Emmanuel BABALA COSTA & Sybille CRIMET
M1 Informatique Groupe 2

Guide Développeur

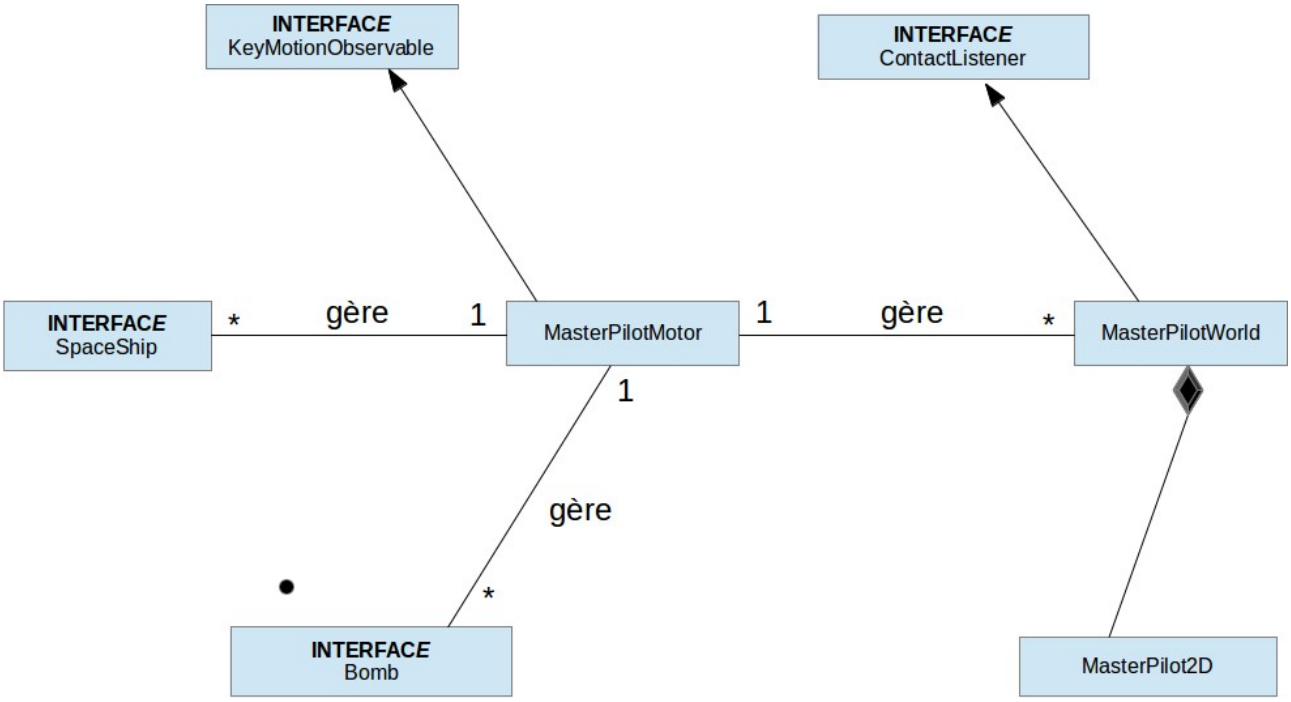
Master Pilot

Emmanuel BABALA COSTA & Sybille CRIMET
M1 Informatique Groupe 2

Sommaire

Emmanuel BABALA COSTA & Sybille CRIMET
M1 Informatique Groupe 2

Diagramme UML général du projet.



Emmanuel BABALA COSTA & Sybille CRIMET
M1 Informatique Groupe 2

Choix d'implémentation

Comme vous avez pû le constater d'après le diagramme UML, le projet est divisé en plusieurs sections. Ces sections correspondent à des packages.

Les packages :

MasterPilot:

→ classe **MasterPilotMotor** :

Elle contient la logique générale du jeu. C'est elle qui charge le jeu depuis la config xml, peuple le monde en fonction, applique la logique des vagues, des planètes, des bombes et des ennemis.

Elle possède une classe interne GameSpec qui se charge de stocker les spécificités du jeu qui est chargé depuis le fichier de configuration xml. Elle contient la boucle générale du jeu et sait détecter la fin d'une partie : soit lorsqu'il n'y a plus d' ennemis, soit lorsque le temps est écoulé. Sa méthode run contient la boucle générale de jeu.

La génération des bombes est basique : pour chaque ennemis détruit on essaie de générer une bombe en fonction du pourcentage récupéré dans le fichier de configuration du niveau.

Les vagues sont lancés lorsqu'il n'a plus d'ennemis dans le monde.

==> Un bug possible à ce niveau serait de que certains ennemis soient bloqués par une planète car notre algorithme de génération ne gére pas les cas des collisions

Bomb:

Contient les fichiers nécessaires à la gestion des bombes et des méga bombes dans le jeu. On gère notamment la collision avec les ennemis, le sac du hero qui ne peut contenir qu'une bombe à la fois.

La classe **GenericBomb** :

C'est une classe générique de bombes qui créer une bombe dont le comportement est définit par un booléen. En effet, la seule difference entre une ***MegaBomb*** et une ***Bomb*** est leur action (implosion ou explosion). Cette action est implementée par la méthode applyBlastImpulse. Une bombe explosive aura un blastPower positif alors qu'une bombe implosive aura un blastPower négatif. Elle applique une force circulaire dans un rayon spécifique. Plus la distance devient grande moins la force sera grande.

Emmanuel BABALA COSTA & Sybille CRIMET
M1 Informatique Groupe 2

Common:

Contient les ressources part&gées par plusierus packages différents notamment tout ce qui concerne les collisions.

→ classe **UserSpec** :

Cette interface est implémentée par tous les objets de MasterPilot. Dans l'object on stocke un élément Fixture.UserData. Ainsi dans les callback des méthodes on pourra recupérer cette interface et appelé ces méthodes.

Graphic:

Contient tous les fichiers nécessaires pour gérer l'affichage du monde et des objets dans ce monde.

→ classe **MasterPilot2D** :

Contient les méthodes nécéssaires pour dessiner des objects dans le monde.

Parser:

Contient les fichiers relatifs au parsing des fichiers xml correspondant à la description des différents niveaux.

Ship :

→ La classe **SpaceSHIP** :

C'est l'interface implémentée par les vaisseaux du jeu. Tous les vaisseaux du jeu implementent l' interface SpaceShip. Un TIE, Cruiser ou autre ennemi est simplement vu comme un SpaceShip et, par polymorphisme, on peut appeler la méthode doMove() qui appellera celle de la bonne classe, sans se soucier du type.

→ La méthode *doMove*(): C'est cette méthode qui définit le déplacement des vaisseaux ennemis et communique avec les méthodes left(), right(), up() et down() afin de définir quand et comment ils doivent se mouvoir.

→ La classe **Hero** :

Le hero c'est l'objet principal du jeu. Il implémente en plus de l'interface Spaceship, le KeyMotionObserver afin d'être notifié de l'action sur une touche du clavier. En fonction de la touche appuyée, elle appelle la bonne action : espace pour tire, b pour tire une bomb, espace pour tire etc.

L'astuce pour le bouclier c'est de rajouter une fixture et de lui attribuer un comportement détaché du hero. Etant donné qu'il entoure constamment le body du héro, quand il réagit à une collision il protege le hero.

→ Classe **HeroBehavior** :

Cette classe détermine le comportement du héro en cas de collision. C'est elle qui gère le niveau de vie du héro dans le mode hardcore.

Emmanuel BABALA COSTA & Sybille CRIMET
M1 Informatique Groupe 2

→ Classe **HeroShieldBehavior** :

C'est le comportement du bouclier en cas de collision. En mode CHEAT, il va détecter une collision et va se placer automatiquement.

→ Les **ennemies** :

Ils implémentent la même interface Spaceship et implementés de la même façon à quelque différences près. Ils se différencient par leurs implementations de la méthode doMove(), la fréquence de tire, la distance optimale de tire et leur forme.

Pour gérer la fréquence de tire, chaque ennemis créer une thread de tire qui contient un booleen fire. Lorsque ce booleen (un volatile) est à true, une autre verification est faite sur la distance. Si ces deux conditions sont réunies, l'objet peut faire feu. La thread sera préalablement arretée avant la destruction de l'objet.

→ La classe ***EnemyBehaviour*** :

Elle définit le comportement par defaut des ennemis au moment d'une collision. Par défaut, tous les ennemis mettent à true la valeur de la variable qui les rend destructible. Cette variable sera demandée plutard afin de savoir si l'objet doit ou non être placé dans la liste des objets à détruire.

→ La classe **EnemyShieldBehaviour :**

C'est le comportement du bouclier des ennemis. Il se place automatiquement pour un nombre limité de fois.

→ La classe **RadarBehaviour** :

C'est un radar pour les ennemis. En fait c'est une fixture qui detecte les collisions entre différents objets et possède une méthode ***getPositionOfCollision()*** qui renvoit la position de la collision détectée. Cette collision est passive puisqu'elle n'interagit pas avec les objets. En fait, on détecte la collision avant qu'il n'y ait collision.

→ La classe **TriangleBehaviour** :

C'est le comportement des triangles qui protégent le squadron.

→ La classe **SpaceShipFactory :**

Elle fournit des méthodes qui créent des vaisseaux de type Spaceship comme le héro ou les ennemis et les enregistrent auprès de leur manager respectifs.

→ La classe **RayFire** :

C'est le tire de tous les vaisseaux du jeu.

Emmanuel BABALA COSTA & Sybille CRIMET
M1 Informatique Groupe 2

→ Les classes **RayFireManager et TrailManager :**

Ce sont des classes dont le but final est de connaître toutes les instances des objets créés afin de les supprimer. Elles ont des méthodes qui renvoient la distance qu'elles ont parcourues.

Star:

Contient les fichiers relatifs à la création des planètes

→ La classe StarFactory :

Elle créer une Star dans le monde et l'enregistre auprès de son manager.

→ La classe StarBehavior :

Elle définit le comportement des Star en cas de collision.

world:

C'est le monde du jeu. Il possède plusieurs managers pour les différents objets qui le composent : ennemies, planetes, bombes et héro. Ces managers permettent au monde de connaître les objets présent, c'est pourquoi il a la responsabilité d'appeler les méthodes de la classe MasterPilot2D (méthode de dessin des formes des objects du monde).

Dans sa méthode draw il va itérer sur chaque objet du monde présent dans jbox2D et, pour chacun d'eux appeler la méthode drawShape qui, en fonction de leur forme appelera la bonne m éthode de MasterPilot2D pour les dessiner.

Afin d'etre à l'écoute des collisions, cette classes implémente aussi le ContactListener de jbox2D qui fournit les méthodes beginContact et endContact. Ces méthodes sont appelées respectivement en début et en fin de collision. C'est dans ces méthodes que l 'on récupère l'instance de l'objet qui est entré en collision pour y appliqué la logique du comportement à suivre. Ces deux méthodes prennent en paramètre deux objets du monde, contact A et Contact B qui entrent en collision, on se sert alors de leur methode getFixture pour réccupérer le champ userData qui possède la methode qui connaît le comportement à adopter. Tous implémentent l'interface UserSpec. Ensuite, la méthode onCollide éffectue le comportement désiré en cas de collision.

On réccupère la sensibilité au sens jBox2D ce qui permet de savoir si un objet est sensible aux collisions ou non. Si il est sensible alors on le dessine. La methode beginContact permet l'activation du bouclier lorque l'objet est sensible aux collisions et la récuperation des items.

Dans la méthode endContact, on vérifie via la méthode isDestroyable de userSpec si l'objet doit être supprimé ou non. Si tel est le cas on le place dans la liste des objets du monde à suprrimer. C'est le moteur du jeu qui se chargera de les retirer ultérieurement du monde. Ceci à cause du fait que jbox2D bloque le monde pendant un callBack de collision.

La méthode getDestroyBody va renvoyer la liste des objets du monde à detruire. Cette liste est remplie soit durant un callback de collision dans la méthode endContact, soit durant l'appel de la méthode elle même qui va intérroger le manager des traits du vaisseau et des balles tirées pour récupérer les objets qui doivent être retirés

Emmanuel BABALA COSTA & Sybille CRIMET
M1 Informatique Groupe 2

Main;

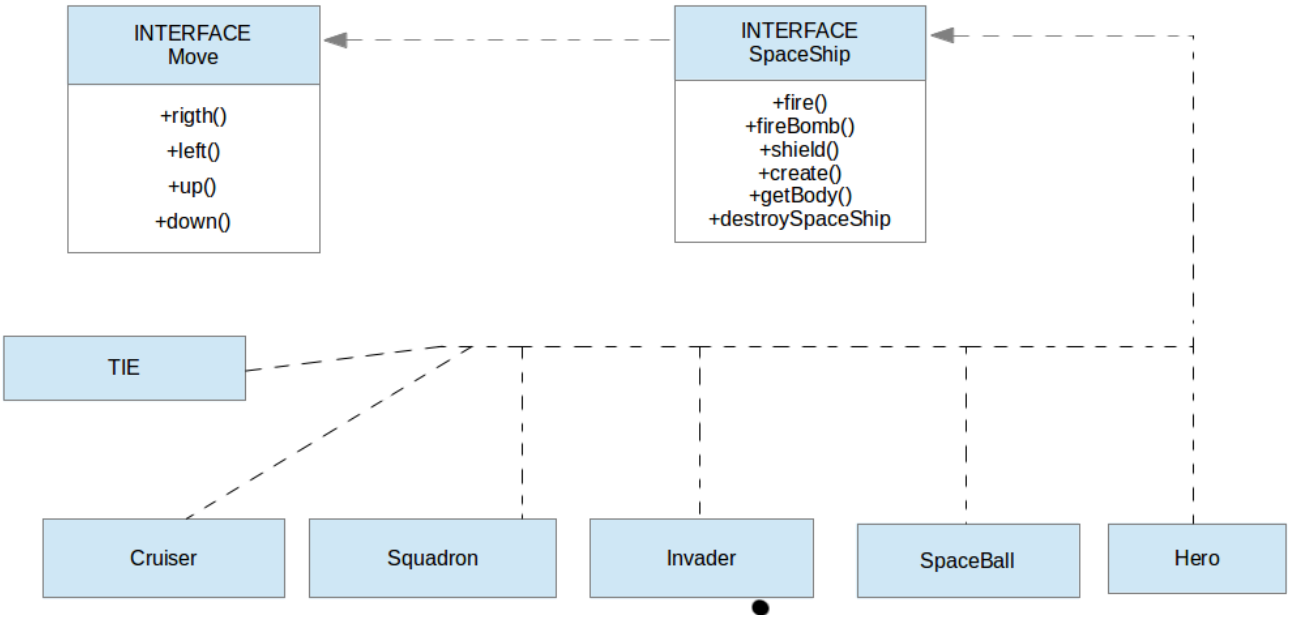
Contient le fichier main qui permet le lancement du jeu.

Les Interfaces :

SpaceShip :

toutes les classes qui représentent des vaisseau implémentent cette interface puisque qu'elle contient le méthodes générales de tir, de déplacement etc.

Voici comment se présente l'interface :

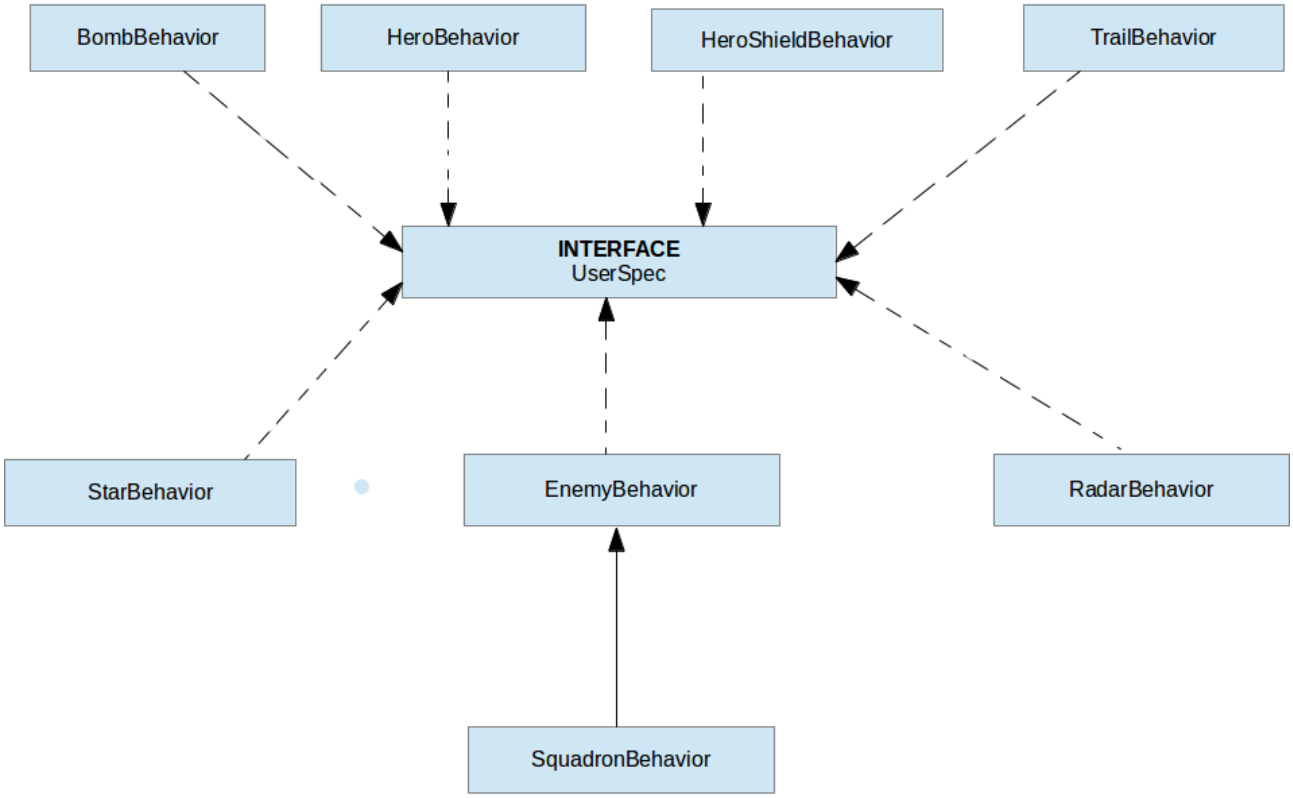


Emmanuel BABALA COSTA & Sybille CRIMET
M1 Informatique Groupe 2

UserSpec :

Heritent de cette interface tous les objets qui sont utilisés dans le champ userdata des fixtures de la librairie jbox2D. Ces objets ont pour suffixe commun bihavior. Cette interface est particulièremet utile pour gérer les collisions.

Voici la réprésentation des liaisons qui sont faites avec cette interface :

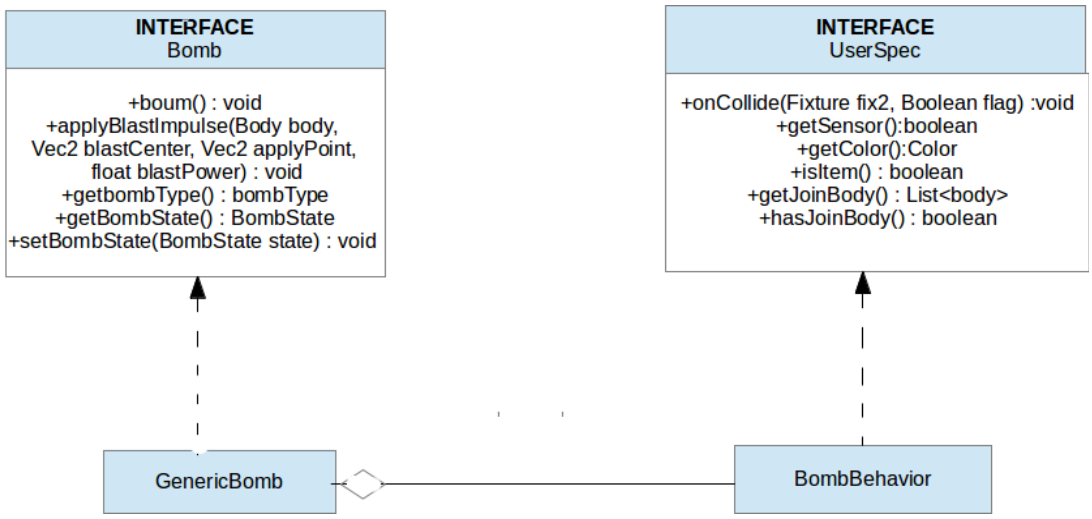


Emmanuel BABALA COSTA & Sybille CRIMET
M1 Informatique Groupe 2

Bomb :

Permet connaître le type de la bombe (implosive ou explosive) .

Voici comment se présentent l'interface :



Améliorations possibles

Nous aurions pu rajouter une barre de vie aux ennemis. Plutôt que d'afficher des formes géométriques, nous aurions pu charger des images pour afficher les planètes, le héro et les ennemis. Nous aurions pu aussi rajouter des spécificités aux planètes c'est-à-dir avoir des planètes de différentes couleurs, de différentes taille et de différente densité.

Conclusion

Ce projet s'est révélé intéressant puisqu'il nous a permi d'appprendre a nous organiser pour travailler en binôme d'une part, et d'autres parts, parce qu'il nous a permi de maitrîser plus largements les notions abordées en cours, notamment en ce qui concerne les lambdas et le design pattern pour avoir un code source plus facilement modulable.