Spécifications techniques

Aurélien Svevi

14 mars 2015

Table des matières

Technologies employés

— Nous coderons notre projet en langage java avec les normes de développement fournis.

— Pour le partage des donnés nous utiliserons le logiciel **git** qui nous permettra de conserver un historique de toutes les modifications sur le projet.

Fonctionnement du jeu

1 Le plateau de jeu

Le plateau de jeu sera composé d'un certaine nombre de *Position* définit par l'utilisateur. Pour ce faire nous disposerons d'une class *map* contenant une hashmap de *Position* avec comme clé un String contenant une lettre et un chiffre (c4,a2...) et comme valeur une instance de la case. De plus chaque cases aura plusieurs attributs booleans permettant de savoir ce qui est présent sur cette case (base, obstacle, mine...)

2 les robots

Il y a 3 types de robots :Tireur, piégeur et char. Tous ces robots ont des caractéristiques qui leurs sont propres (portée, vie, déplacement...) en revanche leur fonctionnement est semblable, ils peuvent tous se déplacer, attaquer et sont tous sur une *cases*.

Ainsi nous allons créer une class abstraite *Robot*. Dans cette class nous créerons divers fonctions visant à gérer la vie des robots, leur position..., Ensuite nous créerons 3 autres class : *Piegeur* , *Tireur* et *Char* qui hériterons de la class *Robot* et dans laquelle nous ajouterons les éventuels particularité des robots (quantité de mines...).

3 les attaques

Pour gérer l'attaque des robots qui diffère de l'un à l'autre (par exemple, le piégeur pour attaquer pose une mine) nous allons créer une class *attaque*. Cette class aura 2 constructeurs :

— Un constructeur avec 2 robots en paramètres

— Un constructeur avec 1 robot et une case en paramètre

Ainsi par exemple, le piégeur ciblera une case pour poser sa mine alors que le char ciblera un autre robot pour attaquer.

Algorithmes complexes

1 Chemin disponible à travers les obstacles

Sur la carte, seront disposés des obstacle mais avec une contrainte : toujours avoir un chemin possible entre la bas A et la base B.

```
List<case> chemin;
case c = new case();

Tant que c n'est pas sur la base oppose au depart:
        Se deplacer aleatoirement (x+1 || y+1)
        Mettre a jour c avec les nouvelles positions
        chemin.add(new case(c.x,c.y);

int nbrObstacles; //nbrObstacle ne doit pas
                  //depasser (largeur+hauteur)/2

Pour i allant de 1 a nbrObstacles:
        Tirer une case aleatoirement dans
        la hashmap de la carte

        SI case appartient a chemin:
                case.obstacle = vrai
        SINON:
                Tirer une autre case
```

2 Déplacement des robots

Les robots en fonction de leur type ne se déplacent pas de la même manière :

```
position caseActuel = robot.pos
position caseCible

SI((robot instance of Tireur || robot instance of Piegeur) &&
 caseCible.x == caseActuel.x +/- 1 ||
 caseCible.y == caseActuel.y +/- 1):

        robot.pos = caseCible

SINON SI:((robot instance of Char) &&
 caseCible.x == caseActuel.x +/- 2 XOR
 caseCible.y == caseActuel.y +/- 2):'

        robot.pos = caseCible
```

3 Historique et sauvegarde

Au cours de la partie, toutes les actions des joueurs seront stockés dans un tableau. Ainsi, lorsque le joueur demande à sauvegarder sa partie, nous créerons un fichier texte sous cette forme :

```
//en-tete
Nom de la sauvegarde
Nombre d'obstacles
Nombre de robots et type de chacun

//corps du fichier
robot realisant le deplacement -- type d'action -- case cible
4 1 c4
6 0 c1
```

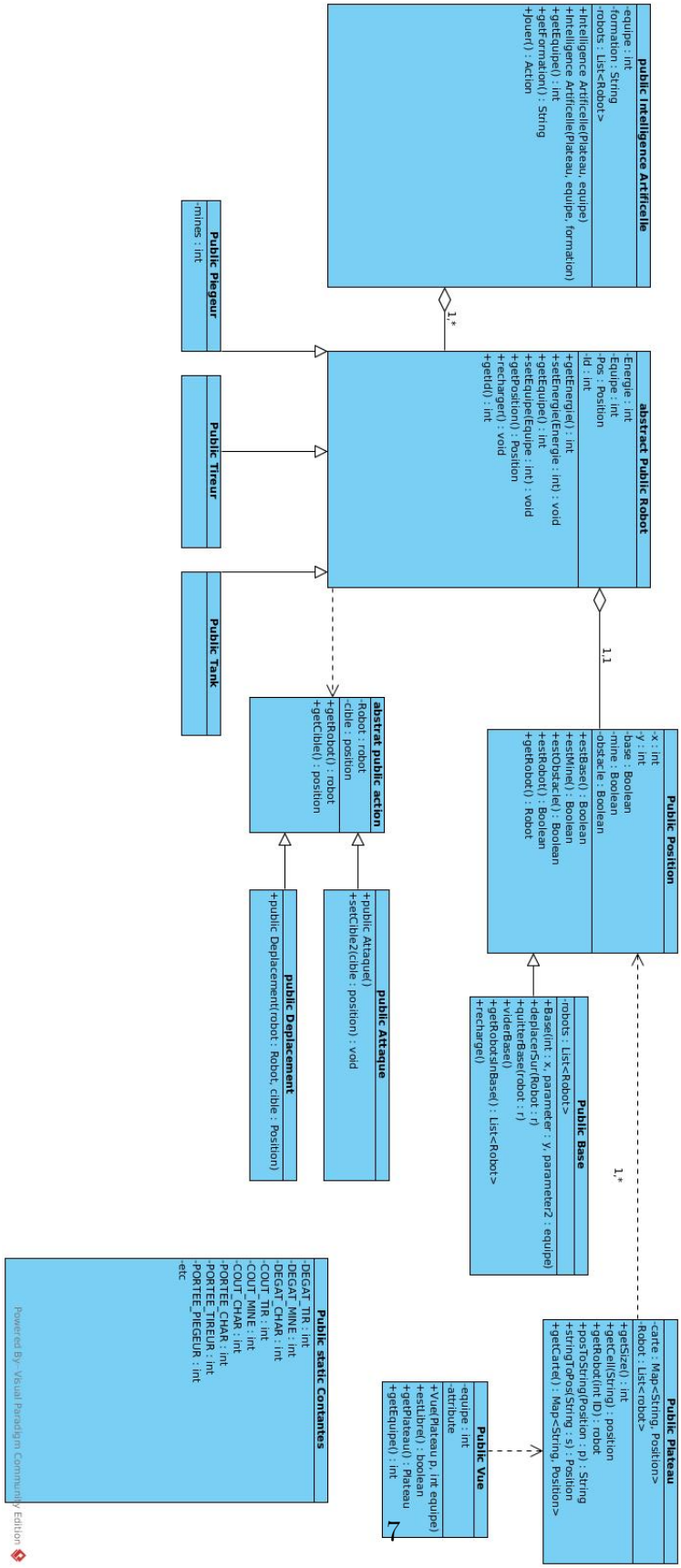
Lors du chargement, La partie sera refaite entièrement jusqu'à arrivé à la fin du fichier ce qui permettra de revenir au moment de la sauvegarde.

4 Vérifier si au moins un robot est hors de la base

Le joueur est toujours obligé d'avoir au moins un robot en dehors de sa base, sauf le premier tour ou alors si le joueur a perdu tous ses robots. Ainsi il faudra vérifier à chaque tour si au moins un robot est dehors grâce à cet algorithme :

```
1 int nbrTotalRobots;
2 Robot[] tab;
3
4 POUR(i allant de 0 a nbrTotalRobots):
5
6         SI(!tab[i].position.base):
7                 retourne vrai
8
9 retourne faux
```

UML


public Intelligence Artificielle
-equipe : int
-formation : String
-robots : List<Robot>
+Intelligence Artificielle(Plateau, equipe)
+Intelligence Artificielle(Plateau, equipe, formation)
+getEquipe() : int
+getFormation() : String
+Jouer() : Action
1.*
abstract Public Robot
-Énergie : int
-Équipe : int
-Pos : Position
-Id : int
+getEnergie() : int
+setEnergie(Énergie : int) : void
+getEquipe() : int
+setEquipe(Equipe : int) : void
+getPosition() : Position
+recharger() : void
+getId() : int
Public Piegeur
-mines : int
Public Tireur
Public Tank
1.1
Public Position
-x : int
-y : int
-base : Boolean
-mine : Boolean
-obstacle : Boolean
+estBase() : Boolean
+estMine() : Boolean
+estObstacle() : Boolean
+estRobot() : Boolean
+getRobot() : Robot
abstrat public action
-Robot : robot
-cible : position
+getRobot() : robot
+getCible() : position
public Attaque
+public Attaque()
+setCible2(cible : position) : void
public Deplacement
+public Deplacement(robot : Robot, cible : Position)
Public Base
-robots : List<Robot>
+Base(int : x, parameter : y, parameter2 : equipe)
+deplacerSur(Robot : r)
+quitterBase(robot : r)
+viderBase()
+getRobotsInBase() : List<Robot>
+recharge()
1.*
Public Plateau
-carte : Map<String, Position>
-Robot : List<robot>
+getSize() : int
+getCell(String) : position
+getRobot(int ID) : robot
+posToString(Position : p) : String
+stringToPos(String : s) : Position
+getCarte() : Map<String, Position>
Public Vue
-equipe : int
-attribute
+Vue(Plateau p, int equipe)
+estLibre() : boolean
+getPlateau() : Plateau
+getEquipe() : int
Public static Contantes
-DEGAT_TIR : int
-DEGAT_MINE : int
-DEGAT_CHAR : int
-COUT_TIR : int
-COUT_MINE : int
-COUT_CHAR : int
-PORTEE_CHAR : int
-PORTEE_TIREUR : int
-PORTEE_PIEGEUR : int
-etc
Powered By: Visual Paradigm Community Edition