Projet Pacman

Thomas Bianchini - Nathanael Couret - Antoine Valette - Clement Taboulot

2 mars 2015

Table des matières

1 Organisation du projet

Étant un groupe de quatre collaborateurs, nous avons commencé par nous organiser afin que chacun connaisse son rôle dans le but de faire avancer le projet correctement.
Tout d'abord, la base de notre travail repose sur la possibilité pour chacun des membres de disposer des ressources nécessaires afin d'avancer ses tâches. Pour cela, nous avons mis en place un repository git (hébergé sur GitHub). Ce choix se justifie car la mise en place est simple et l'équipe avait les connaissances suffisantes d'utilisation.
Ensuite nous avons décidé de réfléchir tous ensemble sur le sujet afin d'ébaucher une architecture pour notre application qui sera expliquée en détail dans la partie structure de l'application. Puis nous nous sommes mis d'accord sur les tâches de chacun :

— Antoine : algorithme minimax
— Clement : algorithme du plus court chemin
— Nathanël : deplacement aleatoire, gestion des cartes pourries, gestion des victoires/défaites
— Thomas : parser de fichier de map, mis en place de la structure, gestion IHM
— Clement - Thomas : deplacement des fantomes
— Groupe : code review, javadoc, rapport

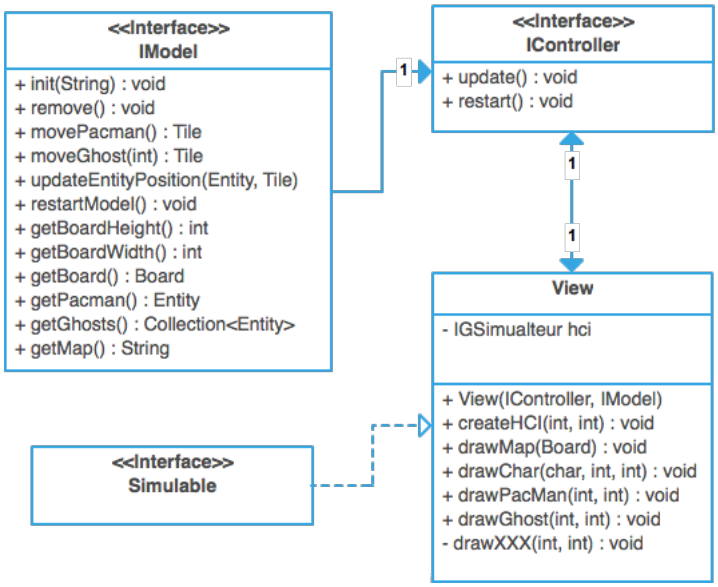
2 Structure de l'application

2.1 Le Modele - Vue - Controlleur

La première idée que nous avons eu a été d'introduire un desgin pattern afin de structurer l'architecture globale de l'application.
Le pattern MVC permet en effet de découper l'application en trois grandes parties :

— l'affichage de données (Vue)
— la sauvegarde et manipulation de données (Modele)
— la gestion des interactions de l'utilisateur (Controlleur)

Nous avons donc adapté le MVC à notre application dont voici le diagramme de classe :



En suivant la logique de ce diagramme de classe, nous remarquons que le controller ne possède que deux méthodes. En effet, nous avons à gérer deux types d'évènement : l'avancement de la simulation gérée par la

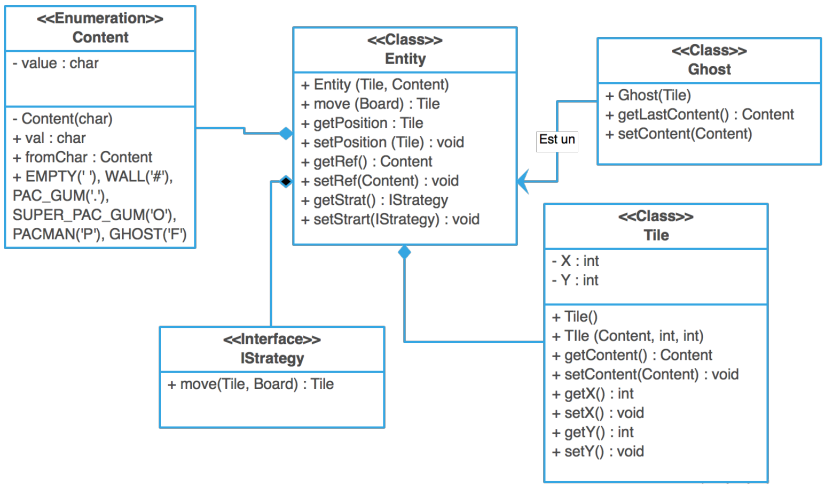
méthode update() et la remise à zéro de la simulation (restart()). Afin de détailler une des deux méthodes, prenons l'exemple de la mise à jour :

```
/* DANS LA VUE */
public void next() {
    controller.update();
}

/* DANS LE CONTROLLEUR */
//Methode update
public void update () {
    updatePacman();
    updateGhosts();
}

//Methode updatePacman
private void updatePacman() {
    Tile tile = this.model.getPacman().getPosition();
    Tile newTilePM = this.model.movePacman();
    this.model.getPacman().setPosition(newTilePM);
    this.model.updateEntityPosition(this.model.getPacman(), tile);
    view.drawPacMan(newTilePM.getX(), newTilePM.getY());
    view.drawSpace(tile.getX(), tile.getY());
}
```

Lorsque l'utilisateur appuie sur le bouton next sur l'IHM, la méthode next() de la vue est appelée. La vue fait alors appel à son controller qui devra gérer cette évènements.
Dans ce cas, la méthode update() du controller a pour rôle de mettre à jour Pacman et les fantômes. En se penchant sur la mise à jour de Pacman, nous voyons le mécanisme de déplacement d'un personnage. Nous récupérons la case courante, puis le modele se charge de calculer la nouvelle position du personnage, se fait ensuite l'attribution de cette nouvelle position. La partie gestion des données est finie, il ne reste qu'à déléguer l'affichage à la vue. Les méthodes de dessin de la vue se servent de l'objet simulable. Quant au modele, le calcul des postions est délégué au personnage (voir la partie gestion de personnage ci-après). Son principal rôle est de stocker les données et de les maintenir à jour à chaque tour. Son second rôle est aussi de parser les fichiers de cartes et créer les données associées.

2.2 La gestion des personnages

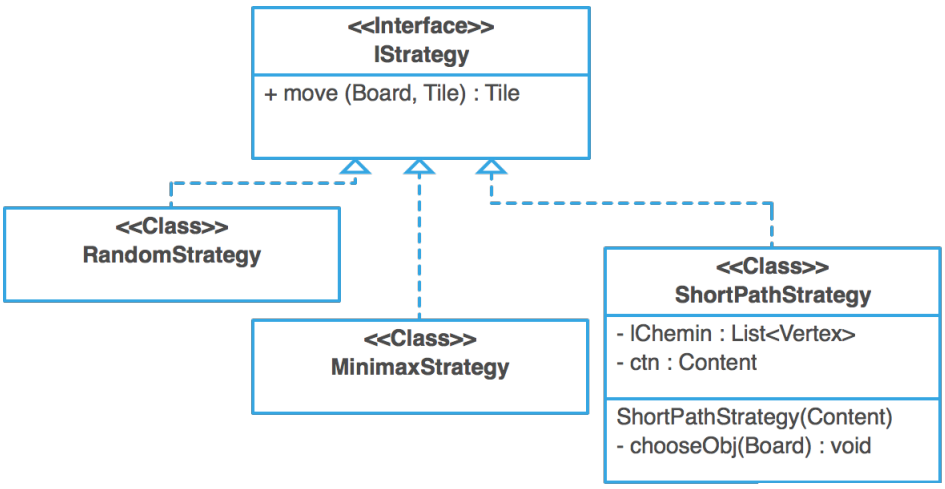
Les personnages sont la pièce maîtresse de l'application. Pour la simulation il y a deux types de personnages : Pacman et les fantomes. Ci dessous, le diagramme de classes lié aux personnages :



Le premier choix était de savoir si Pacamn et les fantômes seraient des classes à part entière. La seule différence notable est le fait que Pacman mange tout ce qu'il trouve sur son chemin (excepté les fantomes) tandis que les fantomes doivent se déplacer sans altérer le contenu des cases qu'ils parcourent (excepté Pacman). C'est pourquoi la classe Ghost a un attribut lastContent afin de sauvegarder le contenu (cf Enum Content dans le diagramme) de la case sur laquelle le fantome se trouve. Ceci permettra notamment de redessiner correctement la carte après passage d'un fantome.
D'autre part les points communs de tous les personnages sont le fait qu'ils se deplacent. Ce déplacement nécessite donc que chaque personnage possède une position, sa position courante. La classe Tile (cf partie Gestion de la carte) sert à stocker l'emplacement de nos personages. La méthode move() du personnage retourne la nouvelle position du personnage et délègue ce calcul à la stratégie (cf partie sur les Stragtégies) du personnage.

2.3 Les stratégies de déplacement d'un personnage

Notre projet inclut des stratégies car les personnages peuvent en effet se déplacer de différentes manières en fonction du contexte ou en fonction du choix de l'utilisateur. Comme le montre le diagramme ci après, chaque stratégie correspond à un type de déplacement (les algorithmes sont expliqués plus loin).

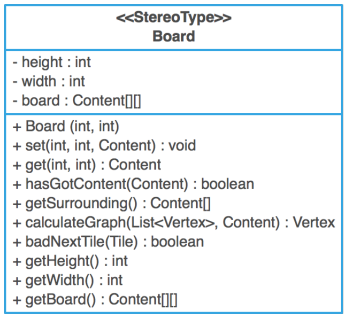


La méthode move() de chaque stratégie prend en paramètre un Board (cf Gestion des cartes) et une Tile

représentant la position courante d'un personnage sur le jeu. La méthode retourne la nouvelle position du personnage.

2.4 Gestion de la carte

Le modele du MVC permet de stocker toutes les données de la simulation. Un carte est un tableau en deux dimensions de taille fixe qui contient des éléments de type Content. Ainsi au démarrage de l'application le modele est initialisé en parsant le fichier de carte sur lequel on veut jouer. Les données lues au moment du parsing sont stockées dans un objet de type Board dont voici la description :



Le choix du tableau de taille fixe nous permet d'autre part de gérer les fichiers malformés, les dépassement ce tableau... En effet, c'est grâce à cette objet que nous pouvons tester les cartes "pourries" (les cartes où Pacman ne pourra jamais gagner car il est bloqué).

Nous avons essayé au travers de cette application de mettre en place des mécanismes de Progammation Orientée Objet tels que les designs patterns (MVC, Strategy), l'héritage... D'autre part, nous avons tenté de découper le code afin que le travail en équipe soit facilité dans le sens où la compréhension de chaque fonction ou classe soit la plus claire et plus rapide pour chaque membre du groupe.

3 Les algorithmes

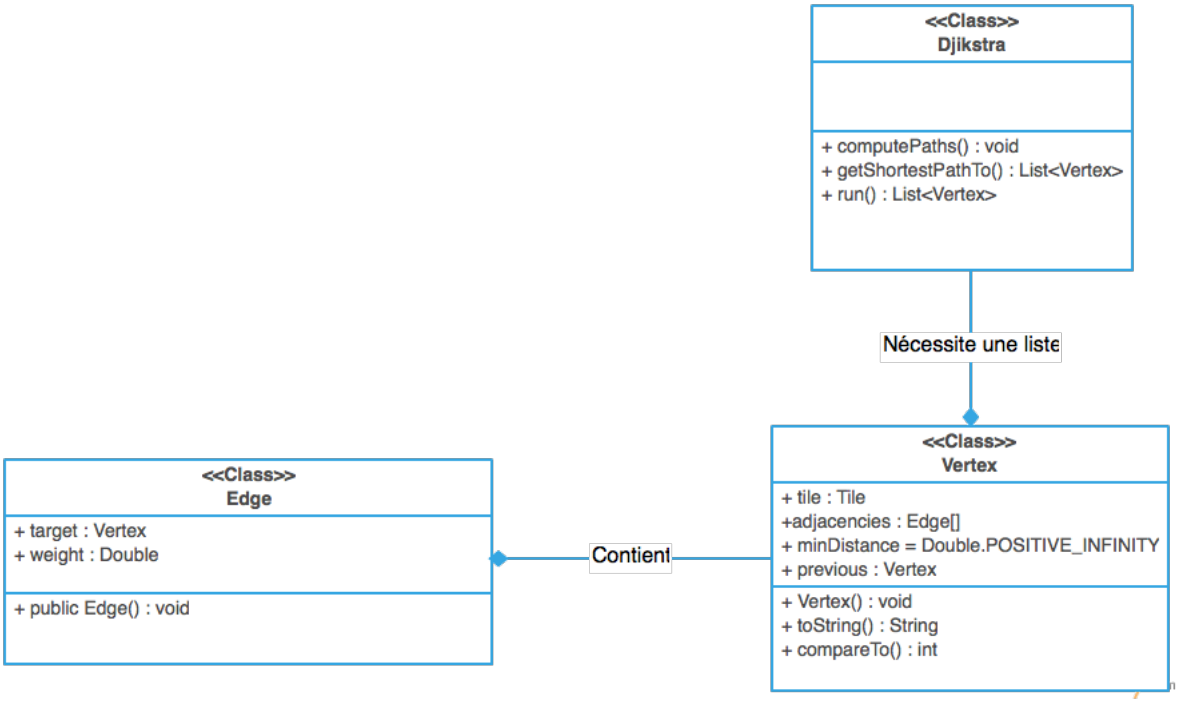
3.1 Déplacements aléatoires

3.2 Algorithme du plus court chemin

3.2.1 Choix de l'algorithme

Pour le calcul du plus court chemin nous avons fait le choix d'appliquer l'algorithme de Dijkstra. C'est un algorithme simple, efficace et que nous avons vu cours de Recherche Operationnelle. Pour l'implémentation nous avons créé un package Djikstra comportant les classes suivantes :

— Vertex : cette classe correspond au noeuds du graphe utilisé pour l'algorithme ;
— Edge : cette classe correspond au arc entre 2 noeuds adjacents ;
— Djikstra : cette classe permet de calculer le plus court chemin entre un source et un objectif.



Le calcul du plus court chemin se deroule en suivant les étapes qui suivent :

— Créer une liste repertoriant tous les noeuds de la carte, c'est-à-dire créer une liste de Vertex ;
— Pour chaque, noeuds déterminer qui sont ses voisins et le poids du passage du noeud A au noeud B (pas defaut 1). Cela correspond a un tableau de Edge ;
— Lancer l'algorithme de Dijkstra avec la methode run() en précisant le point de depart et le point point d'arrivé dans les paramètres.
— Le chemin retourné est une liste de Vertex correspondant au chemin à suivre.

3.2.2 Tests et problèmes rencontrés

Pour pouvoir valider cette fonction de l'application, il fallait qu'elle réponde à 3 critères :

— Si 2 chemins se présentent à Pacman, il doit prendre le chemin le plus court. Le test utilisant la carte Djikstra2.map valide cette fonctionnalité ;
— Si 2 chemins se présentent à Pacman, et que le chemin le plus court contient un obstacle, alors Pacman emprunte un autre chemins. Cette fonctionnalité est validée avec le test utilisant la carte Djikstra3.map ;
— Si la carte contient plusieurs SUPER PAC GUM, alors Pacman mange toutes les SUPER PAC GUM en se dirigeant toujours vers celle qui se trouve la plus proche de lui. Cette fonctionnalité est validé par le test contenant la carte Djikstra1.map ;

Durant l'implémentation de cette fonctionnalité un problème essentiel ont été rencontré.
Il s'agit de la construction du graphe qui permet à l'algortihme de Djikstra de fonctionner. En effet, il devait représenter fidèlement la carte. Plusieurs itérations ont été nécessaire pour obtenir une fonction opérationnelle. Par la suite, pour améliorer cette fonctionnalité nous avons pris en compte qu'il n'était pas nécessaire de recalculer à chaque déplacement un nouveau graphe et un nouveau plus court chemin. En effet, un nouveau chemin n'est calculé que si Pacman vient de manger une super pacgum.
Lorsque Pacman a mangé toutes les super pacgum, son objectif devient alors les simples pacgum. Et à partir du moment où la carte ne contient plus de pacgum alors Pacman n'a plus de réel objectif. Sa statégie passe donc en aléatoire.

3.2.3 Améliorations possibles

Cette fonctionnalité n'est pas parfaite et plusieurs amélioration serait possible. Premièrement si Pacman et les Ghosts ont pour statégie le plus court chemin alors le résultat obtenu n'est pas celui souhaité. De plus, il est possible que Pacman se fasse manger par un Ghost. Quand un chemin est calculé Pacman va suivre se chemin, et l'évaluation de la viabilité de la case suivante n'est pas totalement fonctionnelle. Dans certains cas, la case suivante vers laquelle il va se diriger est aussi une case adjacente à un ghost. Et il arrive que Pacman et un Ghost se retrouvent au même endroit. Une méthode badNextTile() est présente pour évaluer cette viabilité mais son fonctionnement n'est pas opérationnel.
Le calcul de plus court chemin entre un point A et un point B fonctionne. Mais un ajout d'intelligence pour faire face à certaines situations pourrait être un axe d'amélioration.

3.3 Minimax