2012-2013

ULB

Filipchuk Nazar;Ethan Zuckeberg;Rui Vilela Neves

[RAPPORT PROJET INFO H200 - POKÉMON.DOCX]

Le rapport sur le développement du jeu Pokémon sur java

L’Introduction:

Pour le projet du cours d’Informatique en BA2 cette année nous avions un très grand choix de thème pour pouvoir exercer nos connaissances en informatique.

Notre groupe a choisi le projet #1 qui consiste à écrire un projet java qui permettra de jouer au jeu Pokémon. Un personnage que le joueur peut déplacer sur une carte et qui affronte des adversaires lors de combat munis de Pokémon.

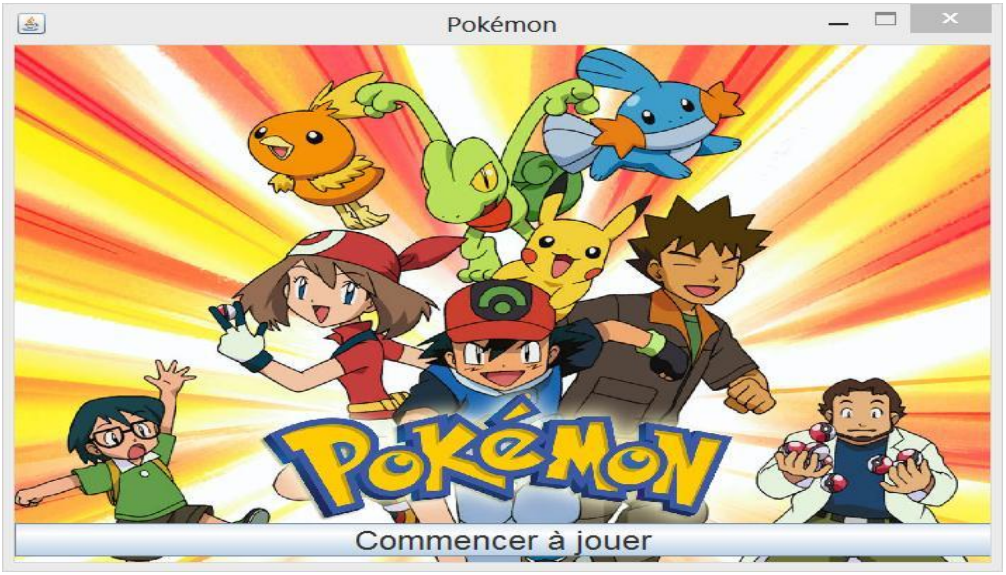
Cette décision a été prise car ce projet est proche des concepts vus pendant les travaux pratiques.

Le programme que nous avons réalisé respecte au maximum le concept de MVC utilisé en programmation java OO, c’est à dire que nous disposons de trois packages: “modèle, vue et contrôleur” qui ont chacun une tâche bien spécifique.

Le premier package, Modèle, représente le corps de notre programme et effectue les calculs, il représente le jeu en lui-même. Le deuxième package, appelé Vue, affiche à l’utilisateur toute l’interface graphique du programme et affiche donc sur l'écran ce qui se passe dans modelé. Le dernier package, Contrôleur, assure l’interaction entre le programme et l’utilisateur. Il prend en charge la gestion des événements et met à jour la vue ou le modèle et les synchronise. Il reçoit donc tous les événements de l'utilisateur et enclenche les actions à effectuer.

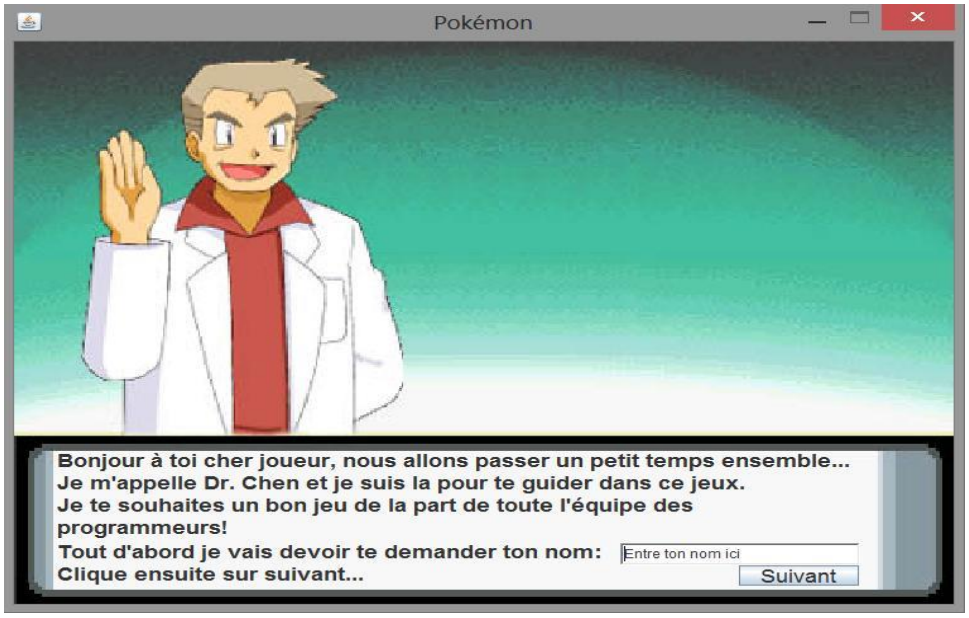
Le Fonctionnement du Jeu:

Ce chapitre explique à partir de la fonction *main* qui se trouve dans la classe TheGame, le fonctionnement du programme.
Main nous avance vers la package ***vue***, avec lancement de la classe Fenetre() qui ouvre la fenêtre demandant à l'utilisateur de commencer le jeu en cliquant sur <Commencer à jouer>.

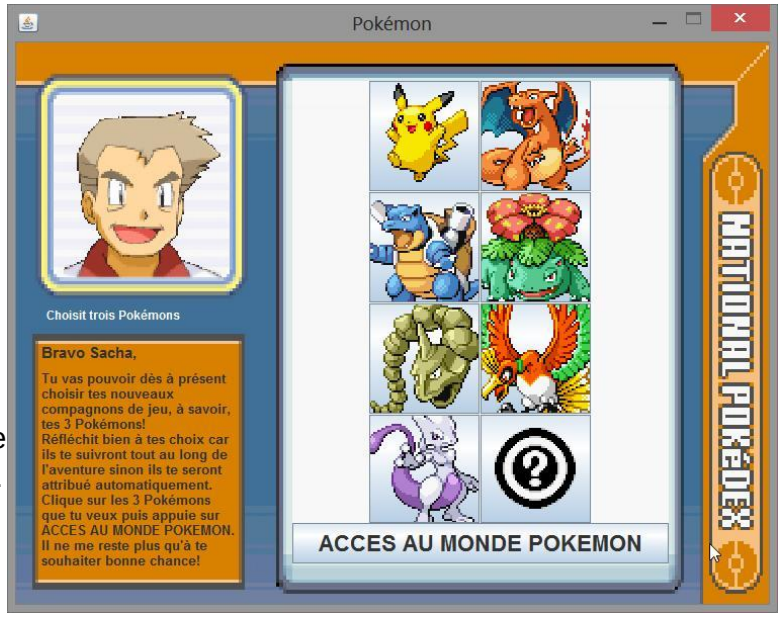


Ce dernier bouton fait appel à la classe Fenetrenom() qui nous amène à la prochaine étape du jeu qui est la demande au joueur de son nom.
Si le joueur n''entre pas de nom, le nom "Sacha" lui sera attribué automatiquement.

En appuyant sur <Suivant>, la classe Fenpoke() est appellée.



Elle permet au joueur de choisir trois Pokémons parmi une liste de sept Pokémons différents. Ainsi que le bouton hasard qui lui fournit un Pokémon parmi les sept.
Les Pokémons choisit sont afficher dans un JLabel sur la gauche, JLabel choix est donc mis à jour des qu'un Pokémon est choisit.



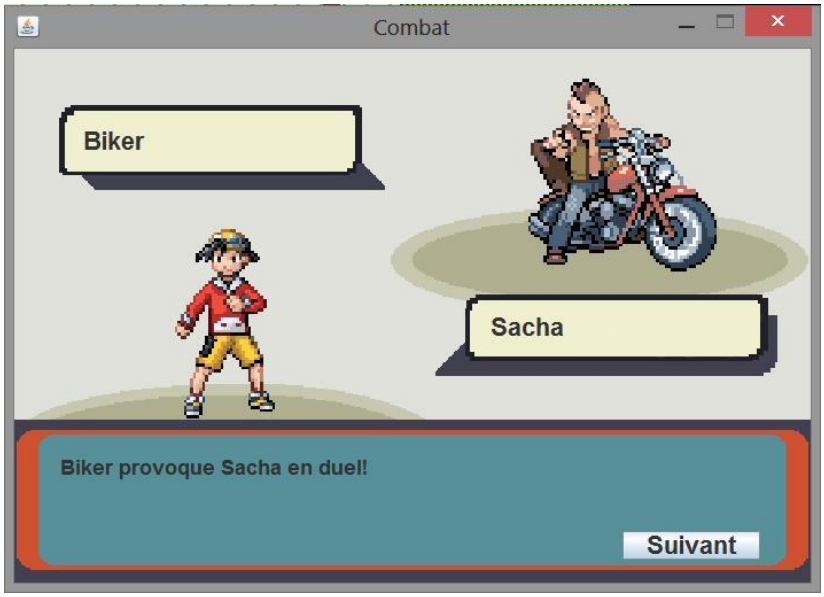
List <String> animal est la liste des Pokémons choisi et est limitée afin que l'on ne puisse en choisir plus que trois. Néanmoins, si le joueur oublie de sélectionner des Pokémons, ils lui seront attribués automatiquement.

Lorsque le joueur clique sur le bouton <ACCES AU MONDE POKEMON> la classe Carte() est appelée.

La classe Carte() nous avance vers le package ***model*** ou la classe *DataCarte()* va generer les trois cartes du jeux, placer les 6 adversaires sur les cartes et générer leur Pokémons *generate_opponents(),* qui est régénéré à chaque fois après le combat pour pouvoir combattre de nouveau si le joueur le souhaites, et crée l'inventaire pour le joueur *generate_inventory()* .
readTXT() va lire les noms des Pokémons et les attribuer leur statistiques(pv, degats, etc) à partir des deux fichiers texte (*pokeFail* et *stats*) et sauvegarder le tout dans une *HashMap*.

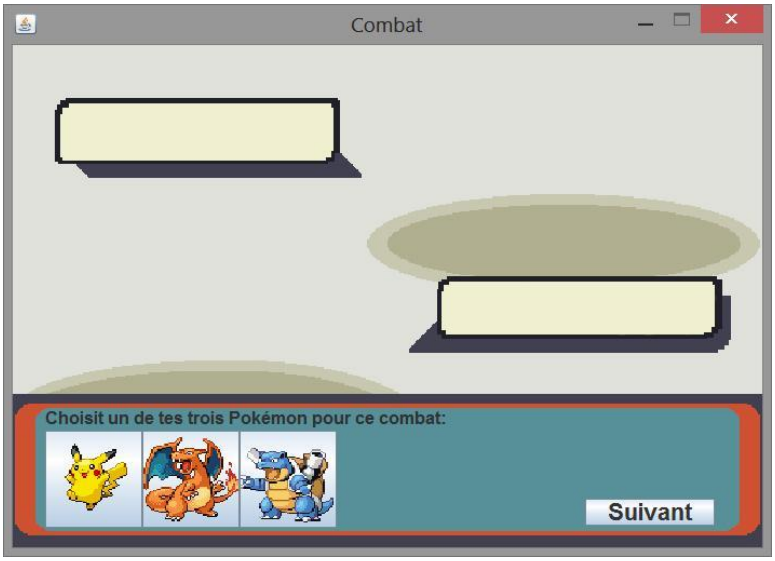
Carte() nous connecte également au **controleur** en définissant un objet de la classe TAdapter(). La méthode *movePlayer()* va permettre au joueur de se déplacer sur la carte. Si le joueur appuie sur les touches de son clavier ce qui est gérer par le **controleur**.

Quant à Bord(), cette classe recevra la DataCarte() et affiche tout ce qui se passe sur la carte. Elle modifie l'image du joueur en fonction de son déplacement. Elle affiche les images des adversaires et affiche les bonnes images de la carte selon le numéro de la carte dans laquelle il se trouve. Pour pouvoir afficher les données de DataCarte() un utilise la méthode run(objet DataCarte()).



De même cette classe fait appel au dialogue devant l'adversaire qui demande au joueur s'il veut entamer un nouveau combat. Ensuite en fonction du choix de joueur le combat est lancé, avec la classe Confrontation(Map(avec les actions), List d'adversaires, Liste d'inventaire), ou permet de rester sur la carte et de continuer à se déplacer.
La classe Confrontation() affiche l'image des deux joueurs et un texte expliquant que l'adversaire provoque le joueur en duel.

En cliquant sur <suivant>, nous faisons appel à la classe Fentrechoix() qui ouvre une fenêtre offrant la possibilité de choisir parmi ses trois Pokémons choisit au départ pour mener le combat face aux Pokémons adverses. Une fois un des trois Pokémon choisit et seulement lorsqu'un Pokémon est choisi le fait de cliquer sur le bouton <suivant> va démarrer l'interface de combat et le combat dans le **modèle** également..



Tout d'abord, nous allons créer le Pokémon choisi par le joueur. Pour ce faire nous faisons appel à la classe creatPokemon(Map_NomeDuPokemon) qui va générer le Pokémon avec toutes les méthodes descendues de l'interface IPokemon.

Ensuite, la classe ICombat(pokemonJoueur, pokemonsEnemies, Inventaire) demare le combat en lui-même dans la partie du modele. *List<IAction> getPossibleMoves(turns, skill_used)* est une liste qui rassemble tous les actions possible autant attaques que potions pour le joueur, en tenant compte du tour qui se passe maintenant pour pouvoir réutiliser l'attaque avec des dégâts spéciales du Pokémon. Méthode *IGameState nextState* qui avance d'un tour c'est à dire qu'il permet au joueur d'attaquer et puis à l'adversaire d'attaquer le Pokémon du joueur. Il permet toutes les modifications de chaque Pokémon (vie, nombre de potion disponible).

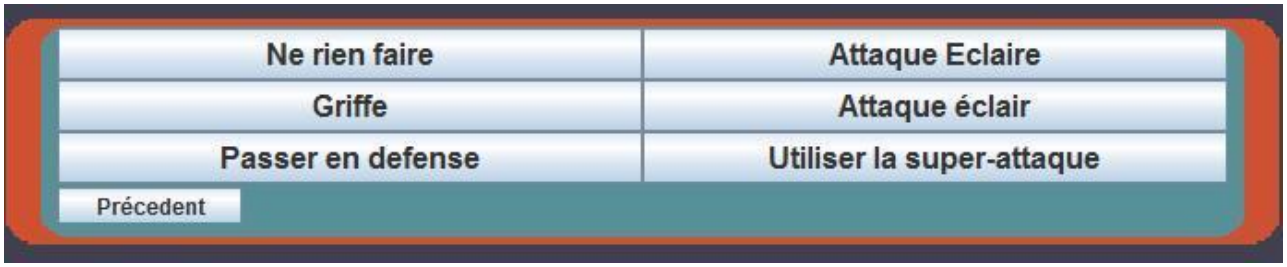
Par après on fait appel à la classe ICombatTerminal() contient tout ce qui concerne la représentation graphique du combat qui sera expliqué plus apres. On la relie au IGameState par la méthode *run()* .

La classe ICombatTerminal() débute avec la fonction *run(IGameState())* et l'importation des Pokémons adverses. La fonction *affichagefixe()* comprend tous les affichages qui ne changent pas tout au long du combat comme le nom et l'image du joueur et de son Pokémon ainsi que l'image de l'adversaire.

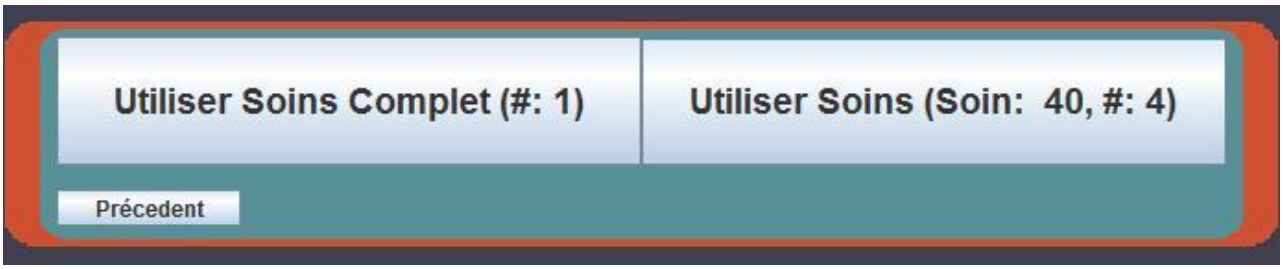
La fonction *lancementpokemonadv(IGameState())* va afficher l'image et le nom du Pokémon adverse à partir du moment où l'adversaire lance un nouveau Pokémon. C'est à dire au début du jeu et à chaque fois qu'un Pokémon est tué.
En cliquant sur <suivant>, on fait appel à la fonction *running(IGameState())* qui va afficher la vie des Pokémon dans leur barre de vie au moyen de la fonction *affichagevie(IGameState game).*

Cette première fonction *run* fera appel à deux fonctions différentes selon si le combat est fini ou pas. Il fera appel à la fonction *affichagechoix (IGameState,List<IAction>)* si le jeux continue et fera appel à la fonction *affichagefinal(IGameState)* si le combat est fini.

La fonction *affichagechoix ()* lancera la fonction *affichageattaque())* si le joueur choisit d'attaquer. Il affichera toutes les attaques que le Pokémon peut utiliser.



La fonction *affichagepotion(,List<IAction>)* est appelée si le joueur choisit d'utiliser une potion.



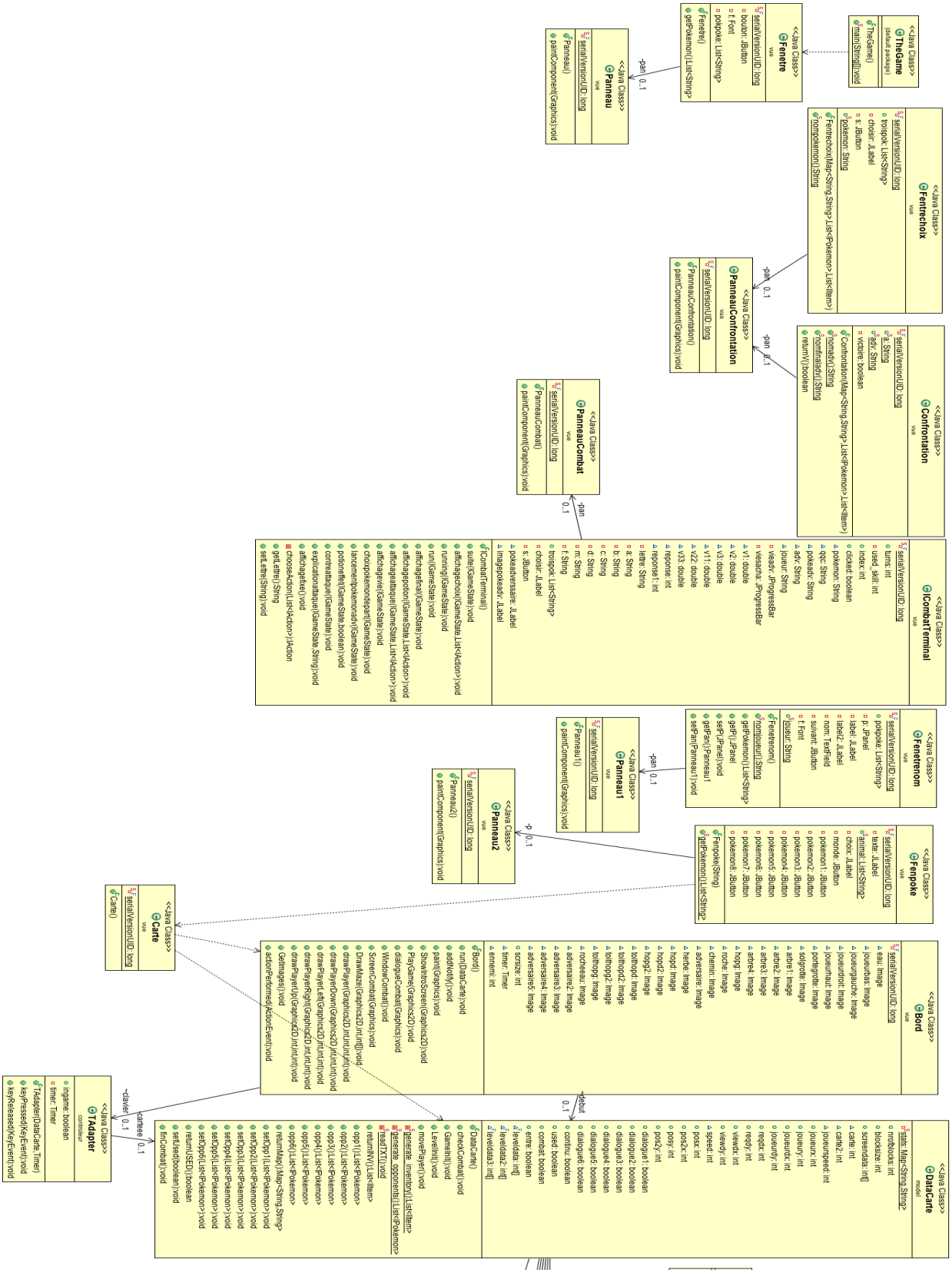
En fonction de la potion choisie ou de l'attaque, le numéro de l''index qui représente les représente dans la liste d'action changera. En cliquant sur une attaque, nous faisons appel à la fonction *explicationattaque()* qui affiche quel attaque le joueur à lancé sur l'adversaire et qui lance *suite().* Cette dernière fonction correspond à l'attaque du Pokémon du joueur adverse. Il crée une action *chooseAction(IGameState.getPossibleMoves(turns,used_skill))* en fonction de l'index et passe au stade suivant via IGameState*.nextState().*

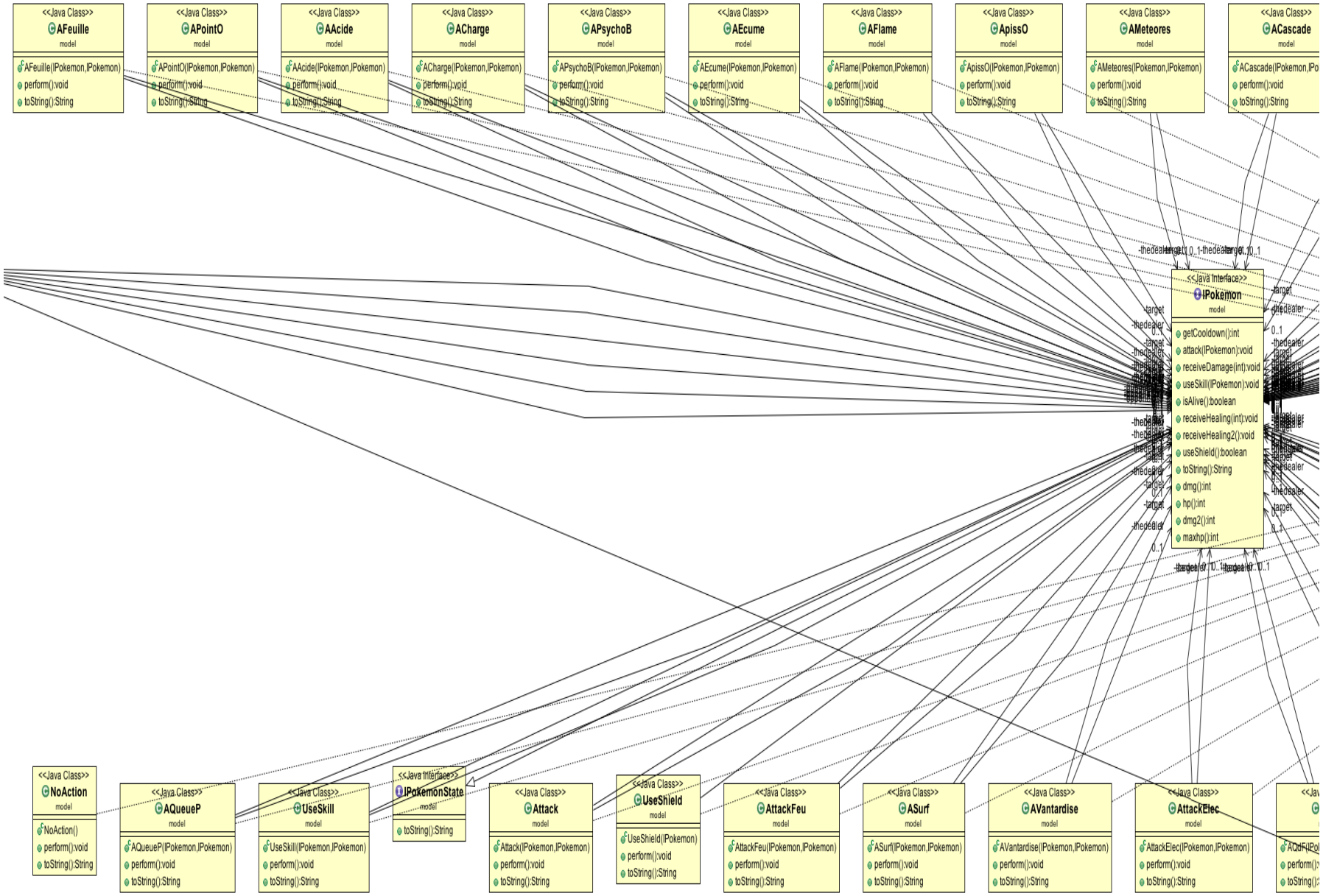
Dans la fonction *ICombat nextState(IAction)* l'action choisie est effectuée avec la methode *perform()* et le Pokémon adverse contre-attaque avec via la fonction *fight ()*.

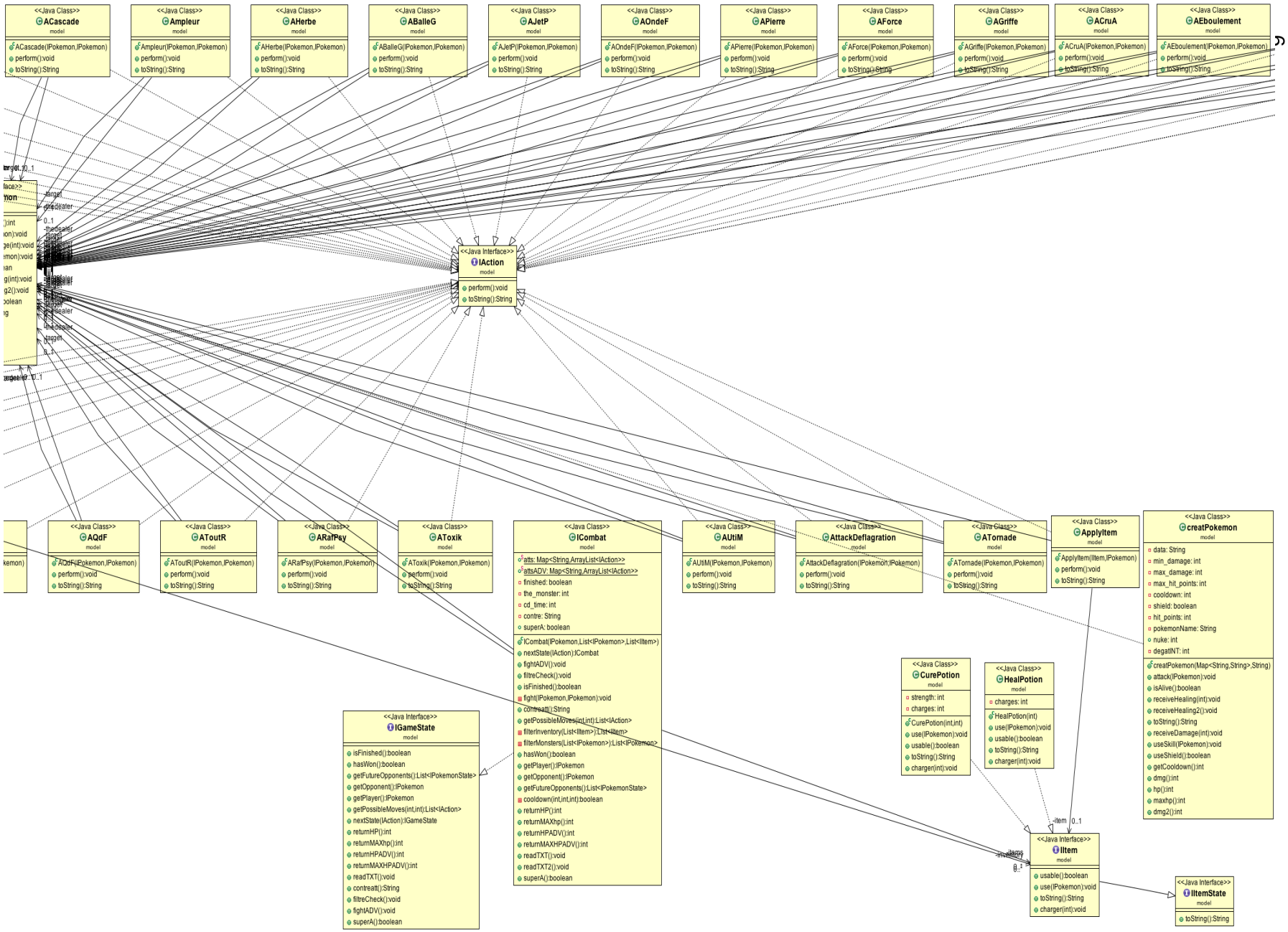
Pour lancer le tour suivant on fait appel à *running()* qui recommence la boucle du combat jusqu'à ce que celui-ci soit finit.

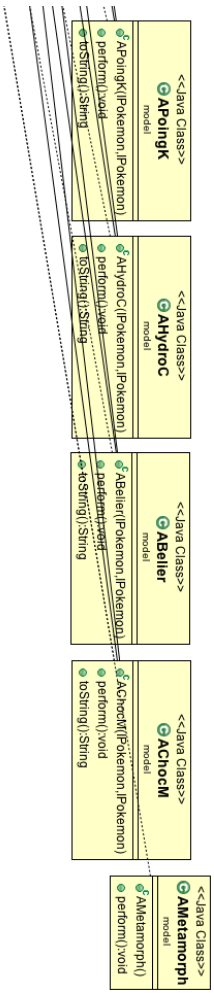
Si le joueur a gagné le combat il est rémunéré des potions qui lui seront utiles pour combattre le reste d'adversaires. Ensuite la fenêtre est fermée et le joueur est renvoyé sur la carte afin de mener un autre combat lorsqu'il entrera dans le champ de vision d'un adversaire.

Pour une description plus détaillé du fonctionnement du programme, description des paramètres qui doivent être passé et l'interaction entre les classes vous pouvait lire le diagramme UML du code :




<<Java Class>>
AFeuille
model
AFeuille(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
APointO
model
APointO(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
AAcide
model
AAcide(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
ACharge
model
ACharge(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
APsychoB
model
APsychoB(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
AEcume
model
AEcume(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
AFlame
model
AFlame(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
ApissO
model
ApissO(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
AMeteores
model
AMeteores(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
ACascade
model
ACascade(IPokemon,IPo
perform():void
toString():String
<<Java Interface>>
IPokemon
model
getCooldown():int
attack(IPokemon):void
receiveDamage(int):void
useSkill(IPokemon):void
isAlive():boolean
receiveHealing(int):void
receiveHealing2():void
useShield():boolean
toString():String
dmg():int
hp():int
dmg2():int
maxhp():int
<<Java Class>>
NoAction
model
NoAction()
perform():void
toString():String
<<Java Class>>
AQueueP
model
AQueueP(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
UseSkill
model
UseSkill(IPokemon,IPokemon)
perform():void
toString():String
<<Java Interface>>
IPokemonState
model
toString():String
<<Java Class>>
Attack
model
Attack(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
UseShield
model
UseShield(IPokemon)
perform():void
toString():String
<<Java Class>>
AttackFeu
model
AttackFeu(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
ASurf
model
ASurf(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
AVantardise
model
AVantardise(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
AttackElec
model
AttackElec(IPokemon,IPokemon)
perform():void
toString():String


<<Java Class>>
ACascade
model
ACascade(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
Ampleur
model
Ampleur(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
AHerbe
model
AHerbe(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
ABalleG
model
ABalleG(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
AJetP
model
AJetP(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
AOndeF
model
AOndeF(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
APierre
model
APierre(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
AForce
model
AForce(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
AGriffe
model
AGriffe(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
ACruA
model
ACruA(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
AEboulement
model
AEboulement(IPokemon,IPokemon)
perform():void
toString():String
<<Java Interface>>
IAction
model
perform():void
toString():String
target
predealer
0..1
<<Java Class>>
AQdF
model
AQdF(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
AToutR
model
AToutR(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
ARalPsy
model
ARalPsy(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
AToxik
model
AToxik(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
ICombat
model
atts: Map<String,ArrayList<IAction>>
attsADV: Map<String,ArrayList<IAction>>
finished: boolean
the_monster: int
cd_time: int
contre: String
superA: boolean
ICombat(IPokemon,List<IPokemon>,List<IItem>)
nextState(IAction):ICombat
fightADV():void
filtreCheck():void
isFinished():boolean
fight(IPokemon,IPokemon):void
contreatt():String
getPossibleMoves(int,int):List<IAction>
filterInventory(List<IItem>):List<IItem>
filterMonsters(List<IPokemon>):List<IPokemon>
hasWon():boolean
getPlayer():IPokemon
getOpponent():IPokemon
getFutureOpponents():List<IPokemonState>
cooldown(int,int,int):boolean
returnHP():int
returnMAXhp():int
returnHPADV():int
returnMAXHPADV():int
readTXT():void
readTXT2():void
superA():boolean
<<Java Class>>
AUltiM
model
AUltiM(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
AttackDeflagration
model
AttackDeflagration(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
ATornade
model
ATornade(IPokemon,IPokemon)
perform():void
toString():String
<<Java Class>>
ApplyItem
model
ApplyItem(IItem,IPokemon)
perform():void
toString():String
<<Java Class>>
creatPokemon
model
data: String
min_damage: int
max_damage: int
max_hit_points: int
cooldown: int
shield: boolean
hit_points: int
pokemonName: String
nuke: int
degatINT: int
creatPokemon(Map<String,String>,String)
attack(IPokemon):void
isAlive():boolean
receiveHealing(int):void
receiveHealing2():void
toString():String
receiveDamage(int):void
useSkill(IPokemon):void
useShield():boolean
getCooldown():int
dmg():int
hp():int
maxhp():int
dmg2():int
<<Java Class>>
CurePotion
model
strength: int
charges: int
CurePotion(int,int)
use(IPokemon):void
usable():boolean
toString():String
charger(int):void
<<Java Class>>
HealPotion
model
charges: int
HealPotion(int)
use(IPokemon):void
usable():boolean
toString():String
charger(int):void
<<Java Interface>>
IGameState
model
isFinished():boolean
hasWon():boolean
getFutureOpponents():List<IPokemonState>
getOpponent():IPokemon
getPlayer():IPokemon
getPossibleMoves(int,int):List<IAction>
nextState(IAction):IGameState
returnHP():int
returnMAXhp():int
returnHPADV():int
returnMAXHPADV():int
readTXT():void
contreatt():String
filtreCheck():void
fightADV():void
superA():boolean
<<Java Interface>>
IItem
model
usable():boolean
use(IPokemon):void
toString():String
charger(int):void
-item
0..1
<<Java Interface>>
IItemState
model
toString():String



On a utilisé la programmation orientée objet pour mieux organiser notre code et le rendre facilement modifiable. Dans cette optique, nous avons créé la classe reatPokemon() qui en est le parfait exemple. Chaque Pokémon est créé grâce à cette classe, donc on peut ajouter autant de Pokémons qu'on veut. De plus les Pokémons peuvent être complétement différents, les données de base sont lues dans un fichier .txt qui peuvent être modifié facilement, les attaques peuvent être complètement différents, car les attaques de Pokémons sont générer dans une liste et ressemblées dans une Map pour un accès facile. Donc les attaques peuvent été ajouter complétement séparément du Pokémon et librement. La fonctionnalité des attaques de même peut être facilement modifié est être très distincte, on a par exemple Attaque de vol de vie chez Nosferapti, Attaque qui permet de se soigné appelé Ampleur, attaque qui permet de se défendre et réduire de beaucoup les dégâts qui seront infligé au prochaine tour UseShield, attaque UseSKill qui est la super attaque avec un temps de réutilisation. De même les potions peuvent être très diverses. Toutes les attaques sont ressemblé dans une interface IAction, mais nous pouvons de même crées plusieurs interface à partir de l'IAction avec les types d'attaques différents (dégâts, défense, heal, vol de vie, dégâts pendant un temps etc) il faudrait juste avoir plus d'imaginations pour les attaques et du temps pour tout ecrire.

De même la programmation OO nous a permis de respecter les consignes d'utiliser le concept de MVC.