Compte-rendu du jeu Meteor Survive

Proposé par **Pierre Bourdiaux** (G1) et **Thomas Devienne** (G2)

dans le cadre du projet tutoré de JavaFX
à l'IUT de Clermont-Ferrand.

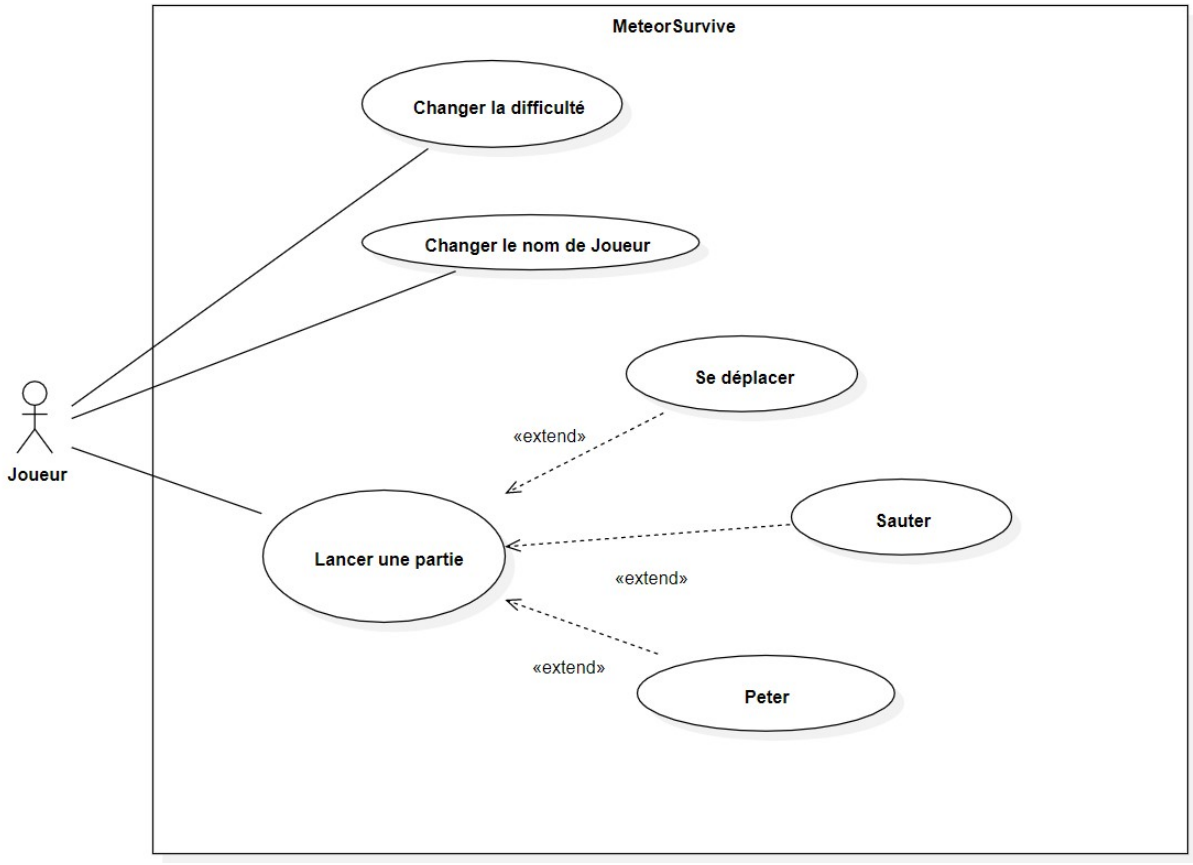
Sommaire

Contexte

Le but du jeu est de faire survivre un dinosaure de la chute de météorites et donc de faire survivre sont espèce. Le jeu sera en 2D avec un dinosaure qui se déplace sur l'axe horizontal. Des météorites tomberons de plus en plus vite et de plus en plus nombreuses au fil du temps. A chaque fois que le dinosaure heurte une météorite il perd de la vie mais il peut en regagner grâce au bonus qui apparaîtrons (gains de vie, augmentation de la hauteur du saut, regains de pets). Le jeu pourra mémoriser les scores et avoir un tableau des meilleurs score établis. Le dinosaure peut péter pour faire exploser les météorites et se défendre. Quand les météorites explosent, elles peuvent libérer des items pour regagner de la vie ou sauter plus haut. Le joueur peut enregistrer son nom lorsqu'il joue et l'associer à son score.

Diagramme des uses cases

Le diagramme de use case de notre projet est le suivant :



Nom	Changer la difficulté
Objectif	Pourvoir changer la difficulté du jeu

Acteur principal	Joueur
Condition initiale	Être sur le menu principal
Scénario d'utilisation	Le joueur appuie sur le bouton prévu à cet effet
Condition de fin	Lancement de la partie

Nom	Changer le nom du joueur
Objectif	Pourvoir changer le nom du joueur
Acteur principal	Joueur
Condition initiale	Être sur le menu principal
Scénario d'utilisation	Le joueur rentre dans le TextField son nom / pseudo
Condition de fin	Lancement de la partie

Nom	Lancer la partie
Objectif	Lancer une partie
Acteur principal	Joueur
Condition initiale	Être sur le menu principal
Scénario d'utilisation	Le joueur appuie sur le bouton prévu à cet effet
Condition de fin	Lancement de la partie

Nom	Se déplacer
Objectif	Déplacer le Dinosaure
Acteur principal	Joueur
Condition initiale	Avoir une partie de lancé
Scénario d'utilisation	Le joueur presse les touche prévue (Q et LEFT

	pour la gauche, D et RIGHT pour la droite)
Condition de fin	Fin de la partie

Nom	Sauter
Objectif	Faire sauter le Dinosaure
Acteur principal	Joueur
Condition initiale	Avoir une partie de lancé
Scénario d'utilisation	Le joueur presse la touche SPACE
Condition de fin	Fin de la partie

Nom	Peter
Objectif	Faire Peter le Dinosaure
Acteur principal	Joueur
Condition initiale	Avoir une partie de lancé
Scénario d'utilisation	Le joueur presse la S / DOWN
Condition de fin	Fin de la partie

Diagramme de package

Le diagramme de package de notre projet est le suivant :

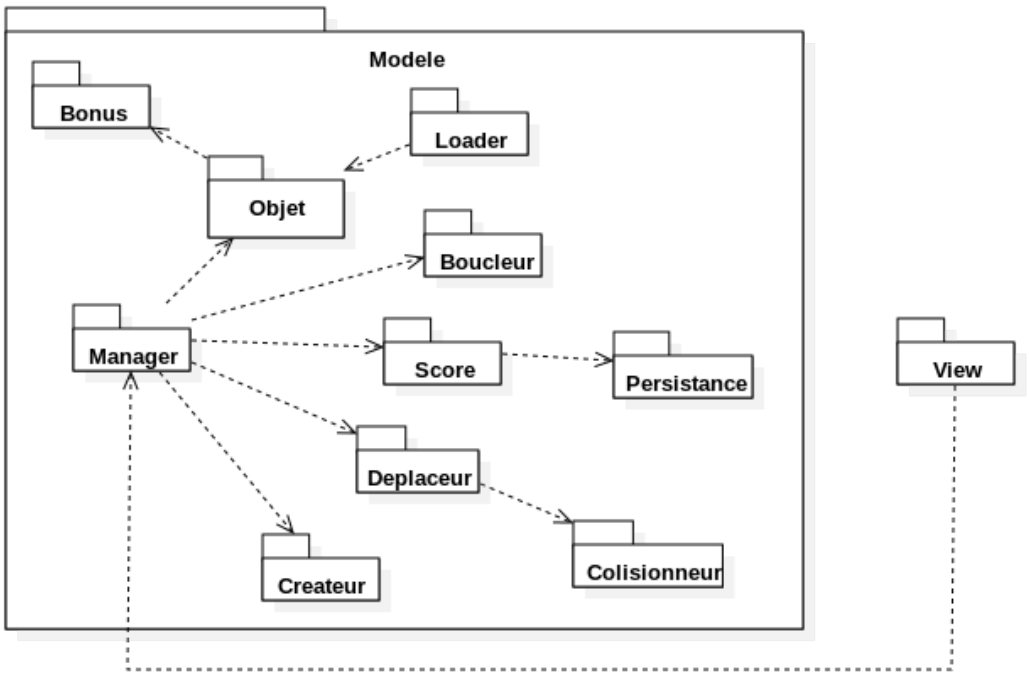


Image 1 : Diagramme de package du projet

Chaque package possède, comme chaque classe, une responsabilité qui lui est propre. Ainsi on a :

- *Manager :* fournir une interface unique pour manipuler et gérer des instances d'objets du modèle ;
- *Objet :* regroupe toutes les classes qui représente des objets ;
- *Bonus :* regroupe des interfaces qui fournissent un modèle commun aux bonus ;
- *Loader :* regroupe le loader (le point de démarrage du programme) ;
- *Persistance :* gère la persistance du programme, c'est-à-dire la sauvegarde et le chargement des données ;
- *Score :* regroupe les classes qui concernent les scores utilisateur ;
- *Déplaceur :* regroupe toutes les classes dont le rôle est de déplacer des objets ;
- *Collisionneur :* regroupe toutes les classes dont le rôle est de vérifier la collision entre deux objets ;
- *Boucleur :* classes chargées de boucler de manière periodique toutes les x ms ;
- *Créateur :* regroupe toutes les classes dont le rôle est de créer des objets ;

Nous avons choisi la classe Manager (du package Manager) pour gérer et manipuler les instances d'objets du modèle depuis le code behind de la vue. C'est pourquoi le package View dépends de Manager. D'ici, les managers du package Manager peuvent gérer les instances d'objets nécessaire aux différentes scènes de jeu (ici durant une partie et sur le menu principal).
Comme les managers fournissent une interface unique à chaque élément du jeu, il dépendent donc de presque tout les packages du modèle. En effet, ils permettent à l'utilisateur de faire persister des données sur le système client (ici les scores), de déplacer des objets, d'en créer et de les connaître. Chacun de ces

packages possède à leur tour des dépendances qui leur sont propre pour fonctionne, en effet les Scores ont besoin de connaître la Persistance pour sauver, les Déplaceurs des Collisionneurs pour pouvoir valider les déplacements et les Objets des Bonus pour savoir comment appliquer des bonus sur des objets (utilisés avec les items ici).

Enfin, le Loader dépends de Objet sur le diagramme mais il dépends en réalité de Objet.GestionnaireJeu qui organise la gestion du programme dans sa globalité (création et affichage des scènes). Une fois une scène chargée et affichée, c'est dans le code behind de cette scène (package View) que nous manipulons une instance d'un manager et qui permet d'accéder aux autre classes du modèle.

Nous proposons d'étudier plus en détail le contenu de chaque package.

Diagrammes de classes

Loader : le point de démarrage du programme

Le diagramme de classe du loader est le suivant :

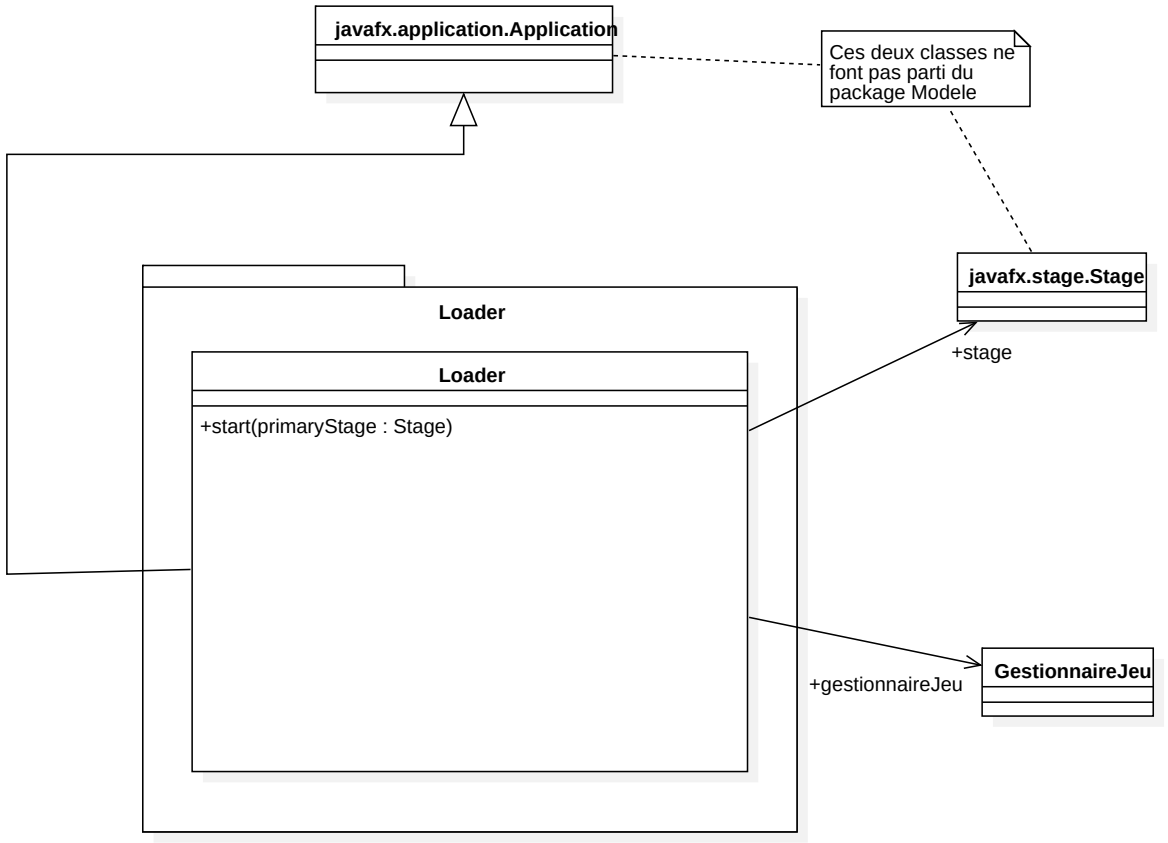


Image 2 : le diagramme de classe du package Loader

Le point d'entrée de chaque application javaFX est la classe Application. C'est pourquoi notre Loader hérite de cette classe. La méthode qui démarre la première fenêtre à besoin d'un objet de Stage, qui est

le container le plus haut en javaFX. Notre loader possède également une instance d'un GestionnaireJeu qui permet d'organiser le passage entre les différentes scènes de jeu (menu, partie).

Objet : regroupe tous les objets de notre application

Le diagramme de classe du package Objet est le suivant :

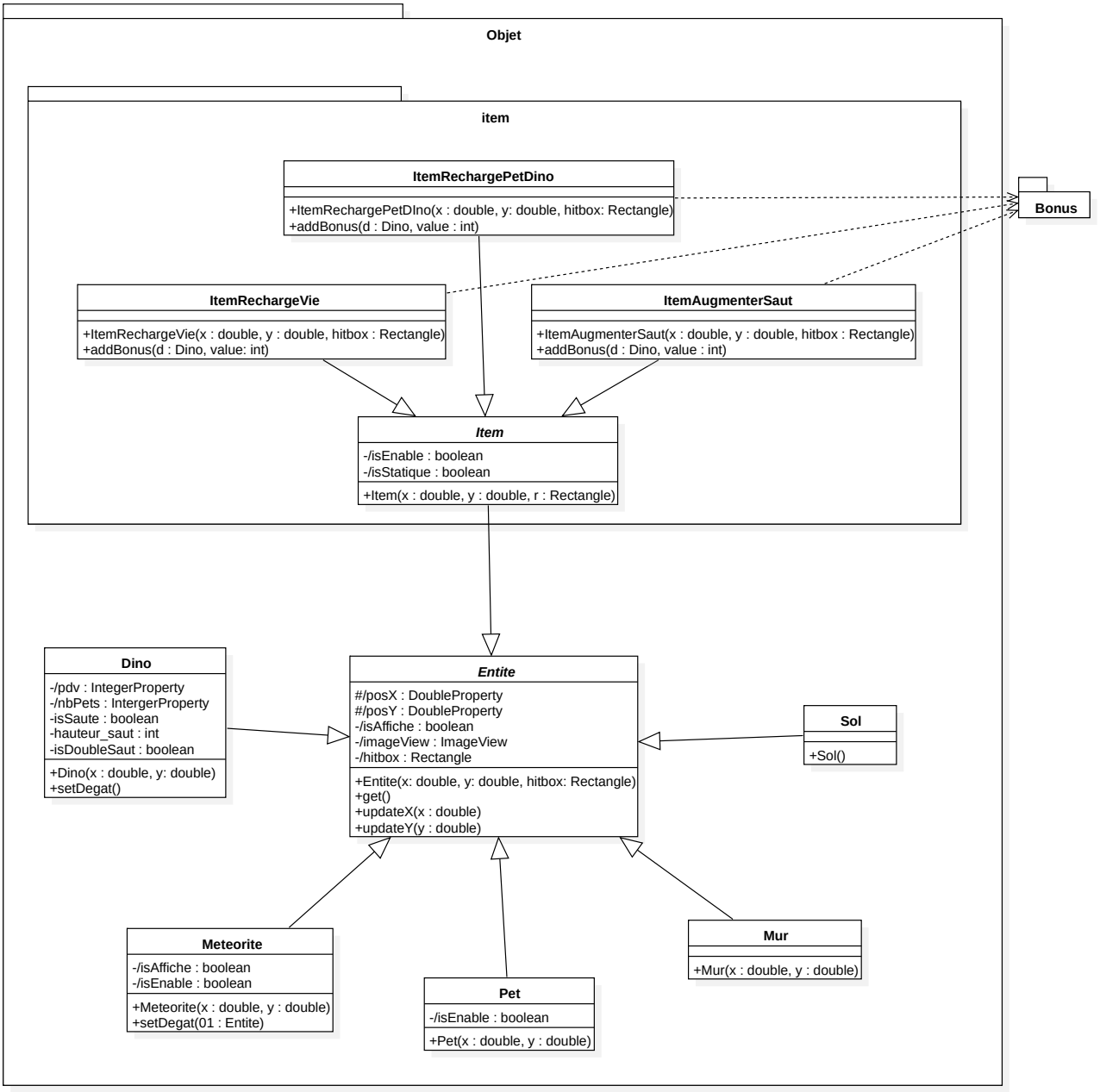


Image 3 : le diagramme de classe du package Objet

Dans ce package, toutes les classes dérivent de la classe Entite. La classe Entite représente une entité quelconque. Chaque objet qui hérite d'Entite hérite aussi de ces attributs et de ces méthodes. Ainsi, chaque entité possède des coordonnées, une hitbox et une image. Les attributs posX et posY d'Entite sont protégées pour que les classes filles puissent y accéder. Nous avons préfixé les attributs d'un « / » quand

on sous-entends qu'il y a un getter et un setter associé à cet attribut et qui n'apparaît pas sur le diagramme. Donc les getter et setter de ces attributs sont aussi accessible dans les classes filles.

Concernant les attributs privés hitbox et imageView d'Entite, nous y accédons depuis les classes filles par le constructeur avec un appel à super(). En effet, nous préférons contrôler la création des hitbox de nos objets lors de la création de nos classes filles plutôt que de laisser cette liberté à un utilisateur de ces classes.

Enfin, nous avons créer un package Item dans Objet pour regrouper les différents types d'items du jeu. Chaque Item est construit selon un modèle qu'est la classe abstraite Item. Comme Item hérite d'Entite, chacun de nos items hérite donc des attributs et méthodes d'Item et d'Entite et donc de coordonnées, de hitbox et d'une imageView. Ces items apportent tous des bonus au joueur, ils dépendent donc tous du package Bonus contenant l'interface IBonus dont le diagramme de classe est le suivant :

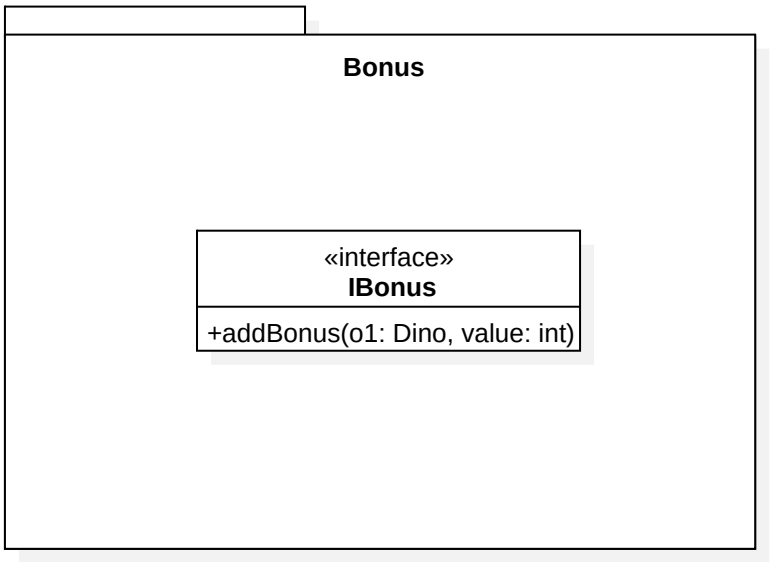


Image 4 : diagramme de classe du package Bonus

Manager : regroupe des interfaces unique d'utilisation des classes du Modèle

Le diagramme de classe du package Manager est le suivant :

Nous avons représenter les classes qui sont utilisées dans les manager dans leur package respectif. La liste des classes dans ces packages n'est pas exhaustif, nous n'avons représenter que les classes utilisées dans les managers et nous n'avons pas détailler leur contenu pour des raison de lisibilité.

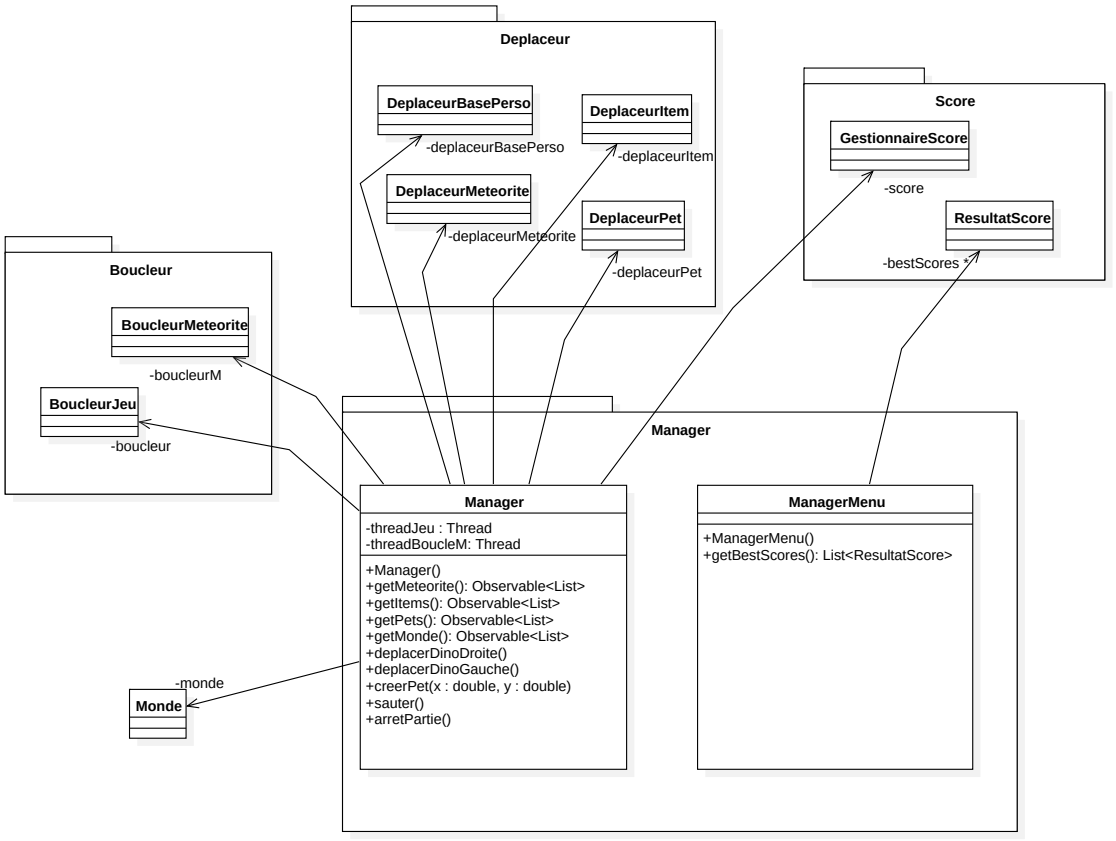


Image 5 : diagramme de classe du package Manager

La classe Manager gère les instances d'objets utiles pour jouer au jeu durant une partie. La classe ManagerMenu gère des instances d'objet quand le joueur est sur le menu principal.

Le Manager possède toutes les instances de déplaceurs et des boucleurs. Grâce à notre implémentation du patron de conception Observateur/Observable, les boucleurs sont capable de notifier les déplaceurs quand une boucle à été faite. Chaque boucle fonctionne dans un Thread pour qu'ils puissent être indépendant de l'exécution du programme. Le Manager possède aussi des instances de score et du Monde pour pouvoir les manipuler. Le Monde est une classe dont la responsabilité est de connaître tout les objets de la scène.

Le ManagerMenu possède la même responsabilité que Manager mais pour le menu principal.

Boucleur : regroupe des classes qui permettent de boucler toutes les x ms

Le diagramme de classe du package Boucleur est le suivant :

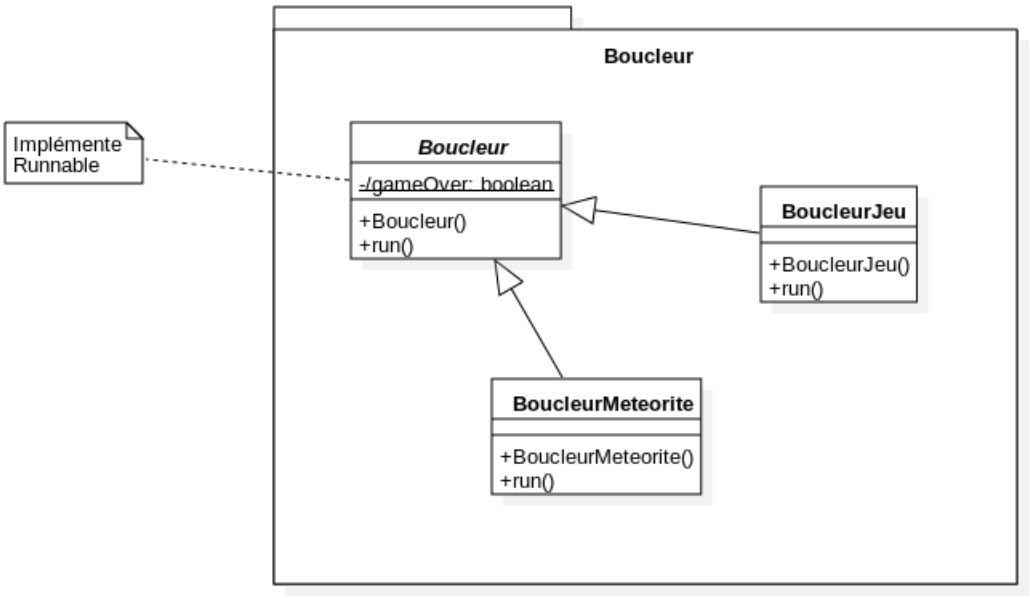


Image 6 : diagramme de classe du package Boucleur

Le package Boucleur contient une classe abstraite qui donne un modèle commun pour tout les boucleurs et les boucleurs en question qui héritent de cette classe. La classe abstraite boucleur implémente l'interface Runnable qui permet l'exécution des boucleurs dans des Thread. La classe Boucleur contient une variable statique gameOver qui permet de stopper tout les boucleurs quand la partie est finie.

Déplaceur : regroupe des classes relatives aux déplacement des objets

Le diagramme de classe du package Déplaceur est le suivant :

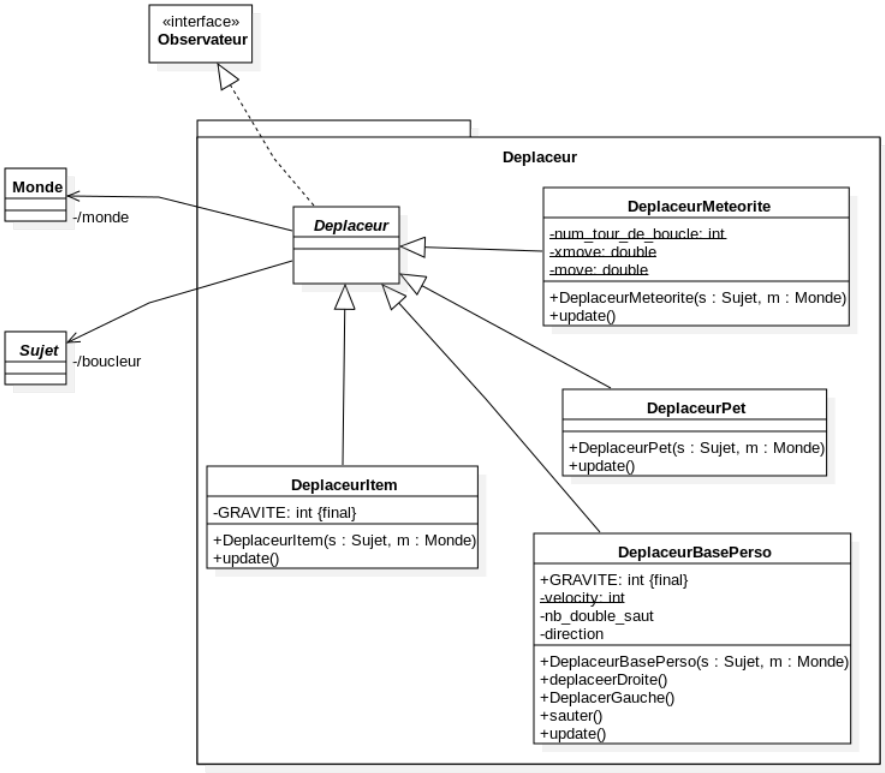


Image7 : diagramme de classe du package Déplaceur

Comme pour le package package Boucleur, le package Déplaceur possède une classe abstraite Déplaceur qui permet de donner un modèle commun à tout les Déplaceurs. La classe abstraite Déplaceur implémente l'interface Observateur ce qui lui permet d'écouter les notifications des boucleurs (qui sont donc les sujets). Le Déplaceur dispose aussi d'une instance du monde pour pouvoir modifier les coordonnées des objets de celui-ci. Dans DeplaceurMeteorite, les attributs sont statiques pour permettre à ces derniers d'être communs à toutes les météorites (xmove et ymove représentent la vitesse de chute des météorites qui augmente au fur et à mesure du temps).

Collisionneur : regroupe des classes relatives aux collisions entre objets

Le diagramme de classe du package Déplaceur est le suivant :

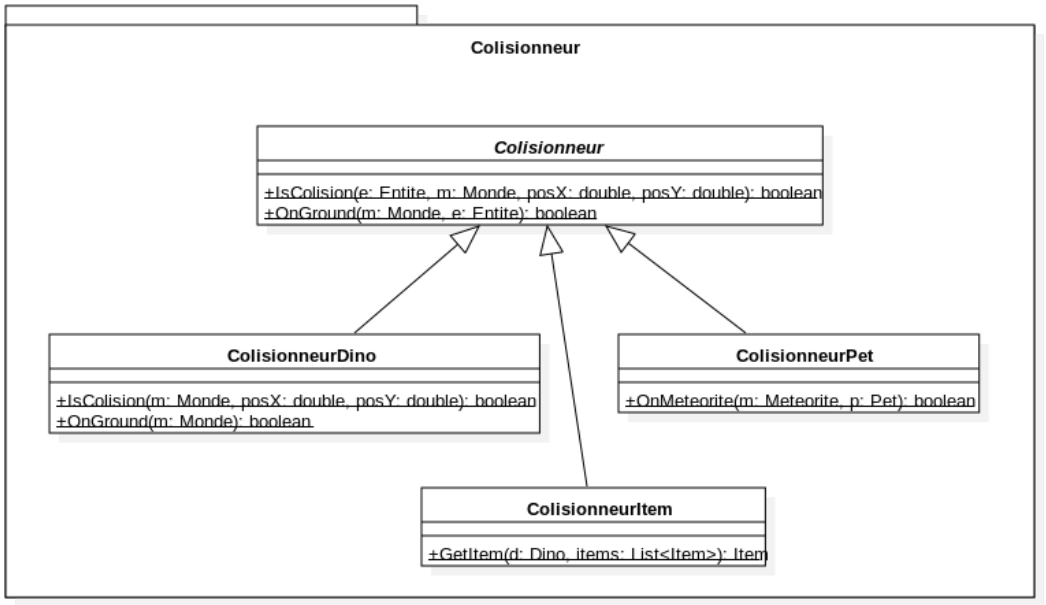


Image 8 : diagramme de classes du package Collisionneur

Comme les packages précédents, le package Collisionneur possède une classe abstraite Collisionneur qui permet de donner un modèle commun à tout les collisionneurs. Les collisionneurs possèdent tous que des méthodes statiques qui nous dispense d'instancier des objets Collisionneurs dans les Déplaceurs alors que ceux-ci ne possèdent pas d'attributs.

Créateur : regroupe des classes qui permettent de créer des objets

Le diagramme de classe du package Déplaceur est le suivant :

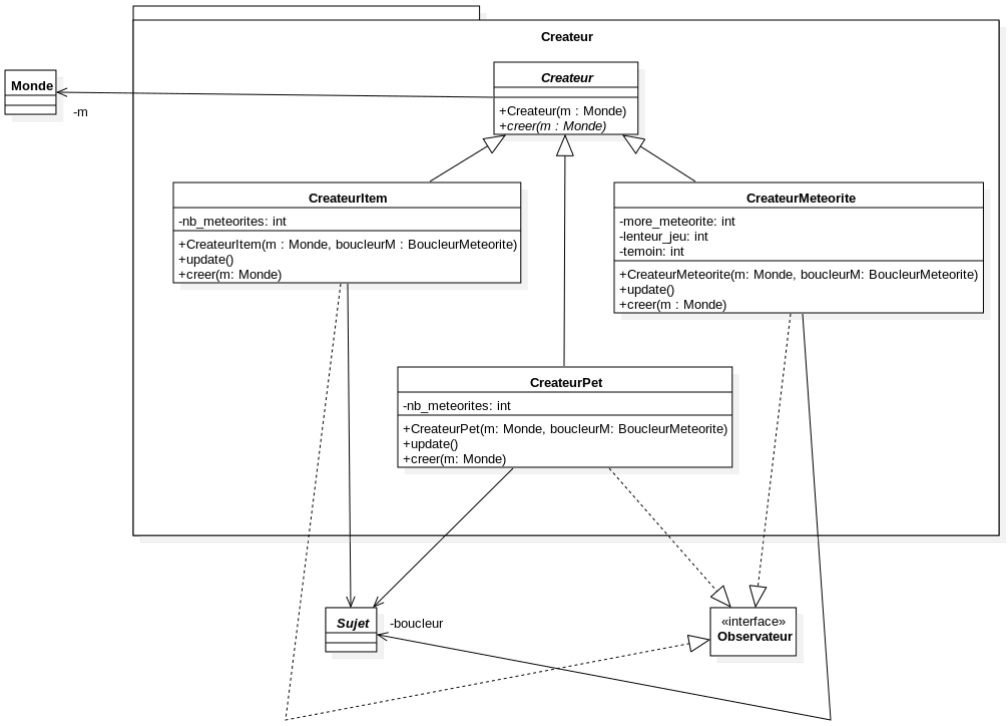


Image 9 : diagramme de classes du package Créateur

Le package ci-dessus est muni d'une classe abstraite Créateur qui, comme précédemment, donne un modèle commun pour les créateurs. Les créateurs disposent d'une instance de monde qui leur permet d'ajouter les objets qu'ils créent dans le monde. Les créateurs implémentent l'interface Observateur car ils écoutent les notifications des boucleurs pour créer à intervalles réguliers.

Score : regroupe des classes qui permettent de représenter et gérer les scores

Le diagramme de classe du package Déplaceur est le suivant :

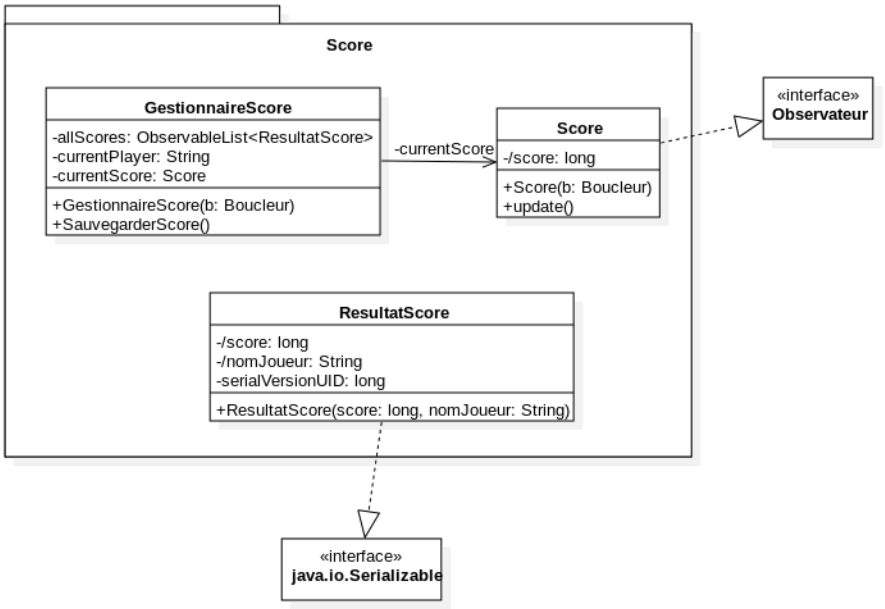


Image 10 : diagramme de classes du package Score

Le GestionnaireScore permet de gérer les scores utilisateur, la classe Score représente un score utilisateur et ResultatScore permet de gérer un score déjà fait.

Persistance : regroupe des classes qui permettent de sauver et charger des objets

Le diagramme de classe du package Déplaceur est le suivant :

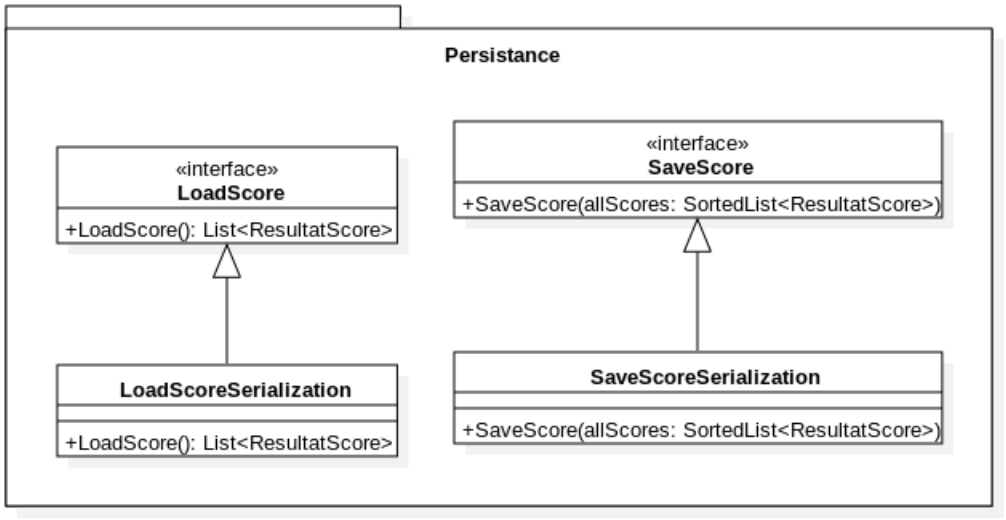


Image 11 : diagramme de classe du package Persistance

Les classes dont le nom est préfixé de Load permettent de charger, celle qui sont préfixés de Save permettent de sauver. Nous avons choisi de créer une interface LoadScore et SaveScore pour prévenir le désir de sauver les scores différemment plus tard. En effet, si nous souhaitons par exemple sauver les scores différemment, il suffit de créer une nouvelle classe qui implémente SaveScore pour le faire.

View : contient les classes du code behind

Le diagramme de classe du package View est le suivant :

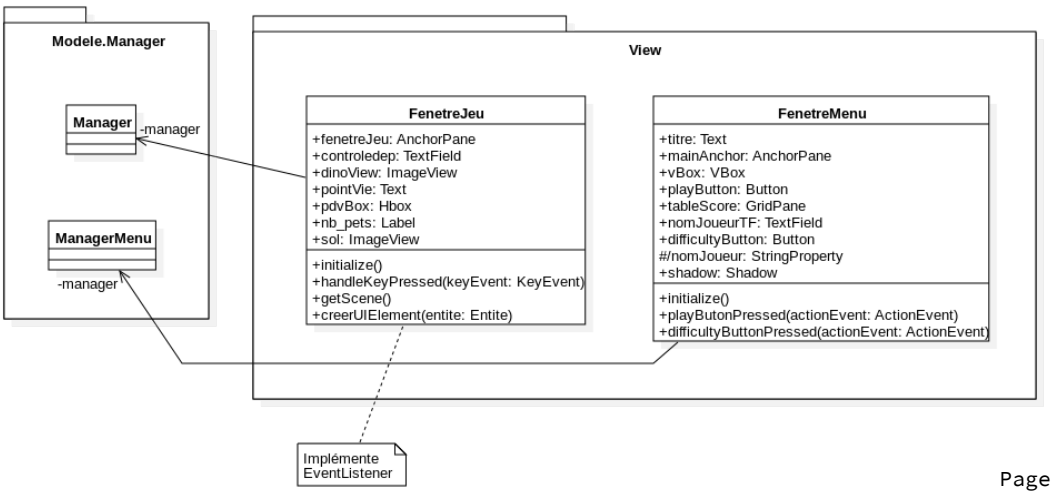


Image 12 : diagramme de classe du package View

Chaque classe possède une instance d'un manager qui lui permet de manipuler les objets du Modèle via une interface unique. La FenetreJeu est le code behind de notre fenêtre FenetreJeu (partie) et FenetreMenu celle du menu. FenetreJeu implémente EventListener pour écouter les actions utilisations lors d'événements.