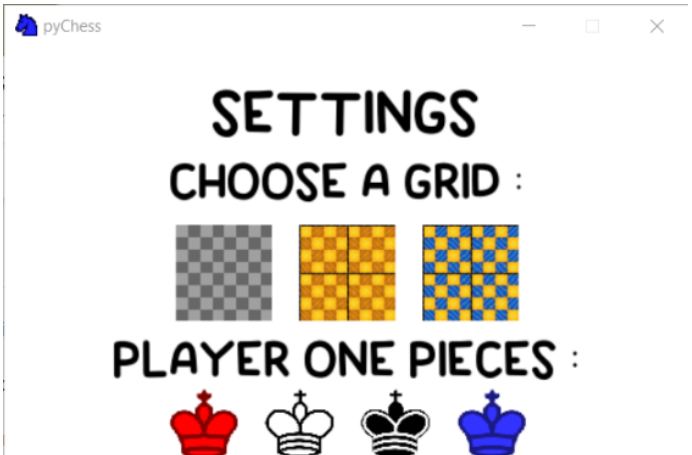
Dans le **else** la méthode permet de vérifier si il y a un bouton sous la souris, si c'est le cas, alors sa couleur est changée pour la couleur de hover.

Chaque bouton de l'accueil a une méthode assignée, le premier bouton "Play" permet de démarrer un nouvel état qui est "Game" et qui permet de jouer au jeu, le second "Settings" permet de passer à l'état "Settings" pour modifier les paramètres du jeu, nous allons maintenant nous intéresser à celui-ci.

b.2 Les paramètres

Les paramètres sont gérés par ces deux classes :

1. SettingsState
2. SettingsCanvas



On peut déjà voir que cet état est plus complet que l'accueil car il y a beaucoup plus de widgets. Cet écran permet aux joueurs de personnaliser le jeu à leur souhait dans les limites de ce qui est disponible. La seule contrainte qui leur est imposée est le fait que deux joueurs ne peuvent avoir la même couleur de pions pour le jeu.

Ecran de paramétrage du jeu

SettingsCanvas.py

Ce ***Canvas*** est plus complet que celui de l'accueil mais il n'en est pas plus complexe, il s'occupe surtout du positionnement des éléments dans la fenêtre. La fenêtre n'étant pas redimensionnable pour le moment, les valeurs sont calculées une seule fois à l'affichage et sont gardées en mémoire.

Voici un extrait de la méthode ***set_up_pieces_img_p1()*** :

```
x = 120
y = 250
new_img = ClickableImage(x=x, y=y, width=50, height=50,
img_path="res/img/pieces/red/king.png",
                         action=self.change_p1_pieces_red)
new_img.set_up_img()
self._clickable_images.append(new_img)
```

On peut voir que les valeurs x et y sont inscritent en dur et qu'ensuite un widget ***ClickableImage*** est créé, initialisé puis ajouté à la liste des ***ClickableImage***s du ***Canvas***.

SettingsState.py

La classe ***SettingsState*** est quasiment identique à celle de ***HomeState*** sauf qu'au lieu d'avoir seulement les boutons dans la méthode ***handle_events(events)*** il y a également les ***ClickableImages***, voici à quoi ressemble le ***for*** de cette méthode :

```
for widget in self._main.get_buttons() + self._main.get_clickable_images():
```

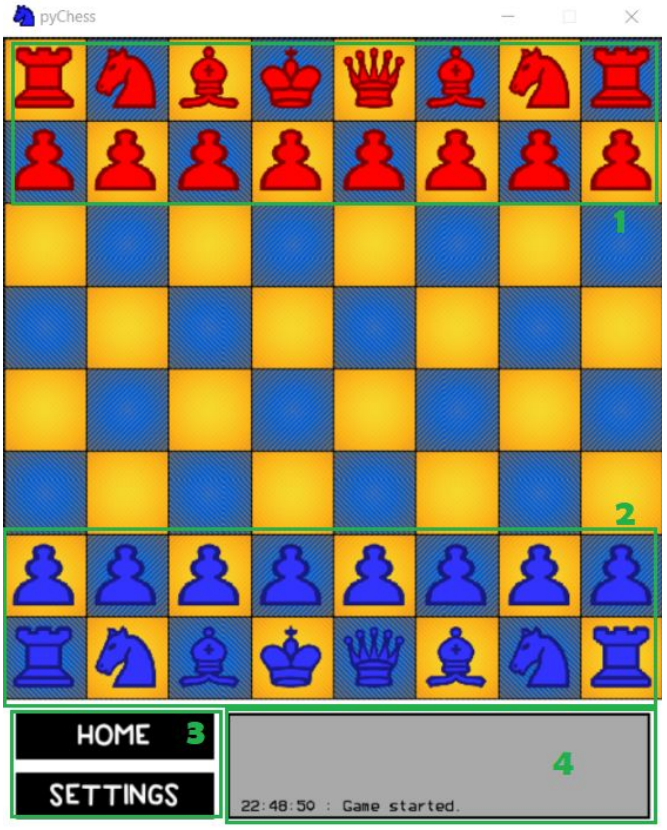
On peut voir qu'on ajoute les ***Button***s ainsi que les ***ClickableImage***s à la collection qui doit être itérée. Les tests effectués sont les mêmes ensuite, c'est à dire que la méthode vérifie les cliques ainsi que les hovers.

b.3 Le jeu à deux joueurs

Nous allons maintenant passer à l'état du programme le plus complet et complexe, l'état du jeu. Pour le moment, seul une fonctionnalité de 1 contre 1 en local est disponible, mais de nouveaux modes devraient voir le jour tels qu'un mode contre une intelligence artificielle, un mode 1 contre 1 en réseau etc …

Comme cet état est complexe, j'ai mit des marques avec des numéros sur l'image pour pouvoir repérer plus simplement les éléments.

1. Les pions rouges correspondent aux pions du joueur n°1
2. Les pions bleus correspondent aux pions du joueur n°2
3. Le cadre numéro 3 correspond aux boutons disponibles sur l'écran de jeu avec lequel le joueur peut interagir.
4. Le cadre numéro 4 est un logger qui permet d'afficher des informations aux joueurs. C'est un ***Canvas*** à part entière.



Ecran en jeu

GameCanvas.py

Cette classe est beaucoup plus complète que les autres déjà présentent car il y a beaucoup plus d'élément à afficher et à mettre à jour. Il y a notamment un élément nouveau qui n'est présent dans aucun autre ***Canvas*** c'est la présence d'un ***Canvas*** dans un autre. En effet, le numéro **4** sur l'image est un logger, mais c'est un objet ***Canvas***, il est intégré au ***GameCanvas***. J'ai pris la décision d'en faire un ***Canvas*** car c'est un objet graphique complexe qui pourrait être réutilisé dans une autre situation, c'est pour cela que c'est une classe à part entière.

Nous allons nous intéresser à la méthode ***draw_entities()*** , voici son implémentation :

```
# We get all the pieces of the players
players_pieces = self._gui.get_pieces()
# For each player
for p_player in players_pieces:
    # For each player's piece
    for i, piece in enumerate(p_player):
        # If the piece is not selected
        if not piece.is_selected():
            # Setting up the x, y pos of the piece
```

```
            y = piece.getY() * self.__square_size + int(15 / 2)
            x = int(piece.getX() * self.__square_size) + int(15 / 2)
            self.blit(piece.getImg(), (x, y))
        else:
            # If the piece is selected, then we need to make it follow the mouse
            self.blit(piece.getImg(), (
                int(pygame.mouse.get_pos()[0] - (62.5 / 2)),
int(pygame.mouse.get_pos()[1] - (62.5 / 2))))
```

Cette méthode permet de dessiner à l'écran toutes les entités donc ici les pièces des joueurs.
La méthode procède comme ceci :

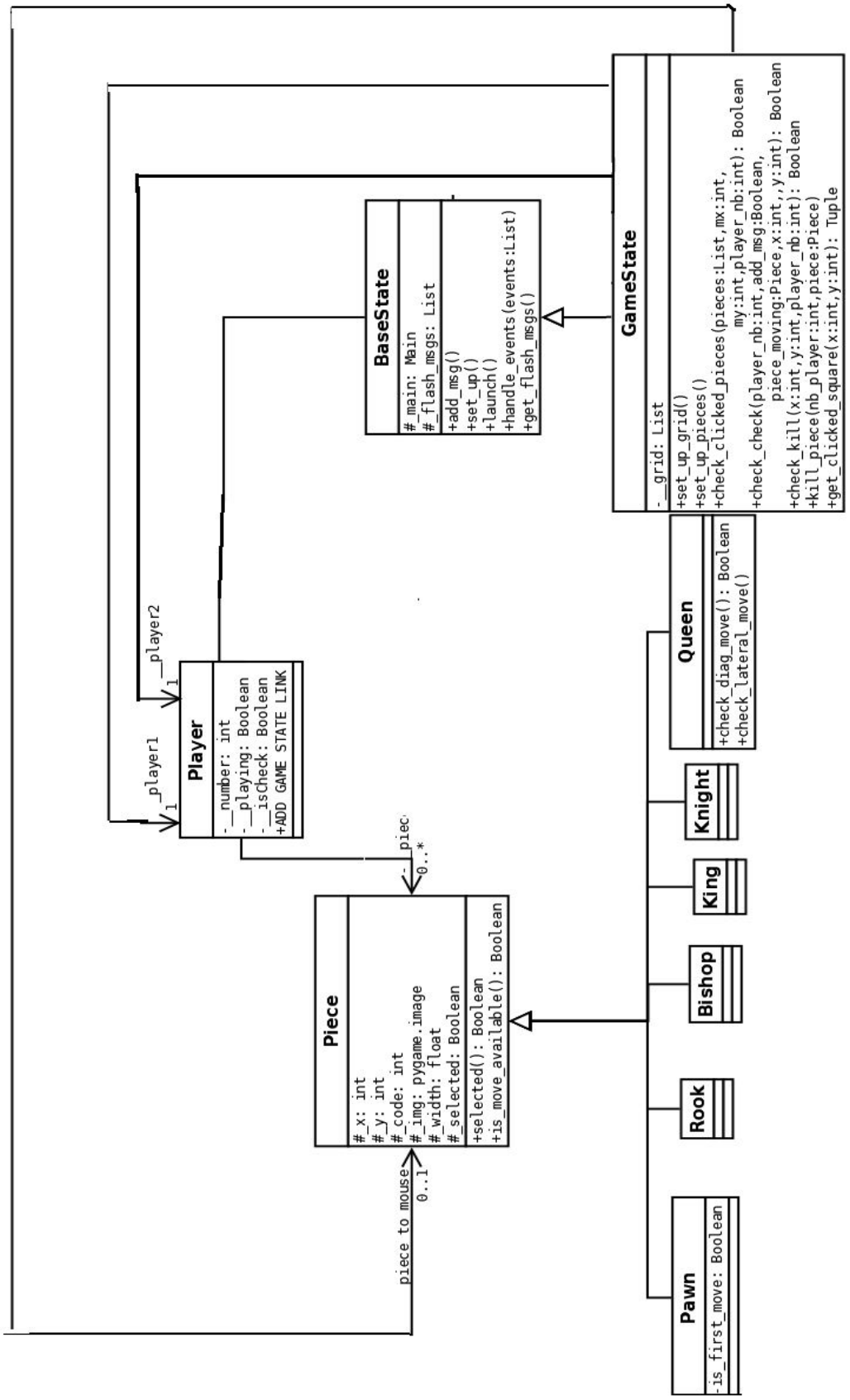
1. Elle récupère les pièces des joueurs grâce à son attribut ***_gui***
2. Elle fait une itération d'abord à travers la liste qui lui est renvoyée qui est une liste de 2 tableaux contenant la liste des pièces respective des joueurs et elle fait finalement une itération sur les pièces de chaque joueur.
3. Elle fait un test ***if*** qui permet de tester si la pièce est sélectionnée, c'est à dire si le joueur a cliqué dessus pour la déplacer. Si c'est le cas, alors la pièce sera affichée aux coordonnées de la souris pour indiquer à l'utilisateur qu'il est entrain de déplacer sa pièce. Sinon elle est dessinée aux coordonnées x et y qui lui correspondent.

GameState.py

C'est maintenant à la plus grosse partie du programme que nous allons nous intéresser. C'est la classe qui gère toute la logique du jeu.

ATTENTION : Certaines fonctionnalités ne sont pas encore implantées dans le jeu telle que l'échec et mat. Le jeu n'est donc pas complétement fonctionnel, il est encore en développement.

Comme cette classe est complexe, voici son diagramme UML avec ses association pour avoir une meilleur compréhension des différentes opérations présentes.



Voici le fonctionnement de la classe :

1 - Initialisation :

Dans un premier temps, lorsque le jeu commence, la classe doit tout initialiser pour que la partie puisse commencer.
Cette initialisation est faite grâce aux méthodes :

1. Le constructeur : Il créer les deux joueurs de cette façon :

```
self.__player1 = Player(cfg=self._ownConfig["players"]["1"], game=self, number=1)
self.__player2 = Player(cfg=self._ownConfig["players"]["2"], game=self, number=2)
```

Une fois cette étape performée, les deux joueurs sont créés.

2. Ensuite la méthode ***set_up_pieces()*** est appelée. Elle permet d'initialiser les pièces des joueurs et de leur attribuer.

Voici un extrait de cette méthode :

```
pieces = []
# Player 1
pieces.append(Rook(y=0, x=0, code=Piece.ROOK_CODE, player=self.__player1))
pieces.append(Knight(y=0, x=1, code=Piece.KNIGHT_CODE, player=self.__player1))
pieces.append(Bishop(y=0, x=2, code=Piece.BISHOP_CODE, player=self.__player1))
pieces.append(King(y=0, x=3, code=Piece.KING_CODE, player=self.__player1))
pieces.append(Queen(y=0, x=4, code=Piece.QUEEN_CODE, player=self.__player1))
pieces.append(Bishop(y=0, x=5, code=Piece.BISHOP_CODE, player=self.__player1))
pieces.append(Knight(y=0, x=6, code=Piece.KNIGHT_CODE, player=self.__player1))
pieces.append(Rook(y=0, x=7, code=Piece.ROOK_CODE, player=self.__player1))
for i in range(8):
    pieces.append(Pawn(y=1, x=i, code=Piece.PAWN_CODE, player=self.__player1))
    # pass
self.__player1.setPieces(pieces)
```

Cela crée et initialise les différentes pièces du joueur n°1 pour ensuite lui ajouter à son tableau qui contient les pièces.

3. La dernière phase d'initialisation est de mettre l'attribut ***__playing*** de ***__player1*** à *True* pour indiquer au jeu que c'est lui qui va commencer à jouer.

2 - Le jeu :

Etant donné qu'aucun événement n'est déclenché par le jeu (Comme des objets qui évoluent dans le temps par exemple) mais par les événements que les joueurs déclenchent, tous les appels aux méthodes sont situé dans la méthode ***handle_events(events)***. Cette méthode est utilisée pour gérer les événements du jeu et agir en fonction de ceux-ci.

C'est la même méthode que les classes qui héritent également de ***BaseState*** sauf qu'elle est beaucoup plus remplie.

Dans un premier temps, elle itère à travers la liste ***d'events*** qui lui sont donnés en argument.
Le premier événement vérifié est l'événement qui indique que l'utilisateur a cliqué sur la croix de la fenêtre pour terminer l'application.

```
if event.type == pygame.QUIT:
    self._main.stop_gui()
```

Si cet event ne correspond pas, le programme passe à la suite.

Voici la suite de la méthode :

```
elif event.type == pygame.MOUSEBUTTONUP:
1  ----------------------------------
    # Getting mouse coordinates
    mx, my = pygame.mouse.get_pos()
    # Checking button click
    for b in self._main.get_buttons():
        if mx > b.get_x() and mx < b.get_x() + b.get_width() and my > b.get_y() and
my < b.get_y() + b.get_height():
            b.action()
    # Checking if it's the first player is playing
2  ----------------------------------
    if self.__player1.is_playing():
        # If it's him then we can check what to do with his click
        played = self.check_clicked_pieces(self.__player1.getPieces(), mx, my,
self.__player1.getNumber())
        if played:
            # self.check_check(1)
            # Change playing player
            self.__player1.set_playing(False)
            self.__player2.set_playing(True)
            self.check_check(2)
    else:
        played = self.check_clicked_pieces(self.__player2.getPieces(), mx, my,
self.__player2.getNumber())
        if played:
            # Change playing player
            self.__player2.set_playing(False)
            self.__player1.set_playing(True)
            self.check_check(1)
```

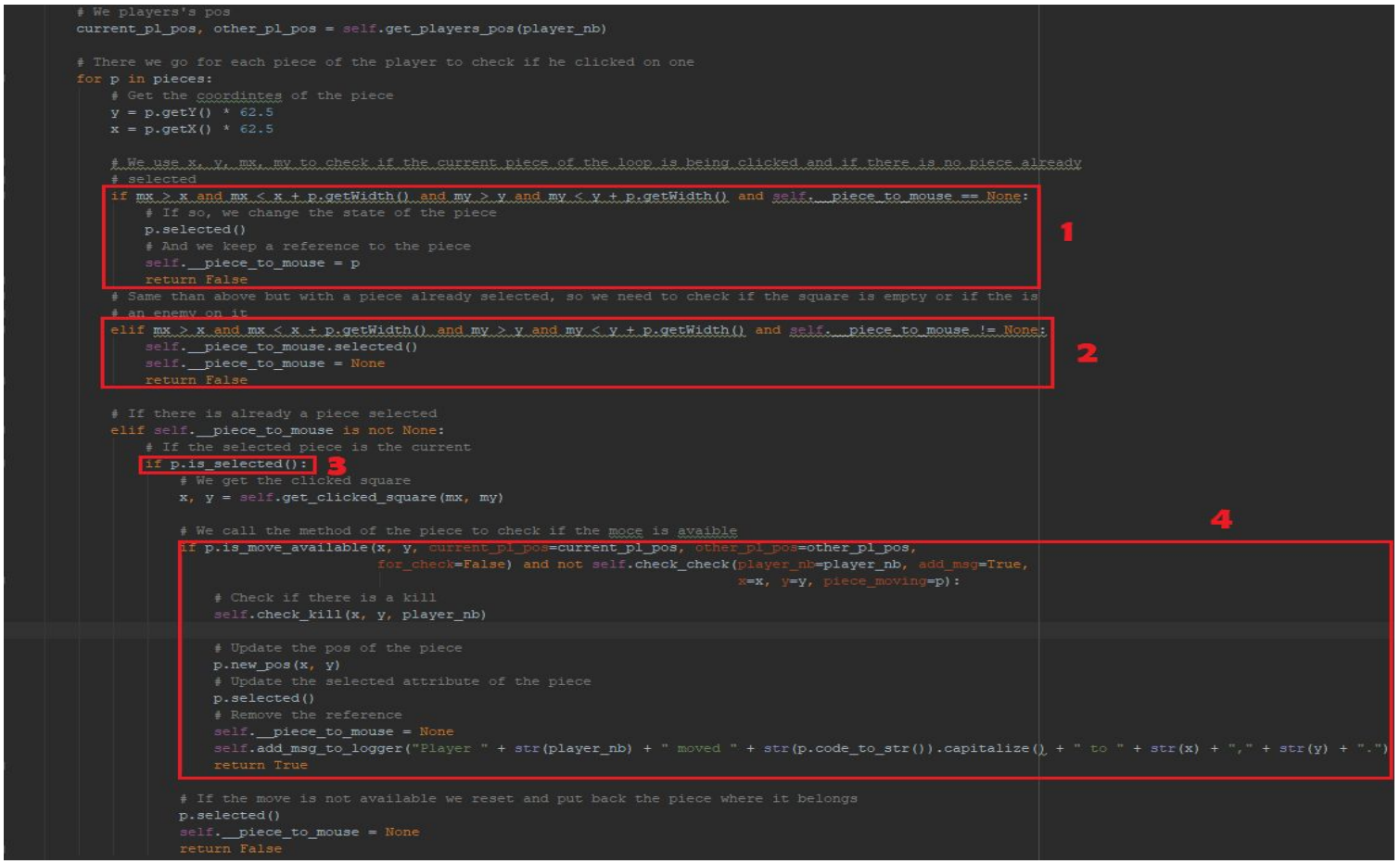
Dans le bloc n°1 la méthode s'occupe de vérifier si les boutons présents ont été cliqué. Si c'est le cas, elle appelle alors la méthode liée au bouton.

C'est dans le bloc n°2 que la logique même du jeu prend place.
La première partie est un test ***if*** pour savoir quel joueur joue et agir en conséquence, ce sont les mêmes actions mais qui chaque fois corresponde au joueur en train de jouer.

La ligne :

```
played = self.check_clicked_pieces(self.__player1.getPieces(), mx, my,
self.__player1.getNumber())
```

permet de faire appel à la méthode ***check_clicked_pieces(pieces : List d'objet Piece, mx : int pour la position x de la souris, my : int pour la position y de la souris, player_nb : int pour le numéro du joueur courant)*** qui permet de gérer les cliques sur les pièces et d'agir en conséquence. Elle retourne un booléen True si le joueur a déplacé sa pièce, False sinon.



Voici le corps de la méthode annoté pour une description :

Voici les étapes de la méthode :

1. Le corps de ce ***if*** s'exécute seulement si il y a une pièce ou le clique de la souris a eu lieu et si il n'y a aucune pièce de déjà sélectionnée (*if self.__piece_to_mouse == None*). Si il s'exécute, alors l'état de la pièce courante sera modifié, grâce à la méthode *selected()* qui passera de **False** à **True** l'attribut *__selected* de l'objet **Piece**. Il attribuera également cette pièce en tant que valeur pour l'attribut *__piece_to_mouse* de la classe pour pouvoir s'en servir plus tard. Le bloc se termine par un ***return False*** pour signifier à l'appelant que le joueur n'a pas jouer mais qu'il a juste sélectionner une pièce.
2. Ce bloc est semblable au premier seulement il diffère dans le sens ou c'est dans le cas ou il y a déjà une pièce de sélectionnée, cela a pour but de gérer les cas où le joueur clique sur l'emplacement d'origine de la pièce ou sur une autre de ses pièces. Il retourne également **False** car le joueur n'a pas jouer.
3. Ce 3ème bloc se situe dans le **elif** et il permet de vérifier que la pièce courante du **for** est la même que la pièce qui est déjà sélectionnée par le joueur. Cela permet ensuite de vérifier si le joueur :
    a. a pas cliqué sur l'une des ses pièces

    b. a cliqué sur un ennemi et agir en conséquence
    c. a cliqué sur une case qui n'est pas atteignable avec la pièce sélectionnée (par exemple une tour qui souhaite sauter par dessus un allié/ennemi)
4. Ce bloc permet de vérifier les situations énumérées ci-dessus. Dans un premier temps un ***if*** est réalisé sur l'appel de la méthode d'une pièce qui est ***is_move_available(x : coordonnée x cible de la pièce, y : coordonnée y cible de la pièce, current_pl_pos : liste des positions des pièces du joueur jouant, other_pl_pos : liste des positions des pièces du joueur adverse, for_check : Boolean permettant de savoir si on veut tester le cas d'échec/échec et mat)*** et sur la méthode ***check_check(player_nb : int numéro du joueur courant, add_msg : Boolean pour savoir si on affiche des messages aux joueurs, x : coordonnée x cible de la pièce, y : coordonnée y cible de la pièce, piece_moving : Piece la pièce qui est en train d'être déplacée)*** ces deux méthodes retournent des Booléen, la première retourne **True** si le déplacement de la pièce est possible et la seconde retourne **True** si le joueur jouant est en état d'échec et si son déplacement n'enlève pas cet état.
    a. Si les deux méthodes retournent **True**, alors le code est atteint. Ce code dans un premier temps vérifie si la pièce qui bouge tue un ennemie grâce à la méthode ***check_kill(x : int nouvelle coordonnée x de la pièce, y : int nouvelle coordonnée y de la pièce, player_nb : int numéro du joueur jouant)***. Elle met ensuite à jour la position de la pièce avec la méthode ***new_pos(x, y)*** puis met à jour l'attribut *__selected* de la pièce avec la méthode ***selected()***, déréférence la pièce de l'attribut *__piece_to_mouse* et ajoute un message au logger pour dire quel joueur a joué quelle pièce, à quelle position. Puis retourne **True** pour dire que le joueur a bien joué et qu'il a terminé son tour.
    b. Si il n'est pas atteint, alors le programme met à jour l'attribut *__selected* de la pièce avec la méthode ***selected()***, déréférence la pièce de l'attribut *__piece_to_mouse* et return **False** pour signifier à l'appelant que le joueur n'a pas joué.

Donc si la méthode décrite ci-dessus retourne **True** alors ce code est exécuté :

```
# Change playing player
self.__player1.set_playing(False)
self.__player2.set_playing(True)
self.check_check(2)
```

Il a pour effet de :

1. Mettre à jour l'attribut *__isPlaying* du joueur en train de jouer à **False** pour signifier que ce n'est plus lui qui est en train de jouer.
2. Mettre à jour l'attribut *__isPlaying* de l'autre joueur à **True** pour signifier que c'est à lui de jouer maintenant.