Projet long Java

Adonis Najimi
Louis Qiu
Luca Rochereau
Joseph Mechali
Taha Yassine Elleuch

PLAN

I- Présentation du jeu SmashBros2D:

Principes généraux du jeu

SmashBros2D est un jeu de combat en vue de côté semblable à la célèbre licence de Nintendo Super Smash Bros. Contrairement aux jeux de combat 2D classiques (par exemple Street Fighter), l'objectif n'est pas de réduire la vie de son adversaire à 0 sur un niveau plat, mais à l'expulser d'une aire de jeu constituée de diverses plateformes. Le système de point de vie est remplacé par un système de pourcentages qui croissent à chaque coup reçu et augmentent la sensibilité du personnage aux coups reçus.

Contrôles

Mouvements:

Un personnage peut de déplacer vers la droite ou la gauche ou encore sauter lorsqu'il se trouve sur une plateforme. Lorsqu'il est en l'air, il peut aussi effectuer un saut à condition qu'il n'en ait exécuté qu'un seul avant. En plus de cela il peut légèrement de déplacer vers la gauche ou la droite mais de façon marquée que lorsqu'il marche sur une plateforme.

Attaques:

Chaque personnage dispose d'une palette de coups qui lui est propre. Elle est toujours constituée de 4 coups normaux et de 4 coups spéciaux. Ces coups sont effectués par combinaison entre une direction et la touche d'attaque normale ou spéciale. Certaines de ces attaques peuvent influencer directement la vitesse du personnage tels que les dashs ou les coups B+Haut qui sont l'équivalent d'un saut supplémentaire.

Les personnages du jeu:

Mai

A -> coup de poing
B -> lancer d'éventail
A + cote -> coup de pied moyenne portée diagonal
B + cote -> roulade dash
A + bas -> coup de robe
B + bas -> strip stun
A + haut -> coup pied vertical
B + haut -> salto arrière

Rie

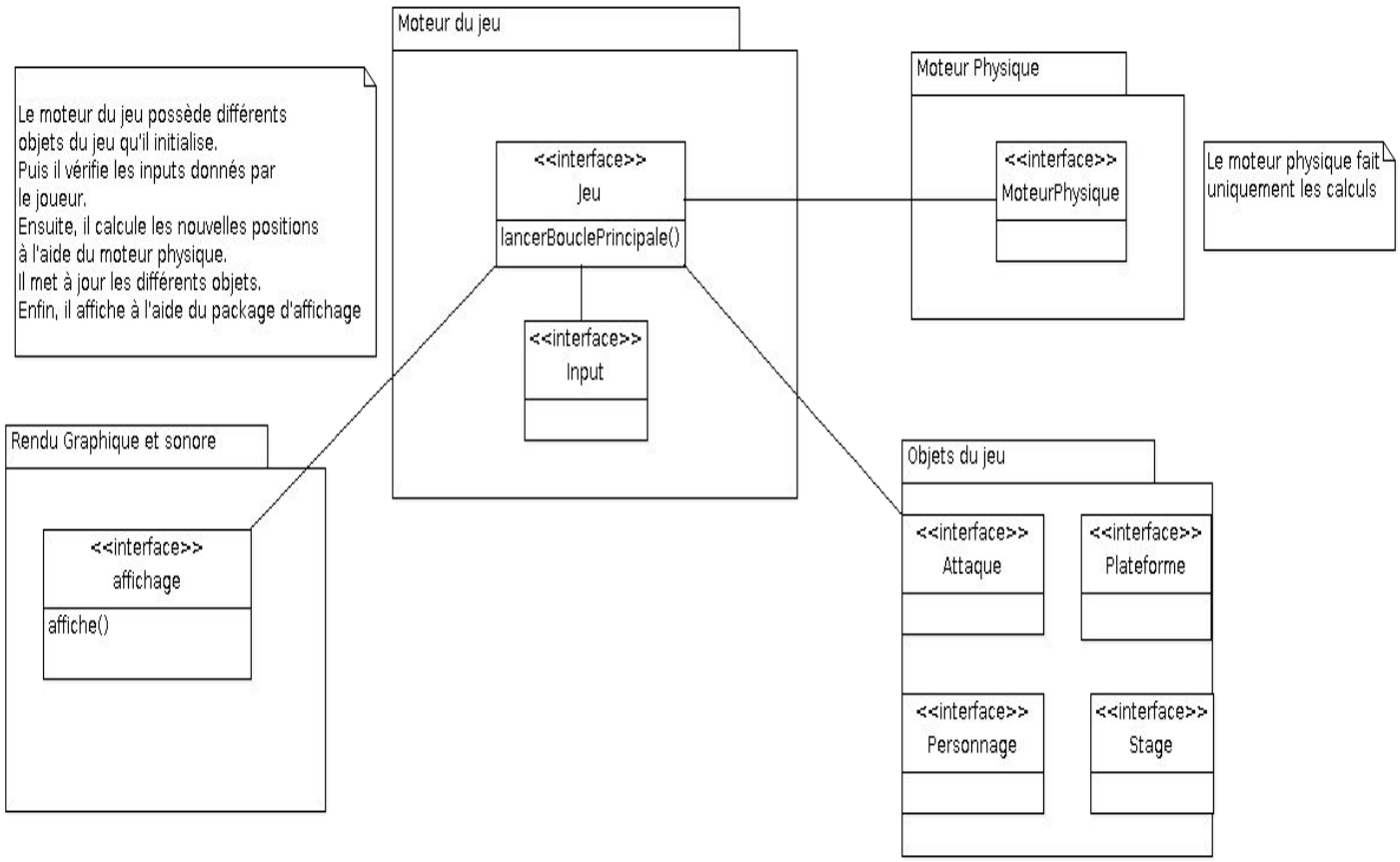
A -> coup de violoncelle
B -> boule de feu
A + cote -> gros coup de son arme
B + cote -> dash avant avec son arme
A + bas -> coup de pied bas
B + bas -> force field
A + haut -> gros coup vers le haut
B + haut -> saut avec son arme

III- Conception de l'application

1- Séparation en packages

Nous avons découpé la structure du jeu de la manière suivante:

-Moteur du jeu
-Moteur physique
-Objets du jeu
-Rendu Graphique et Sonore



2- Les classes importantes

2.1 Gestion des états des personnages :

Description des états des personnages:

L'une des principales mécaniques utilisées pour gérer les personnages est un tableau d'état décrivant la situation dans laquelle se trouve le personnage, ce qui détermine les actions qu'il peut à cet instant effectuer. Dans la version finale du jeu, il y en a 8 mais il est aisé d'en ajouter afin d'affiner encore la description de l'état général du personnage. Par exemple, un état « EN_RECOVERY » a été rajouté assez tard, en quelques minutes, pour satisfaire aux problèmes d'animations d'attaques produisant des projectiles.

Etat « SUR_PLATEFORME »

Cet état est égal à 1 lorsque que le personnage se trouve sur une plateforme et à 0 sinon. Il permet de déterminer la liberté de déplacement latérale et la réinitialisation du nombre de sauts possibles. En effet quand un personnage est sur une plateforme il pourra se déplacer plus vite vers la droite et la gauche que lorsqu'il est dans les airs. De plus, il n'est alors plus soumis à la gravité dui fait qu'il ne s'enfonce pas dans la plateforme, ni ne chute vers elle en permanence.

Etat « STUN »

Un personnage en état de STUN n'est plus capable d'effectuer la moindre action (ni mouvement, ni attaque). L'entier associé à cet état détermine la durée en frame durant laquelle le personnage reste dans cet état. Il peut être induit par une collision à haute vitesse avec une plateforme ou par certaines attaques.

Etat « SAUT_EFFECTUE»

Cet état compte le nombre de sauts effectués d'affilée sans avoir atterri au sol. Un personnage ne pouvant effectuer que 2 sauts d'affilée, ce compteur s'incrémente après chaque saut et interdit un saut s'il est égal à 2. Il se remet à 0 quand le personnage est sur une plateforme. L'attaque spéciale haut des personnages permet souvent d'effectuer une sorte de saut supplémentaire. L'exécution d'un tel saut incrémente directement à 2 le compteur car aucun saut n'est possible après.

Etat « SUBIT_ATTAQUE »

Un personnage subissant une attaque n'est pas capable d'attaquer et dispose d'une capacité de déplacement limitée. L'entier associé à cet état détermine la durée en frame durant laquelle il se trouve dans cet état. Il est induit, comme son nom l'indique, lorsque le personnage est touché par une attaque. La durée dépend de l'attaque que le personnage a subit.

Etat « EN_ATTAQUE »

Comme son nom l'indique, un personnage est « EN_ATTAQUE » lorsqu'il est en train d'attaquer. Dans cet état, il ne peut ni bouger, ni se déplacer. Ici aussi, l'entier associé détermine la durée durant laquelle le personnage va rester dans cet état. C'est l'attaque lancée qui va déterminer la durée initiale de mise dans cet état.

Etat « INVULNERABLE »

Lorsqu'un personnage est invulnérable il n'est plus sensible aux attaques de son adversaire. Le seul moment où un personnage est dans cet état est après être réapparu, après une mort.

Etat « EN_MOUVEMENT »

Cet état signale que le personnage est en mouvement quand il est égal à 1, 0 sinon. Il est surtout présent afin d'effectuer un rendu graphique correct lorsqu'un personnage bouge.

Etat « EN_RECOVERY »

Dans cet état, un personnage ne peut pas attaquer mais se déplacer. Il est induit par une attaque lancée par le personnage. Ainsi un personnage ne peut pas « spammer » des attaques sans arrêt et il permet aussi un affichage propre des attaques à projectiles.

2.2 Gestion affichage

L'affichage se fait grâce à la classe vue.
Cette classe contient une liste d'afficheurs qui définissent chacun une méthode afficher.
Cette méthode permet de bien séparer la vue du modèle et d'avoir la définition de l'affichage dans la vue et non dans le modèle.
Ainsi, la vue est un JPanel qui redéfinit paintComponent. La redéfinition consistant à parcourir la liste des afficheurs et de les afficher.

De plus, le moteur de jeu ajoute les éléments dans l'ordre de priorité, ceci permet d'afficher les éléments par dessus les uns des autres pour avoir un affichage convenable.
En effet, l'affichage se rafraîchit à chaque frame quand le moteur de jeu appelle la méthode render de la vue. Il faut donc que les éléments s'affichent dans le bon ordre.
Nous avons donc établi une précondition sur l'ordre des éléments de la liste d'afficheurs, nous supposons tout le temps que cette liste est bien ordonnée.
Par exemple, le stage est ajouté en premier, puis les plateformes et ensuite les personnages.

2.3 Moteur physique

Le moteur physique a pour but de gérer les phénomènes physiques des éléments du jeu.

Pour le moteur de jeu, il se comporte comme une boîte noire, prenant en paramètres le jeu et la durée d'une frame et renvoyant une liste des collisions ayant eu lieu durant la frame en cours.

De plus, le moteur déplace les objets à déplacer selon leur vitesse et les collisions, il modifie aussi les vitesses des personnages afin de prendre en compte un coefficient de frottement, la gravité et les collisions. La seule méthode du moteur physique est majPhysique(Jeu jeu) (et le constructeur).

La séquence de fonctionnement de majPhysique est la suivante :

Pour chaque attaque obtenue via jeu.getAttaques(), mettre à jour l'attaque et la liste des collisions via majAttaque pour toutes les attaques, tous les personnages et toutes les plateformes du jeu.

Puis pour chaque personnage obtenu via jeu.getPersonnages(), mettra à jour le personnage et la liste des collisions via malPersonnage pour tous les personnage et toutes les plateformes du jeu.

Les plateformes (obtenus via jeu.getPlateformes) ne sont pas directement traitées par majPhysique, étant donné que l'on a choisi de ne pas considérer les collisions plateforme/plateforme et que tous les autres cas de collision sont traités précédemment.

La séquence de fonctionnement de majAttaque est la suivante :

Appeler la méthode evoluer() de l'attaque et si et seulement si celle-ci renvoie true, traiter le déplacement de l'attaque (utile pour les projectiles, les rebonds, etc …).

Tester pour chaque élément du jeu si il y a collision avec l'attaque (en ignorant l'attaque elle-même et les doublons). Si il y a collision, on ajoute la collision à la liste, puis on appelle majCollision si il s'agit d'une collision attaque/personnage ou attaque/plateforme.

La séquence de fonctionnement de MajPersonnage est la suivante :

La méthode commence par vérifier si la vitesse du personnage n'est pas trop elevé (limite fixée de telle sorte qu'un seul déplacement ne puisse pas dépasser la moitié de la taille du collider du personnage dans une direction ou une autre), si celle-ci est trop élevé, on divise la durée sur laquelle le calcul est effectué par 2 jusqu'à ce que la distance parcourue soit suffisamment faible (ou que le nombre de calculs engendrés soit trop grand).

Ensuite on déplace autant de fois que nécessaire le personnage et on vérifie et prend en compte ses collisions avec les autres personnages et les plateformes (ajout à la liste des collisions et appelle de majCollision). Le coefficient de frottement n'est prit en compte qu'au dernier calcul et les attaques sont ignorées car déjà testées précédemment.

La méthode majCollision peut-être appellée avec plusieurs types de paramètres :

majCollision(Personnage,Personnage,Collision) pour une collision Personnage/Personnage : cette méthode sert à pousser un personnage immobile avec un personnage en mouvement.

majCollision(Personnage,Plateforme,Collision) pour une collision Personnage/Plateforme

Cette méthode sert à gérer la collision entre un personnage et une plateforme, elle se sert de la méthode collideSide pour obtenir le face percutée et ainsi replacer le personnage à l'extérieur de la plateforme, modifier se vitesse et son état si besoin.

majCollision(Attaque,Personnage,Collision) pour une collision Attaque/Personnage

Cette méthode gère l'impact physique d'une attaque sur un personnage, elle affecte d'abord le personnage via la méthode affecter(Personnage) de l'attaque, selon un schéma fixé par l'attaque (immobilisation, projection forcée, etc …) .

majCollision(Attaque,Plateforme,Collision) pour une collision Attaque/Plateforme

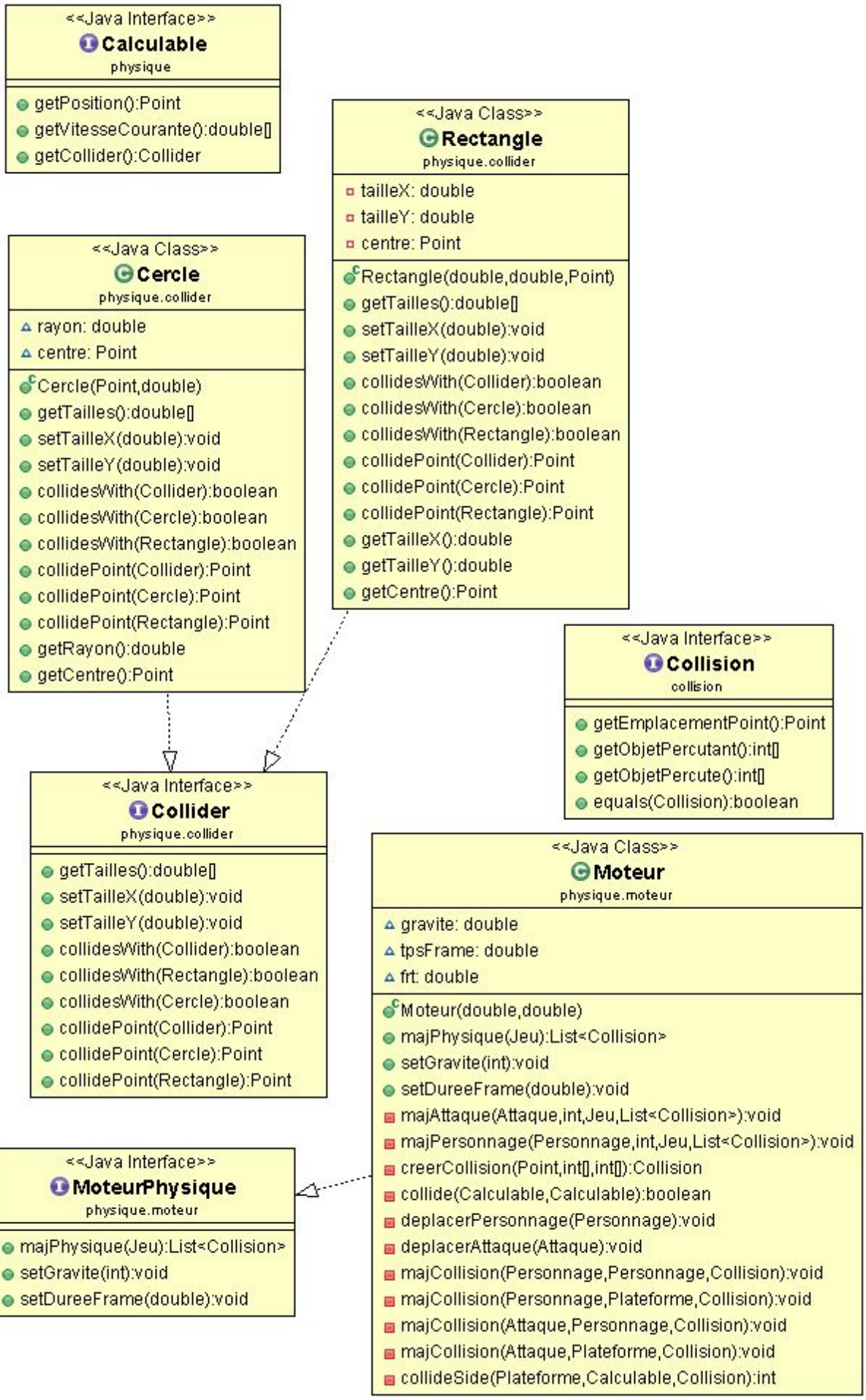
Méthode permettant le rebond d'une attaque sur une plateforme en fonction de la valeur retournée par collideSide.

Collider :

Le sous-package collider fournit les hitbox au jeu et au moteur physique. Il s'agit des objets ayant une réalité physique. Le sous-package fournit un ensemble de méthodes permettant de détecter les collisions, calculer les points d'impact et modifier les caractéristiques géométriques des colliders (centres, tailles). Il fournit 2 types de collider : Cercle et Rectangle.

Collision :

Le type collision sert à représenter la collision entre 2 objets. Une collision est composée de deux couples d'indices, un pour l'objet Percutant, l'autre pour l'objet Percute, et d'un point d'impact. Les couples d'indices représentent le type du Calculable via le premier indice : 0 pour une Attaque, 1 pour un Personnage, 2 pour une Plateforme ; et le deuxième indice est l'indice de l'objet dans la liste correspondant a son type dans le jeu.


<<Java Interface>>
Calculable
physique
getPosition():Point
getVitesseCourante():double[]
getCollider():Collider
<<Java Class>>
Rectangle
physique.collider
tailleX: double
tailleY: double
centre: Point
Rectangle(double,double,Point)
getTailles():double[]
setTailleX(double):void
setTailleY(double):void
collidesWith(Collider):boolean
collidesWith(Cercle):boolean
collidesWith(Rectangle):boolean
collidePoint(Collider):Point
collidePoint(Cercle):Point
collidePoint(Rectangle):Point
getTailleX():double
getTailleY():double
getCentre():Point
<<Java Class>>
Cercle
physique.collider
rayon: double
centre: Point
Cercle(Point,double)
getTailles():double[]
setTailleX(double):void
setTailleY(double):void
collidesWith(Collider):boolean
collidesWith(Cercle):boolean
collidesWith(Rectangle):boolean
collidePoint(Collider):Point
collidePoint(Cercle):Point
collidePoint(Rectangle):Point
getRayon():double
getCentre():Point
<<Java Interface>>
Collision
collision
getEmplacementPoint():Point
getObjetPercutant():int[]
getObjetPercute():int[]
equals(Collision):boolean
<<Java Interface>>
Collider
physique.collider
getTailles():double[]
setTailleX(double):void
setTailleY(double):void
collidesWith(Collider):boolean
collidesWith(Rectangle):boolean
collidesWith(Cercle):boolean
collidePoint(Collider):Point
collidePoint(Cercle):Point
collidePoint(Rectangle):Point
<<Java Class>>
Moteur
physique.moteur
gravite: double
tpsFrame: double
frt: double
Moteur(double,double)
majPhysique(Jeu):List<Collision>
setGravite(int):void
setDureeFrame(double):void
majAttaque(Attaque,int,Jeu,List<Collision>):void
majPersonnage(Personnage,int,Jeu,List<Collision>):void
creerCollision(Point,int[],int[]):Collision
collide(Calculable,Calculable):boolean
deplacerPersonnage(Personnage):void
deplacerAttaque(Attaque):void
majCollision(Personnage,Personnage,Collision):void
majCollision(Personnage,Plateforme,Collision):void
majCollision(Attaque,Personnage,Collision):void
majCollision(Attaque,Plateforme,Collision):void
collideSide(Plateforme,Calculable,Collision):int
<<Java Interface>>
MoteurPhysique
physique.moteur
majPhysique(Jeu):List<Collision>
setGravite(int):void
setDureeFrame(double):void

2.4 Moteur de jeu

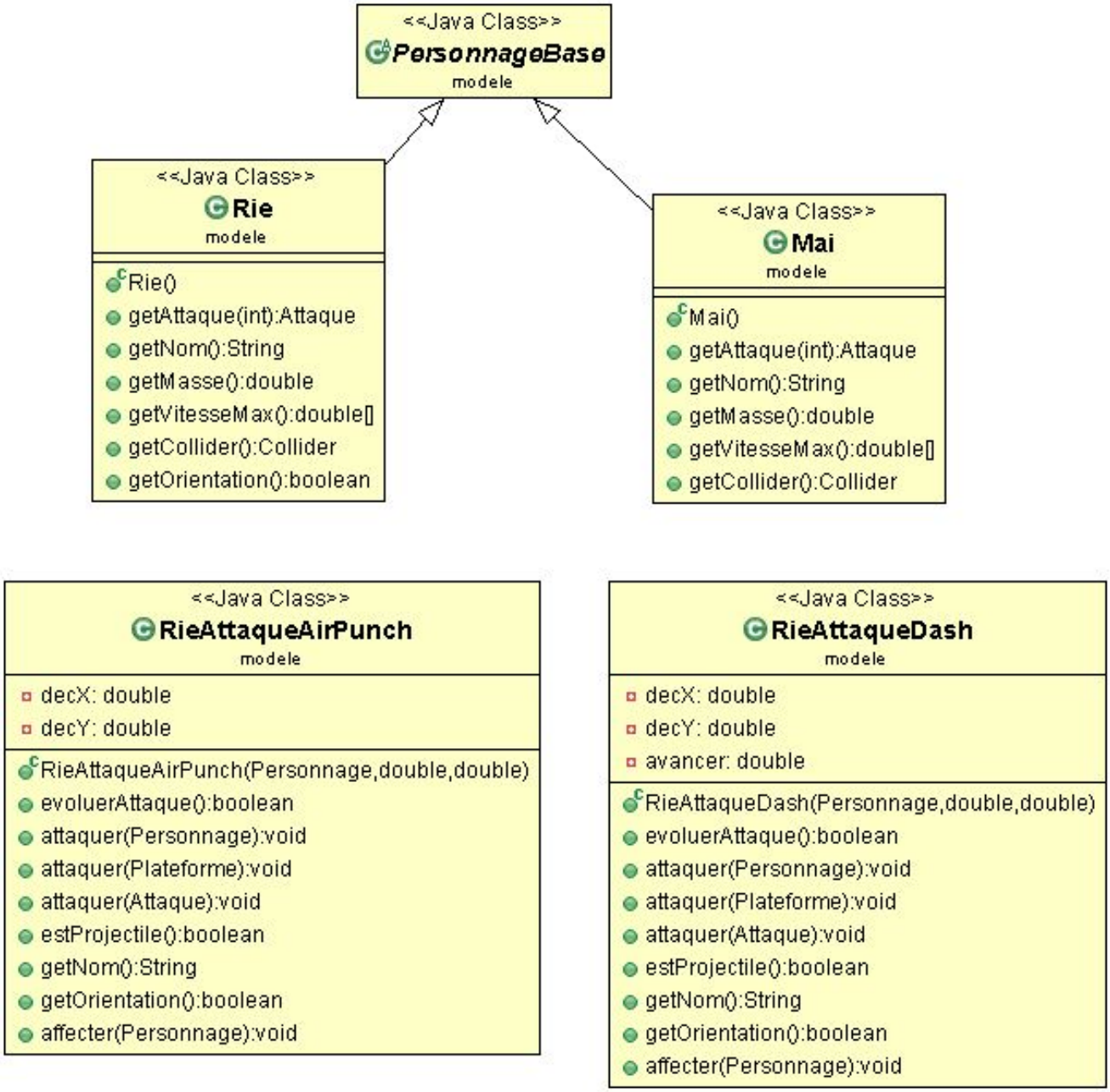
Le moteur permet d'effectuer des actions à chaque frame.
C'est le coeur du programme, il utilise une vue, un moteur physique, un clavier, un jeu et des joueurs.
Il permet de faire le lien entre toutes les classes du programme.
A chaque tour de boucle il vérifie si des touches sont appuyées, puis il execute les actions correspondantes. Ensuite, il appelle le calcul du moteur physique, le moteur physique lui donne une liste de collisions. Avec ces collisions, il applique les règles du jeu.
Enfin, il met à jour l'affichage.
Ce fonctionnement est détaillé dans la figure du diagramme de séquences du moteur de jeu.
C'est aussi lui qui choisit quand ajouter des éléments à afficher ou quand jouer du son.

Le moteur de jeu a la plupart de ses méthodes protected, ceci permet de pouvoir facilement créer un nouveau moteur de jeu à partir de celui déjà fait.

3- Facilité à ajouter du contenu

Créer un nouveau personnage et ses attaques

La création d'un nouveau personnage se fait par la création d'une nouvelle classe héritant de la classe abstraite PersonnageBase. Il faut pour cela implémenter les méthodes suivantes :
- **getAttaques**, cette méthode fait correspondre un code d'attaque (A, A + bas, B etc etc) à une attaque, elle détermine donc le set d'attaque du personnage
- **getNom**, elle retourne le nom du personnage, il faut le faire correspondre à la Sprite Sheet associée au personnage qu'on créera dans la classe SpriteSheet
- **getMasse**, la masse du personnage qui détermine la manière dont le personnage va tomber dans le jeu
- **getVitesseMax**, retourne la vitesse maximum du personnage
- **getCollider**, détermine la hit-box du personnage



Rie et Mai sont des personnages implantés, Rie possède plusieurs attaques dont RieAttaqueAirPunch et RieAttaqueDash

On peut utiliser des attaques déjà implantées ou créer de nouvelles attaques pour le personnage. La création d'une nouvelle attaque se fait en créant une classe héritant de la classe abstraite AttaqueBase. Les caractéristiques intrinsèques d'une attaque sont les suivantes :
- son propriétaire, le personnage qui a lancé l'attaque
- les dommages, le nombre de pourcent infligés à l'adversaire touché
- la force de l'attaque, la capacité de projection de l'adversaire touché
- sa durée de vie, la durée de l'attaque en frames
- la durée de cooldown qu'elle inflige à son utilisateur
- le fait qu'une attaque soit un projectile ou pas

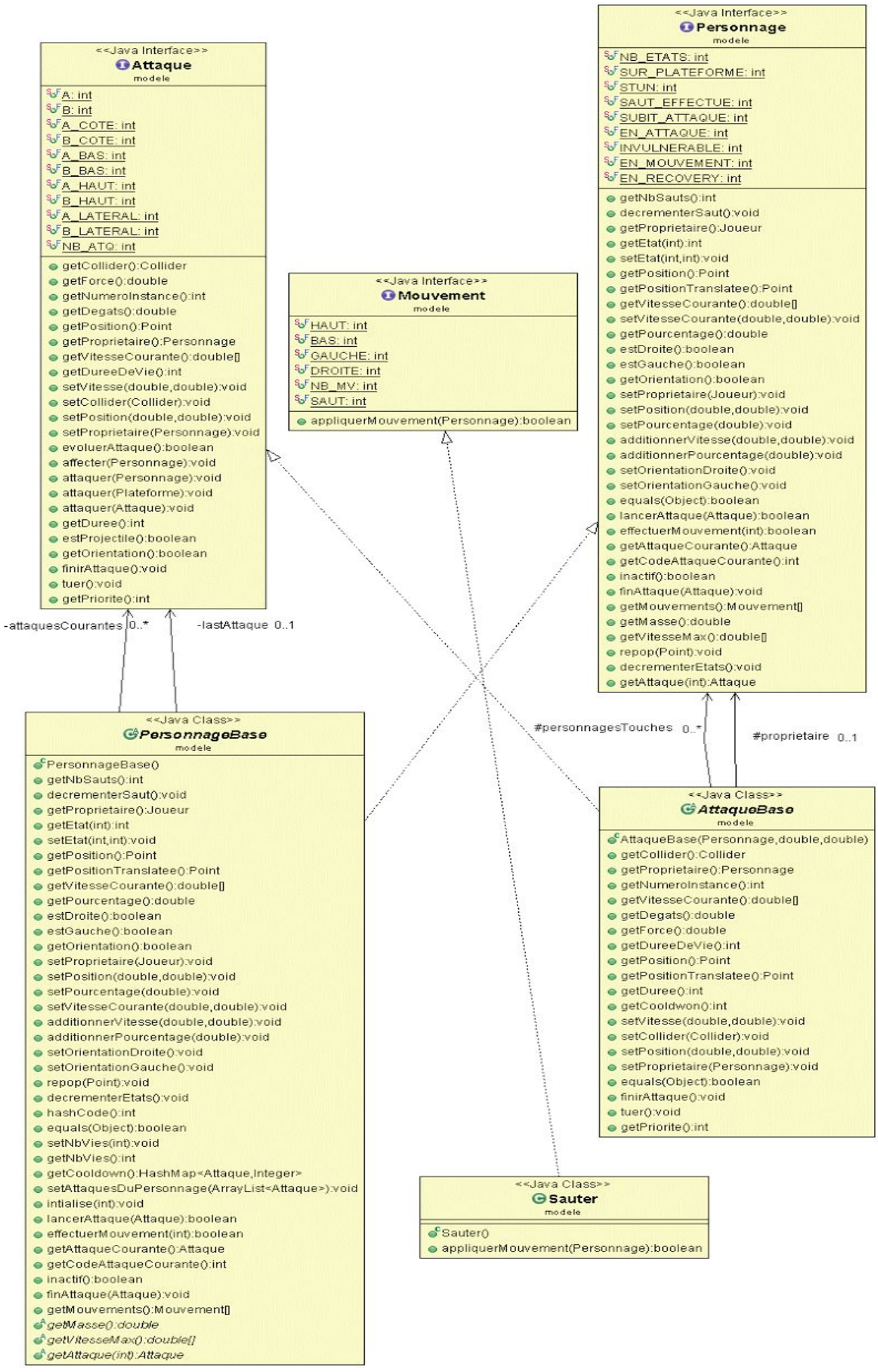
En plus de ces caractéristiques intrinsèques, les caractéristiques physiques sont aussi à ajuster :
- le collider de l'attaque, c'est la hit-box de l'attaque qui peut être de taille variable au cours de l'attaque
- la vitesse du collider
- la position du collider

L'évolution de l'attaque doit être décidée dans la méthode évoluerAttaque. Elle se fait par modification des caractéristiques physiques en fonction de la durée de vie restante de l'attaque.

Ainsi, un coup de poing sera décomposé en plusieurs phases, une première durant laquelle le collider est réduit à 0 car le coup ne porte pas réellement puis ensuite une phase durant laquelle le collider apparait et avance. Enfin, il repart en arrière et disparaît.

Il faut ensuite implémenter les sprites de l'attaque si c'est une attaque projectile. Pour cela, il faut l'ajouter à la classe SpriteSheet.

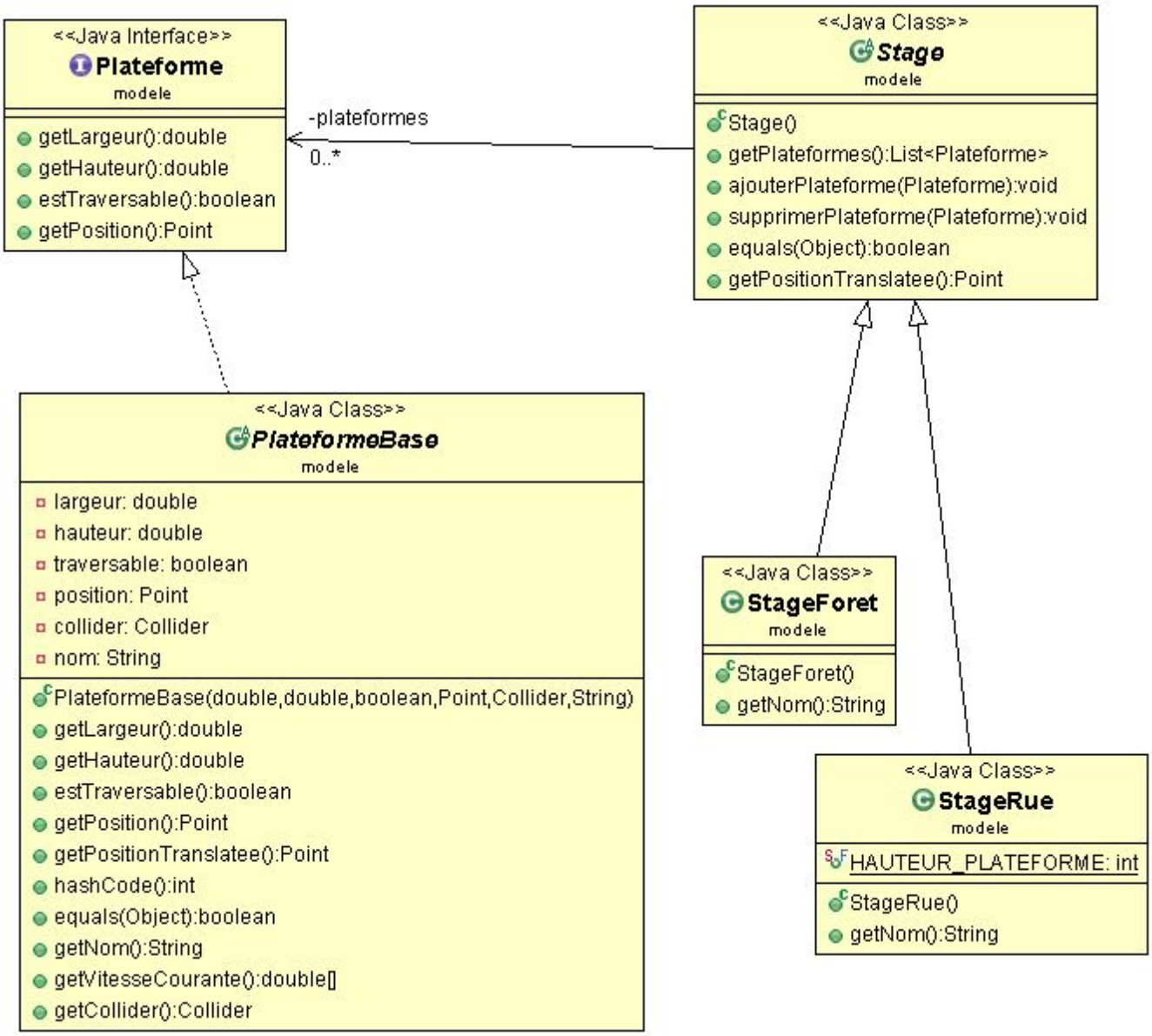
Il faut aussi inclure toute la partie graphique du personnage. Elle consiste en une Sprite Sheet de 13 lignes de sprites. Chacune correspond à une action différente choisie selon la convention suivante :
1)stun
2)saut
3)subit attaque
4)inactif
5)en mouvement
6)attaque A
7)attaque B
8)attaque A+côté
9)attaque B+côté
10)attaque A+bas
11)attaque B+bas
12)attaque A+haut
13)attaque B+haut
Les sprites doivent toutes faire la même taille.
Il faut l'ajouter aux sprites utilisés par le jeu dans la classe SpriteSheet.


<<Java Interface>>
Attaque
modele
A: int
B: int
A_COTE: int
B_COTE: int
A_BAS: int
B_BAS: int
A_HAUT: int
B_HAUT: int
A_LATERAL: int
B_LATERAL: int
NB_ATQ: int
getCollider():Collider
getForce():double
getNumeroInstance():int
getDegats():double
getPosition():Point
getProprietaire():Personnage
getVitesseCourante():double[]
getDureeDeVie():int
setVitesse(double,double):void
setCollider(Collider):void
setPosition(double,double):void
setProprietaire(Personnage):void
evoluerAttaque():boolean
affecter(Personnage):void
attaquer(Personnage):void
attaquer(Plateforme):void
attaquer(Attaque):void
getDuree():int
estProjectile():boolean
getOrientation():boolean
finirAttaque():void
tuer():void
getPriorite():int
<<Java Interface>>
Mouvement
modele
HAUT: int
BAS: int
GAUCHE: int
DROITE: int
NB_MV: int
SAUT: int
appliquerMouvement(Personnage):boolean
<<Java Interface>>
Personnage
modele
NB_ETATS: int
SUR_PLATEFORME: int
STUN: int
SAUT_EFFECTUE: int
SUBIT_ATTAQUE: int
EN_ATTAQUE: int
INVULNERABLE: int
EN_MOUVEMENT: int
EN_RECOVERY: int
getNbSauts():int
decrementerSaut():void
getProprietaire():Joueur
getEtat(int):int
setEtat(int,int):void
getPosition():Point
getPositionTranslatee():Point
getVitesseCourante():double[]
setVitesseCourante(double,double):void
getPourcentage():double
estDroite():boolean
estGauche():boolean
getOrientation():boolean
setProprietaire(Joueur):void
setPosition(double,double):void
setPourcentage(double):void
additionnerVitesse(double,double):void
additionnerPourcentage(double):void
setOrientationDroite():void
setOrientationGauche():void
equals(Object):boolean
lancerAttaque(Attaque):boolean
effectuerMouvement(int):boolean
getAttaqueCourante():Attaque
getCodeAttaqueCourante():int
inactif():boolean
finAttaque(Attaque):void
getMouvements():Mouvement[]
getMasse():double
getVitesseMax():double[]
repop(Point):void
decrementerEtats():void
getAttaque(int):Attaque
-attaquesCourantes 0..*
-lastAttaque 0..1
#personnagesTouches 0..*
#proprietaire 0..1
<<Java Class>>
PersonnageBase
modele
PersonnageBase()
getNbSauts():int
decrementerSaut():void
getProprietaire():Joueur
getEtat(int):int
setEtat(int,int):void
getPosition():Point
getPositionTranslatee():Point
getVitesseCourante():double[]
getPourcentage():double
estDroite():boolean
estGauche():boolean
getOrientation():boolean
setProprietaire(Joueur):void
setPosition(double,double):void
setPourcentage(double):void
setVitesseCourante(double,double):void
additionnerVitesse(double,double):void
additionnerPourcentage(double):void
setOrientationDroite():void
setOrientationGauche():void
repop(Point):void
decrementerEtats():void
hashCode():int
equals(Object):boolean
setNbVies(int):void
getNbVies():int
getCooldown():HashMap<Attaque,Integer>
setAttaquesDuPersonnage(ArrayList<Attaque>):void
intialise(int):void
lancerAttaque(Attaque):boolean
effectuerMouvement(int):boolean
getAttaqueCourante():Attaque
getCodeAttaqueCourante():int
inactif():boolean
finAttaque(Attaque):void
getMouvements():Mouvement[]
getMasse():double
getVitesseMax():double[]
getAttaque(int):Attaque
<<Java Class>>
AttaqueBase
modele
AttaqueBase(Personnage,double,double)
getCollider():Collider
getProprietaire():Personnage
getNumeroInstance():int
getVitesseCourante():double[]
getDegats():double
getForce():double
getDureeDeVie():int
getPosition():Point
getPositionTranslatee():Point
getDuree():int
getCooldwon():int
setVitesse(double,double):void
setCollider(Collider):void
setPosition(double,double):void
setProprietaire(Personnage):void
equals(Object):boolean
finirAttaque():void
tuer():void
getPriorite():int
<<Java Class>>
Sauter
modele
Sauter()
appliquerMouvement(Personnage):boolean

Création de plates-formes et de stages

La création de plates-formes se fait aussi par héritage de la classe PlateformeBase. Il suffit d'utiliser le constructeur de la classe mère dont les arguments sont la taille de la plate-forme, un booléen qui détermine si la plate-forme est traversable, le collider et son nom.

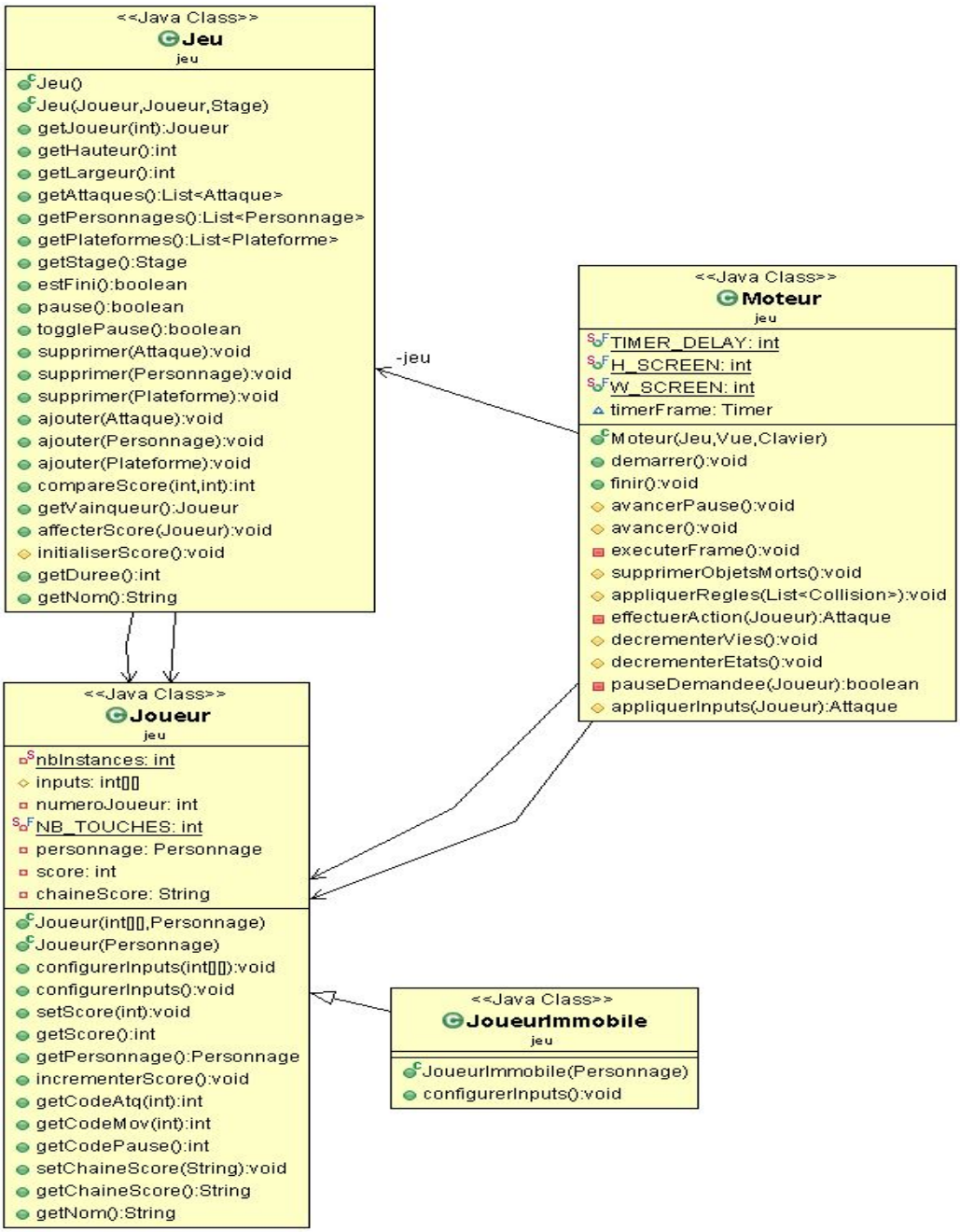
Pour le stage, il faut hériter de la classe Stage. Dans le constructeur, il faut ajouter les différentes plates-formes du stage. Il suffit enfin d'implémenter la méthode getNom avec le nom correspondant à l'image de fond qu'on ajoutera dans la classe SpriteSheet.



4- Facilité à étendre certaines classes

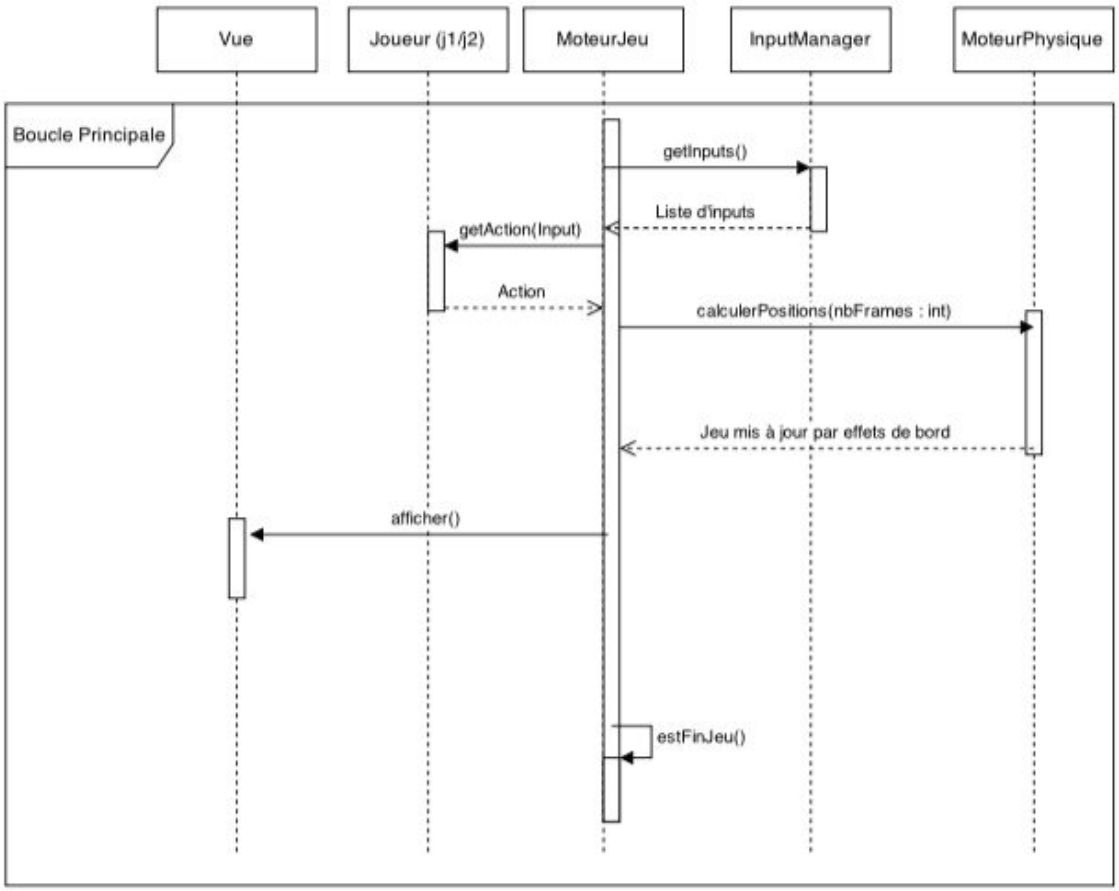
Le mode de jeu dépend du jeu donné au moteur de jeu.
Ainsi, on peut facilement hériter un jeu et modifier les méthodes affecter(joueur) et finJeu pour avoir un nouveau mode de jeu.

De plus, le moteur de jeu est aussi modifiable et nous pouvons par exemple faire un nouveau moteur en redéfinissant la méthode “executerFrame” qui consiste à executer un tour de boucle (une frame).



IV- Diagramme de séquence du moteur de jeu:

Voici le diagramme de séquences que nous avions fourni lors de la conception



Ce diagramme de séquence représente les principales intéractions avec les autres classes.
Nous avons effectué très peu de modifications.
L'inputManager a été remplacé par la classe Clavier et la classe Action n'est plus utilisée.
La classe action n'était pas tout à fait adaptée car elle devait tout connaitre sur le jeu pour pouvoir s'executer. Elle aurait fait le travail du moteur de jeu.
Donc nous n'avons plus d'actions mais une méthode appliquerInputs dans le moteur de jeu qui symbolise la réflexion par action.

V- Répartition du travail

Pendant une première phase durant laquelle aucun affichage n'était réalisé, les tâches étaient réparties ainsi :
Adonis -> moteur de jeu et contrôles
Joseph -> affichage (vue)
Taha -> éléments du jeu (modèle)
Luca et Louis -> moteur physique (modèle)

Ensuite, une fois l'affichage basique réalisé et le moteur physique terminé, la répartition a changé :
Adonis -> vue, contrôles et moteur de jeu
Joseph -> vue et menus
Taha -> son et objets du jeu
Luca -> perfectionnement de la physique et attaques des personnages
Louis -> sprites et attaques des personnages

VI- Organisation du groupe

Afin de communiquer sur les problèmes rencontrés régulièrement, un groupe Facebook a été créé. Les membres ont ainsi aussi pu rapporté de manière régulière leur avancée. Ce groupe a permis de réagir efficacement aux problèmes liés à la structure complexe du projet et de rapporter immédiatement les bugs détectés par chacun à celui qui était responsable de la partie non fonctionnelle et lui permettre de la corriger rapidement.

Conclusion :

Ce projet illustre les avantages du travail de groupe, mais également ses problématiques, comme la séparation des tâches et l'utilisation conjointe (et délicate) du SVN et d'Eclipse. De plus, notre coordination s'appuyant sur les contrats des différentes méthodes, nous avons pu constater l'importance de la documentation. En Effet, il était impensable de regarder le code de chaque méthode pour savoir précisément ce qu'elle fait. Malgré ces difficultés techniques, le résultat est esthétique et fonctionnel, dont seules demeurent des problématiques internes au jeu (attaques équilibrées, sauts à la bonne hauteur, ...). Tous les aspects de la création du jeu, du son à la physique simulée, ont été abordés, ce qui a grandement amélioré nos connaissances en conception, en code et bibliothèques java.

Nous avons ainsi achevé notre premier jeu en temps réel, réactif et beau, ce qui est extrêmement gratifiant.