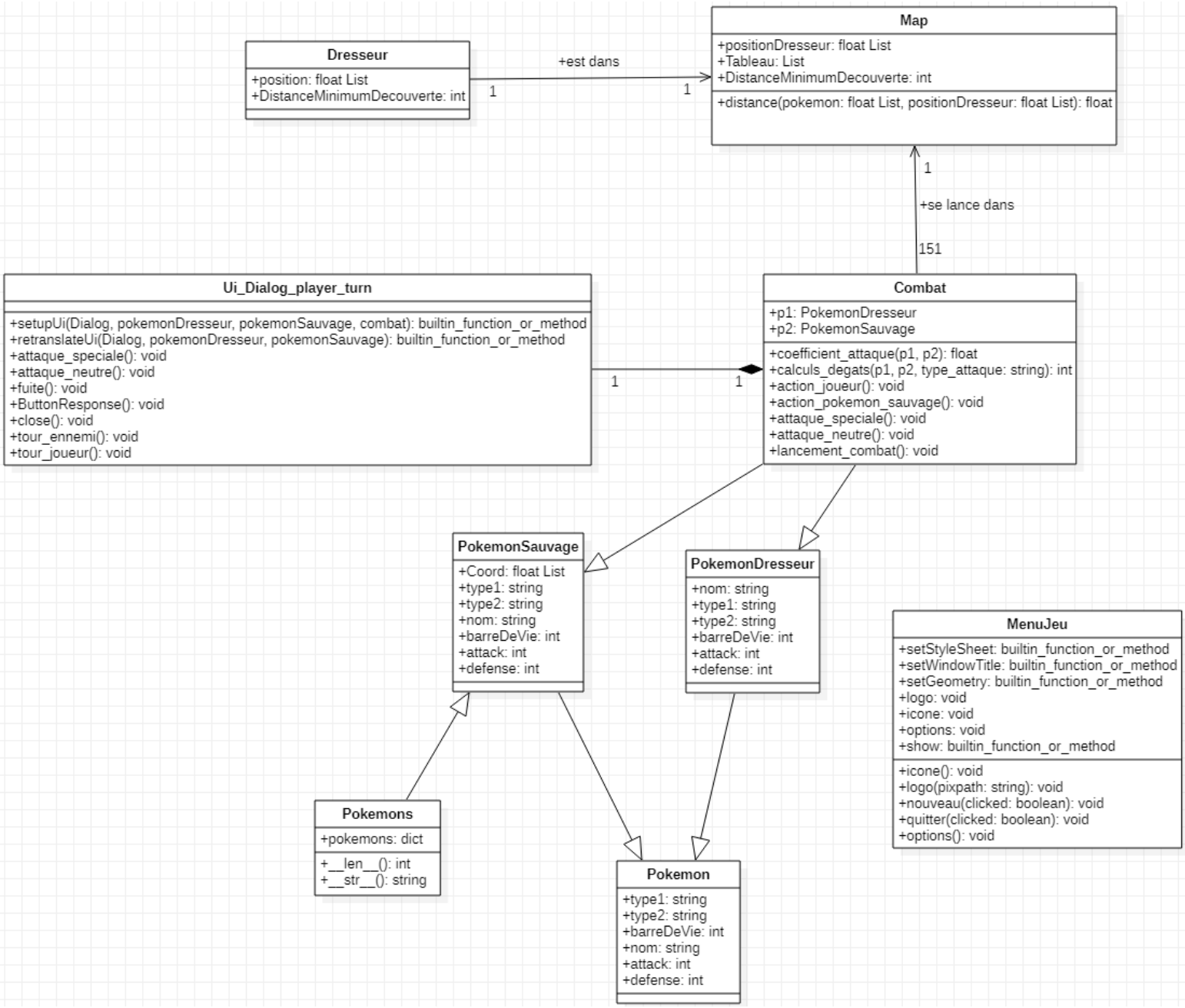
RAPPORT PROJET POKEMON

Organisation du code

Diagramme de classes :



Ce diagramme permet de représenter les structures de données intervenant dans notre jeu et les relations entre elles. On a défini 9 classes dans différents fichiers python décrits par la suite. Les relations d'héritage permettent de montrer que certaines classes utilisent les attributs et méthodes d'autres, comme Pokemons qui hérite de ceux de PokemonSauvage.

La relation de composition permet de montrer que la classe Ui_Dialog_player_turn est dépendante/n'existe pas sans la classe combat.

. Nous avons scindé le code en 6 parties principales (6 fichiers python). La première s'intitulant **"Pokemon"** qui permet dans un premier temps de récupérer les informations sur les pokémons utilisés, comme les points de vie, à l'aide du fichier csv "pokemon_first_gen", puis dans un second temps de créer un dictionnaire où pour chacun de ces pokémons on a leurs caractéristiques, et enfin de voir si ce pokémon est sauvage ou au dresseur.

. La seconde partie, **"player_turn"** permet de créer l'interface de combat ainsi que les mécaniques principales liées à l'interface graphique lorsque c'est le tour du joueur : la possibilité de victoire, de défaite, et les différents boutons cliquables, que sont l'attaque neutre, l'attaque spéciale et la fuite. Cette partie permet également de faire jouer l'adversaire, et définit les conditions de victoire et de défaite. Selon le bouton choisi par le joueur, une ou plusieurs fonctions sont appelées à l'aide de la méthode ButtonResponse() de la classe Ui_Dialog_player_turn qui gère l'interface graphique du combat.

. La troisième partie, nommée **"combat_code"**, utilise les deux premières. Elle utilise la première via les attributs récupérés pour chaque pokémon, pour définir les dégâts pris par ces derniers en fonction de leur type (faiblesse/résistance) via un coefficient multiplicateur appliqué aux attaques spéciales de la deuxième partie. Selon les cas, cela peut aller jusqu'à quadrupler la quantité de dégâts infligés, mais aussi les annuler complètement. Ceci s'applique au pokémon sauvage comme à celui du dresseur, et cette troisième partie permet également l'actualisation de l'interface graphique grâce à la seconde partie, notamment des pv des deux pokémons. C'est ensuite le tour du pokémon sauvage d'attaquer. On commence par cacher les boutons qui permettent au joueur d'effectuer une action pour éviter les problèmes de conflits (si le joueur appuie en même temps que l'ennemi joue cela peut créer des erreurs). Enfin, la fonction lancement_combat affiche et ferme la fenêtre de combat.

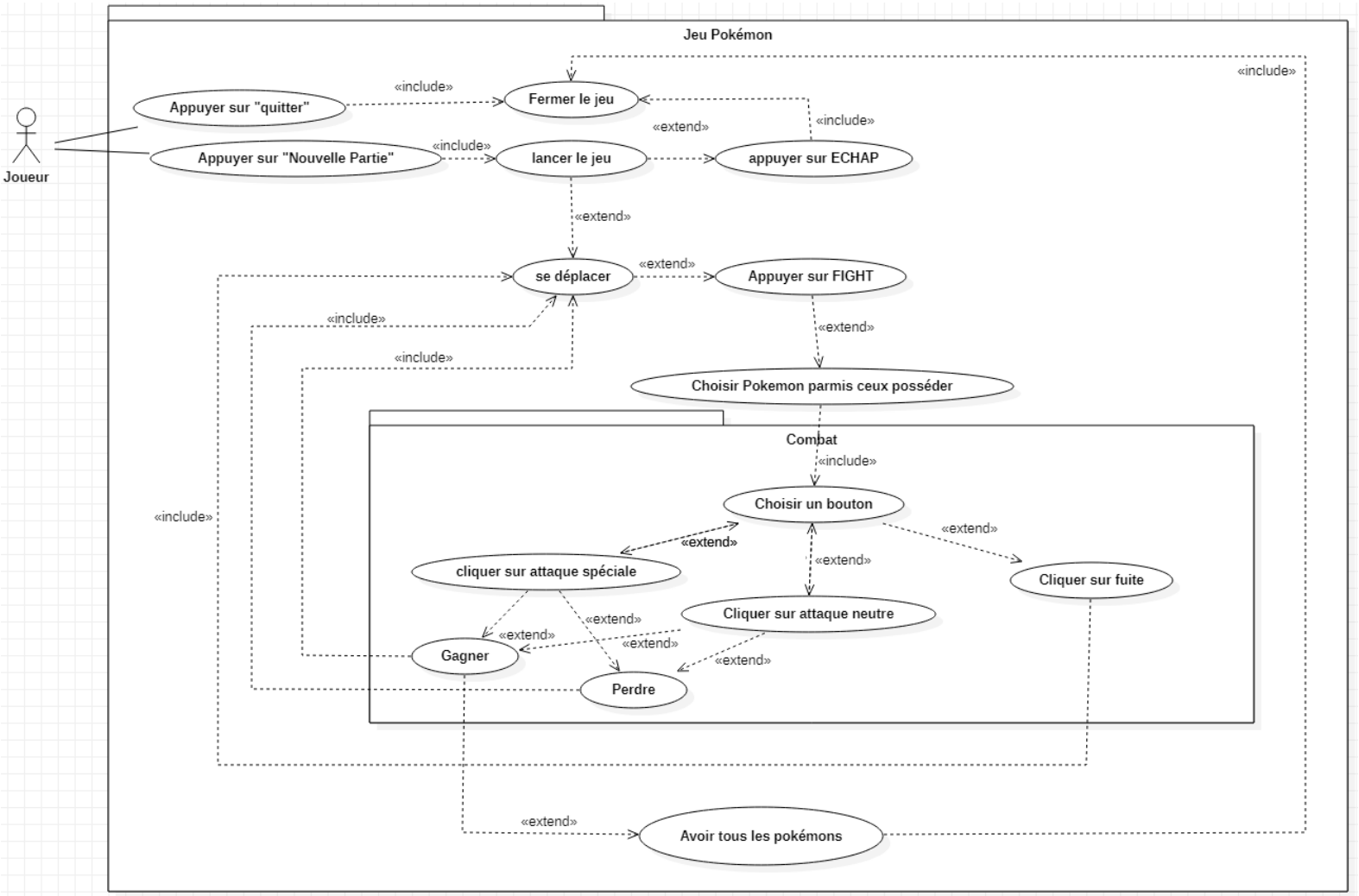
. La quatrième partie, **"ClassesMapDresseur"** sert à créer deux classes, Map et Dresseur. Tout d'abord est récupéré le tableau des noms des pokémons et de leurs coordonnées à l'aide du fichier csv "pokemon_coordinates" qui est réutilisé dans la classe Map. Cette classe permet de calculer la distance entre le dresseur et les pokémons sauvages, ce qui permettra par la suite de savoir s'il faut afficher le pokémon ou non. La classe Dresseur permet d'obtenir le pokémon de départ utilisé par le joueur dans une liste qui grossira au fil des captures du dresseur. Le pokémon de départ étant choisi aléatoirement dans le starter.

. La cinquième partie s'appelant **"main"** est le cœur du jeu permettant de faire fonctionner les quatre parties précédentes ensemble et de faire tourner le jeu en temps réel. Premièrement, on y charge les images de la carte, des pokémons à afficher, du dresseur, et du bouton de combat. Ensuite, on définit deux distances arbitraires, une à partir de laquelle on affichera le pokémon (MIN_DISTANCE), et une autre (DISTANCE_INTERACTION, plus

courte) à partir de laquelle on affichera le bouton permettant de lancer le combat. Dans la boucle du jeu, on affiche d'abord la carte, ainsi que le dresseur, et on implémente les mouvements en fonction de la touche utilisée. Pour afficher les pokémons, on utilise la distance définie arbitrairement et la classe Map de la partie quatre. Même chose pour le bouton de combat. Ensuite, si le joueur presse le bouton de combat, le combat se lance en utilisant la partie trois avec la fonction lancement_combat. On implémente avant le début du combat la possibilité pour le joueur de choisir le pokémon qu'il veut utiliser pour combattre selon ceux se trouvant dans sa liste définie dans la classe Dresseur de la partie quatre, au début il n'y en a qu'un. Dans le cas de défaite et de fuite, il ne se passe rien, la fenêtre de combat se ferme et le joueur retourne sur la map. En cas de victoire (si les pv du pokémon sauvage tombent à zéro) alors le pokémon est ajouté à la liste des pokémons possédés par le dresseur et il est effacé de la map. Enfin il y deux conditions à l'arrêt du jeu : soit le joueur presse la touche ECHAP, soit il a capturé les 151 pokémons présents sur la carte.

. Enfin, la sixième partie, **"Menu",** permet comme son nom l'indique d'initialiser un menu avant que le jeu commence, permettant de cliquer sur "nouvelle partie" ou quitter, appuyer sur nouvelle partie lance le code se trouvant dans "main", et appuyer sur "quitter" ferme simplement la fenêtre.

Diagramme de cas d'utilisation :



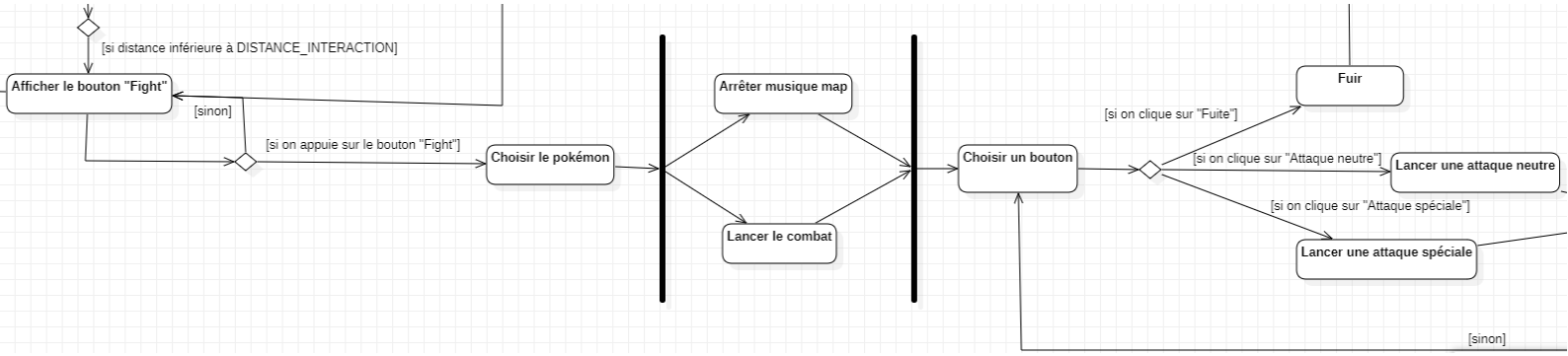
L’acteur principal est le joueur. Le diagramme est complémentaire à celui de classes puisqu’il représente l’ensemble des actions possibles pour le joueur au cours du jeu. Par exemple, la relation d’inclusion partant d’ « appuyer sur quitter » implique nécessairement de « fermer le jeu ». Pour la relation d’extension, « cliquer sur attaque spéciale » peut impliquer de « gagner » le combat, mais pas forcément.

Choix faits (adaptation/compromis)

. Les coordonnées fournies dans l’énoncé sont celles utilisées (elles sont décimales), de fait il a fallu adapter les coordonnées à la taille de la map pour que les pokémons rentrent tous dedans, et ils sont donc à des positions bien précises, parfois sur une colline ou sur un arbre, ce qui ne fait pas forcément sens, mais ce choix d’avoir gardé les coordonnées de l’énoncé pour les 151 pokémons a fait que nous n’avons pas pu implémenter de collisions avec le décor (hormis les bords de la map). Ces coordonnées impliquent aussi une superposition presque totale de certains pokémons sur la map, mais encore une fois, au vu de l’échelle de la carte, c’est un problème non solvable si on garde les coordonnées de l’énoncé.

. Nous avons choisi d’utiliser la librairie Pygame pour ajouter de la musique au jeu, et pour implémenter l’interface graphique de la carte, car elle permettait de lier facilement les déplacements du dresseur au clavier.
Pour l’interface graphique du menu, la librairie PyQt5 était très adaptée puisqu’il ne s’agit que de cliquer sur un bouton (qui permet soit de quitter le jeu, ce qui ferme la fenêtre, soit au contraire de le lancer, ce qui ferme le menu et ouvre la carte du jeu).
Pour la fenêtre de combat, PyQt5 convenait également très bien pour gérer les différents boutons apparaissant à l’écran et pour mettre à jour les PV de chacun des pokémons alternativement.

Diagramme d’activités :



Voir le diagramme complet en png sur le dépôt github

Ce diagramme est complémentaire aux deux autres car il donne une vue d'ensemble de la structure / l'organisation du code et permet de voir le déroulé global d'une partie. Le « fork », par exemple celui après « appuyer sur nouvelle partie » permet de montrer que plusieurs actions se déroulent simultanément, ici lancer la carte et la musique. Le « join » permet la synchronisation d'activités en concurrence, par exemple ici arrêter la musique de la map et lancer le combat, pour revenir à une activité linéaire, ici « choisir un bouton ». Enfin, les branchements conditionnels permettent de poser une condition pour faire le choix entre plusieurs activités, par exemple si lorsqu'on se déplace, la distance entre nous et le pokémon sauvage est suffisamment courte, alors le pokémon s'affiche, sinon on continue de se déplacer.

. Le problème lors de l'affichage du pokémon avec pygame, est qu'il n'est pas possible de le "désafficher" simplement après l'avoir vaincu, nous avons essayé dans un premier temps de passer la portion de la map se trouvant sous le pokémon au premier plan pour "faire disparaître" le pokémon mais comme les coordonnées ont été modifiées pour correspondre à l'échelle de la carte, cette méthode était trop fastidieuse. Nous avons donc simplement rendu transparents le pokémon et son bouton de combat associé après la victoire, ce qui techniquement n'empêche pas le joueur de cliquer à nouveau sur le bouton pour relancer le combat.

. La taille de l'interface de combat n'est pas la même que celle de la fenêtre. (Choix)

<u>Bugs encore présents/ non corrigés :</u>

. Dans le "main" : Lorsque le combat se ferme après la défaite, la victoire ou la fuite, la map ne répond plus. Le problème vient sûrement d'un conflit entre la fenêtre de combat réalisée avec Qt et la fenêtre de la map réalisée avec pygame. On a essayé plusieurs choses, comme ouvrir la fenêtre de combat Qt via pygame dans le "main", ou encore de lancer les deux fenêtres séparément, quitte à ce qu'il soit encore possible de se déplacer sur la map alors que le combat est en cours, mais rien n'y fait, le bug persiste. La seule option aurait été de refaire la fenêtre de la carte avec Qt mais le temps était insuffisant.

. Dans "Menu" : nous avons le même problème avec le menu, c'est pour ça qu'on l'a séparé du "main" et qu'on le fait tourner à part. Avec le menu, la map ne répond plus avant même qu'on lance un combat. La raison est sûrement la même que précédemment, un conflit entre la fenêtre du menu réalisé avec Qt et celle de la map, ou alors simplement que la superposition de fenêtres est mal gérée par pygame.

Il y a de plus une erreur dans le fichier main.py : les pokémons ne doivent être rendus transparents que si le joueur a gagné ıe combat, et la ligne de code le permettant est donc mal indentée.