Université de Bordeaux

Master 1 Informatique

Projet Approche Objet

réalisé par

Rémi Brisset

Sofiane Benzait

Pierre Chauveau

Abdelwahid Hadj Zoubir

22 decembre 2017

Table des matières

1 Présentation du projet

Le but de ce projet est de nous faire créer un jeu de type Pacman. Ce type de jeu permet au joueur de diriger un héro dans un labyrinthe, de récupérer des items dans ce labyrinthe et de se diriger vers la sortie (ici représentée par une porte) tout en évitant les ennemis qui le suivent. Ce projet doit être fait en respectant au maximum le paradygme objet et en utilisant le langage JAVA. Il faudra aussi utiliser quelques patrons de conception tels que singleton, MVC, Observable.

Le joueur doit avoir la posibilité de diriger le héro du jeu grâce aux flèches directionnelles, de plus le héro ne pourra pas traverser les murs du labyrinthe. Pour gagner le joueur devra se diriger vers la sortie qui sera sous la forme d'une porte tout en évitant les ennemis qui le suivront. Le joueur pourra récupérer sur son passage des bonbons qui seront disposés de manière aléatoire dans le labyrinthe.

2 Bilan

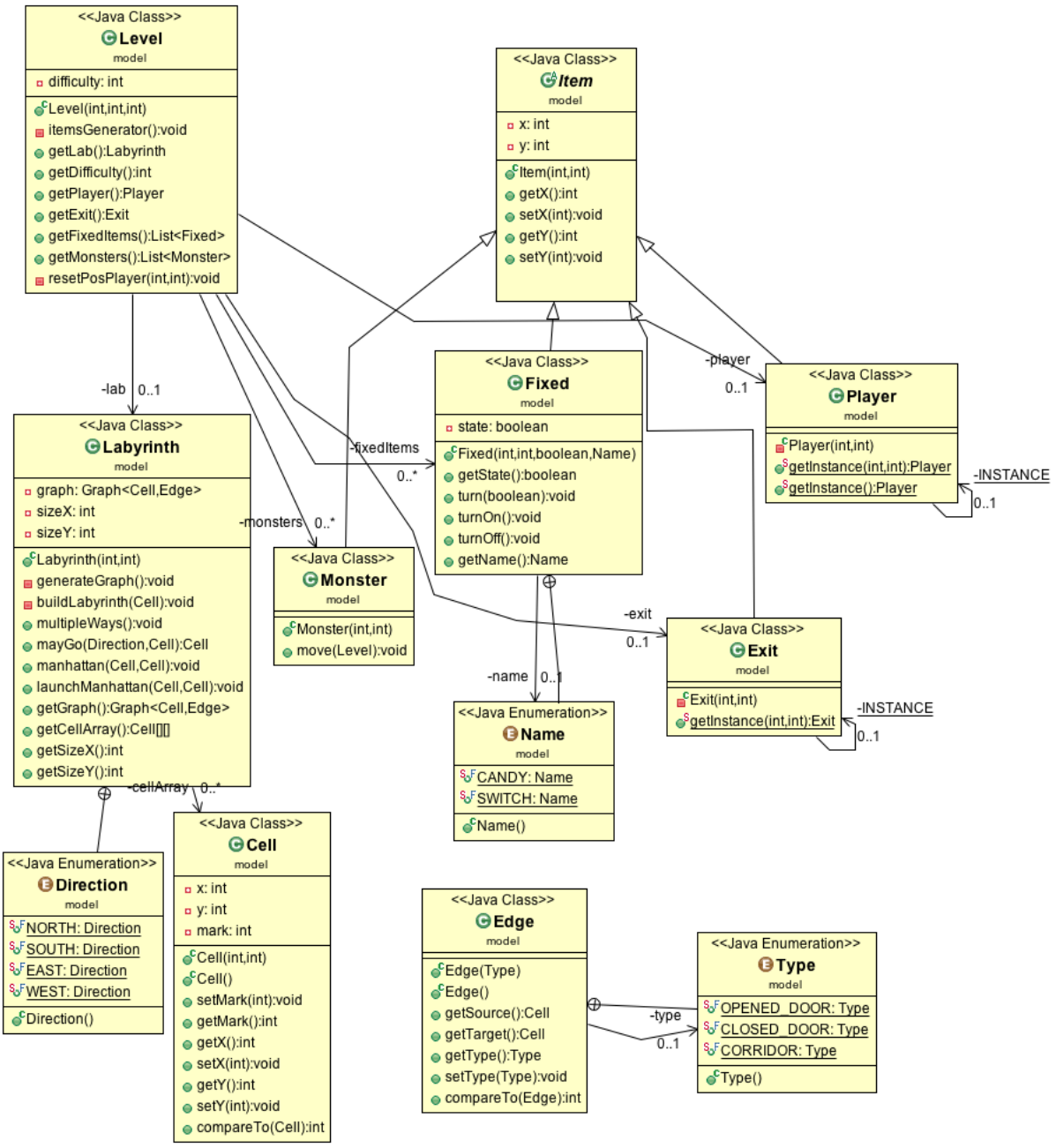
Notre jeu est fonctionnel. Au lancement, un niveau est généré. Si l'utilisateur atteint la sortie, un nouveau niveau de difficulté supérieur est généré et ce jusqu'au troisième et dernier niveau(ce choix est totalement arbitraire). Si le joueur termine le dernier niveau le jeu se ferme.Si il se fait rattraper par un monstre, le jeu s'arrete. A tout moment l'utilisateur peut quitter le jeu en appuyant sur la touche *escape*.

3 Programme

Afin de respecter la contrainte MVC demandé dans le cahier des charges le programme que nous avons crée contient 3 packages : View, Controller, Model. Nous avons implémenté MVC de tel façon que le controller gère les evènements afin de modifier les informations nécessaires dans le modèle puis demande à la partie correspondante dans la vue de se mettre à jour. La vue va alors demander au modèle les informations qui sont nécessaires à cette mise à jour

3.1 Package Modèle

3.1.1 Diagrame UML



3.1.2 Level

Level est la classe regroupant tous les éléments d'un niveau, c'est à dire le Labyrinthe, ainsi que tous les items. Les items sont générés en fonction de la difficulté actuelle, nous avons fait une gestion très basique mais qui pourrait être facilement améliorée, en changeant le nombre de monstres et de bonbons en fonction des niveau de façon plus ergonomique. Actuellement il n'y a que 3 niveaux, le premier sans monstre et sans bonbon, le deuxieme avec un monstre et un bonbons et ainsi de suite. Pour les emplacements du joueur, des monstres et des bonbons nous avons aussi opté pour une génération simple, le joueur apparait en haut à gauche, la porte en bas à droite et les monstres ne peuvent apparaitre dans le quart nord ouest du plateau. Nous aurions pu utiliser l'algorithme de

Manathan pour placer la porte le plus loin possible du joueur, définir un emplacement aléatoire du joueur et placer les montres à une distance raisonnable.

3.1.3 Labyrinthe

Cette classe représente le labyrinthe, c'est dans cette classe que l'on construit le labyrinthe parfait et que l'on ouvre certaines portes. Nous avons choisi de représenter le labyrinthe par un graphe qui contient des cellules entrées dans un tableau, le tableau permet un accès plus rapide aux cases, sans avoir à forcément passer par le graphe et le parcourir. La difficulté n'entre pas en compte pour la création du labyrinthe dans notre implémentation, les labyrinthes sont toujours créés avec les mêmes paramètres.

3.1.4 Item

Cette classe est une classe abstraite qui définit les attributs et les services communs à tout item à savoir une position en x et en y et les getters et setters correspondants. Toutes les autres classes représentant un éléments du jeu hors labyrinthe héritent de cette classe (le joueur, les monstres, les collectables, la sortie etc ...)

3.1.5 Player

Cette classe hérite de la classe Item, de plus le constructeur de cette classe implémente le patron de conception *Singleton*, car nous voulons être sur qu'il n'y ai un seul héro durant notre jeu.

3.1.6 Exit

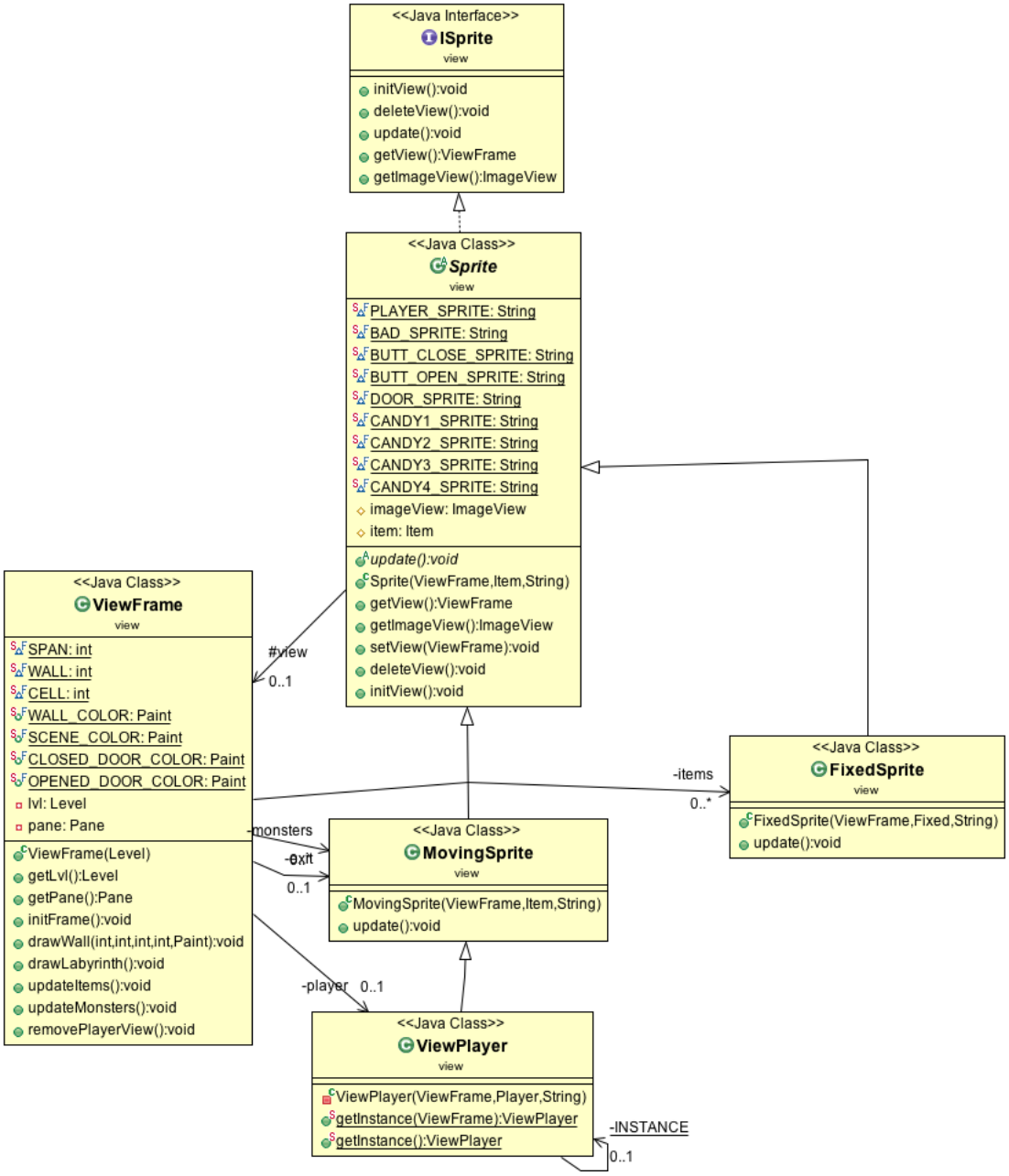
Cette classe hérite de la classe Item, de plus le constructeur de cette classe implémente aussi le patron de conception *Singleton*, l'instance Exit représente la porte de sortie dans le labyrinthe.

3.1.7 Cell

Cette classe représente une cellule du labyrinthe, ces méthodes permettent notamment de récuperer et d'ajouter les coordonnées d'une cellule et même de comparer les coordonnées d'une cellule à une autre.

3.2 Package View

3.2.1 Diagrame UML



3.2.2 ViewFrame

Cette classe hérite de la classe Stage de $javafx$, elle crée la fenêtre puis demande toutes les informations nécessaires au modèle pour dessiner le labyrinthe et afficher les sprites.

3.2.3 ISprite

ISprite est une interface qui définit les services disponibles pour l'utilisateur pour l'utilisation et l'affichage de sprite à l'écran en particulier une méthode d'initialisation *initView()*, de suppression *deleteview()* et de mise à jour *update*.

3.2.4 Sprite

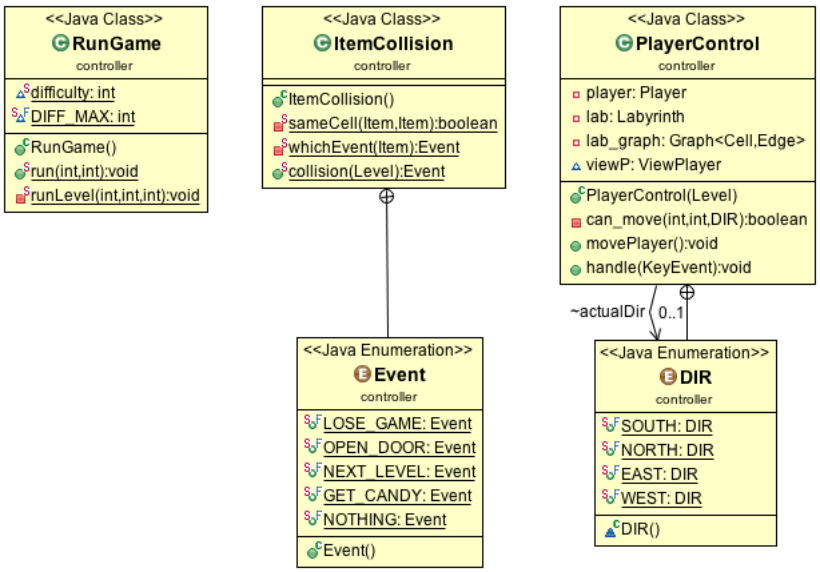
Tout les services précedemment décrits sont implémentés dans cette classe abstraite à l'exception de *update()* qui est déclaré *abstract*. En effet la mise à jour du sprite peut être de changer sa position d'affichage dans le cas d'un élément mobile (joueur ou monstre) ou de vérifier si le sprite doit être retiré de l'écran dans le cas d'un collectable.

3.2.5 ViewPlayer

Cette classe est une classe fille de la classe MovingSprite. Elle correspond au sprite du joueur. Elle respecte donc aussi le patron de conception **Singleton**.

3.3 Package Controller

3.3.1 Diagrame UML



4 Limitations

Nous ne fournissons aucun programme de test et notre documentation est incomplète. Cependant, dans la méthode implémentant l'algorithme de création du labyrinthe, nous nous assurons que le labyrinthe créé est parfait. Nous ne nous assurons par contre pas que le niveau est faisable (c'est à dire que le joueur ne se retrouve pas dans une situation ou il va necessairement perdre). Pour éviter que cette situation arrive, nous avons implémenté une solution très sommaire(voir point *3.1.3*). Cependant, dans une version amélioré du jeu , avec potentiellement beaucoup d'ennemis, cette méthode s'avère très vite obsolète.

4.1 Difficultés rencontrées

Afin de gérer l'animation des sprites nous avons utiliser l'outil *Animation* de la bibliothèque javafx.animation en lieu et place de l'outil *Timeline* demandé.

Nous avons commencé la partie codage trop tôt, alors que notre architecture était trop basique et oubliée plusieurs facteurs. Nous avons donc perdu du temps à recoder et à rectifier certains aspects de l'implémentation au fil du projet. Nous avons pu saisir l'importance du travail de conception en amont de l'implémentation particulièrement en approche objet.

4.2 Améliorations possibles

On détaillera ici les différentes améliorations que nous n'avons pas pu implémenter par manque de temps. Nous n'avons pas utilisé certains design pattern comme *Observable*

qui nous auraient été utile pour la gestion des sprites.

Nous n'avons pas utilisé le mécanisme des portes. Une grande partie est déjà implémentée (l'affichage et le modèle sont fonctionnels) mais un travail algorithmique sur la faisabilité du niveau tel que décrit au point *3.1.3* est nécessaire pour gérer le positionnement des portes et des interrupteurs.

Lorsque l'on finit un niveau, la fenêtre se ferme et le prochain niveau apparait dans une nouvelle fenêtre. Il serait préférable de faire tout apparaître successivement dans le même espace.

Quelques amélioratons esthétiques pourrait être ajoutées comme par exemple un menu de choix de la difficulté, un affichage du score en fonction des bonbons ramassés ou encore un chronomètre pour pouvoir afficher les performances du joueur.