université de Caen
Basse-Normandie

Méthodes de conception
Rapport de Projet

Beauchamp Aymeric 21301016
Demé Quentin 21507097
Jacqueline Martin 21507982
Zaizafoun Sami 21600538

L3 Informatique Groupe A2

Table des matières

Présentation du projet

L'application à développer est un jeu de stratégie au tour par tour. Chaque joueur peut se déplacer sur une grille de jeu et utiliser des armes pour vaincre ses adversaires, le but étant d'être le dernier joueur vivant.
Nous avons pris quelques libertés par rapport au sujet de base : chaque joueur dispose ainsi de points de vie et de points d'action, au lieu d'une simple quantité d'énergie. Les joueurs perdent lorsqu'ils n'ont plus de points de vie, et utilisent les points d'action pendant leur tour pour effectuer leurs actions (se déplacer, tirer, utiliser le bouclier...). Les points d'action sont restitués au début du tour d'un joueur.
Pendant un tour, les joueurs peuvent agir autant qu'ils le veulent, tant qu'ils ont assez de points d'action pour le faire. Le tour d'un joueur se termine quand il n'a plus de points d'actions ou quand il décide de passer sans dépenser ce qui lui reste.

1 Organisation du code

L'application est séparée en deux packages : le package modele qui contient tout ce qui est nécessaire au fonctionnement du jeu, et le package graphics qui permet l'utilisation du jeu en interface graphique.

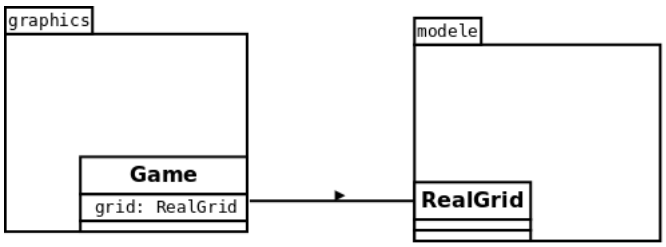

FIGURE 1 – Diagramme de packages

2 Fonctionnement du modèle

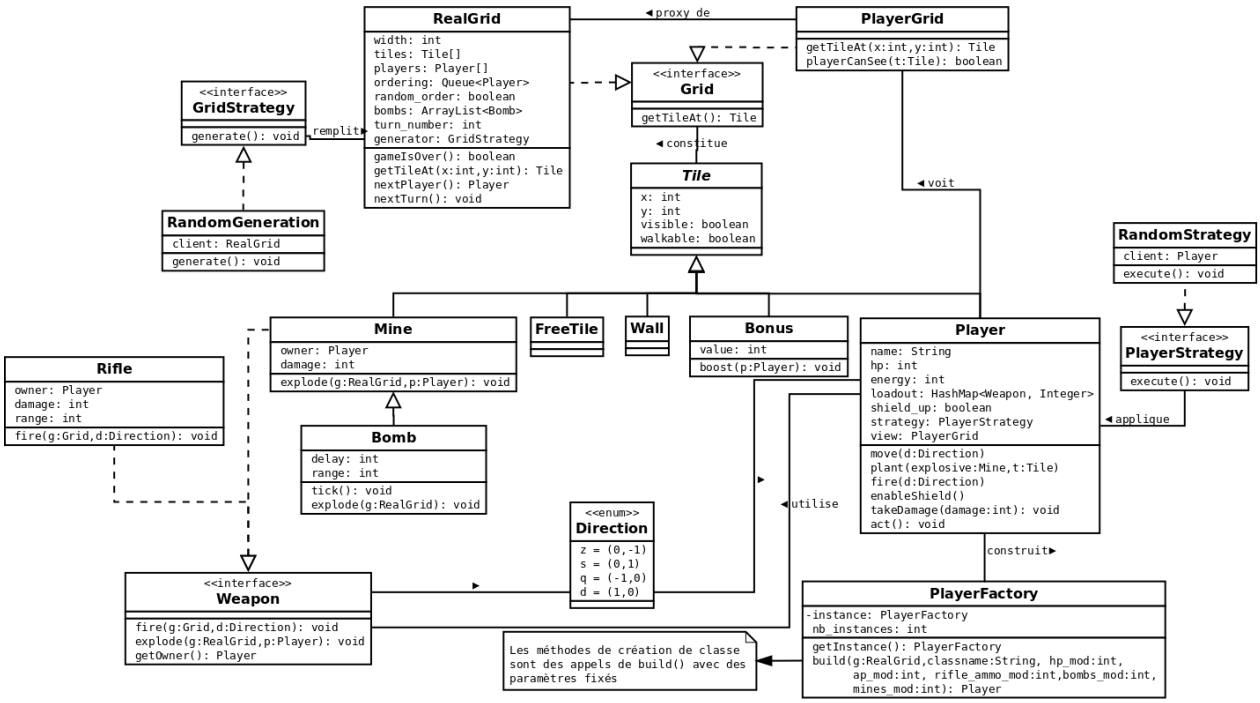

FIGURE 2 – Diagramme de classe du package modele

Le package modele permet de jouer au jeu en console, avec plusieurs options d'exécution.

— S'il est lancé sans option, on lance une partie avec 4 joueurs IA sur une grille en 10x10.
— Avec l'option **-p0**, l'application donne accès à un menu permettant de choisir les dimensions de la grille, le nombre de joueurs et le choix de la présence d'IA ou non.
— Avec l'option **-p1**, on a en plus la possibilité de choisir la classe de chaque joueur.

```
Dimensions de la grille (WxH) :
10x10
Nombre de joueurs :
4
Jeu sans humains ? (y/n)
n
Classe du joueur 1
Choisir parmi : basic, tank, marksman, engineer
basic
Classe du joueur 2
Choisir parmi : basic, tank, marksman, engineer
tank
Classe du joueur 3
Choisir parmi : basic, tank, marksman, engineer
marksman
Classe du joueur 4
Choisir parmi : basic, tank, marksman, engineer
```

FIGURE 3 – Menu avancé

2.1 Pattern factory

L'instanciation des différents joueurs se fait via la classe PlayerFactory, qui contient une méthode *build()* qui renvoie une instance de Player paramétrée par de nombreux arguments. On peut ainsi facilement créer des archétypes de joueurs en définissant des méthodes appelant *build()* avec des paramètres définis, par exemple un joueur résistant avec un bonus de points

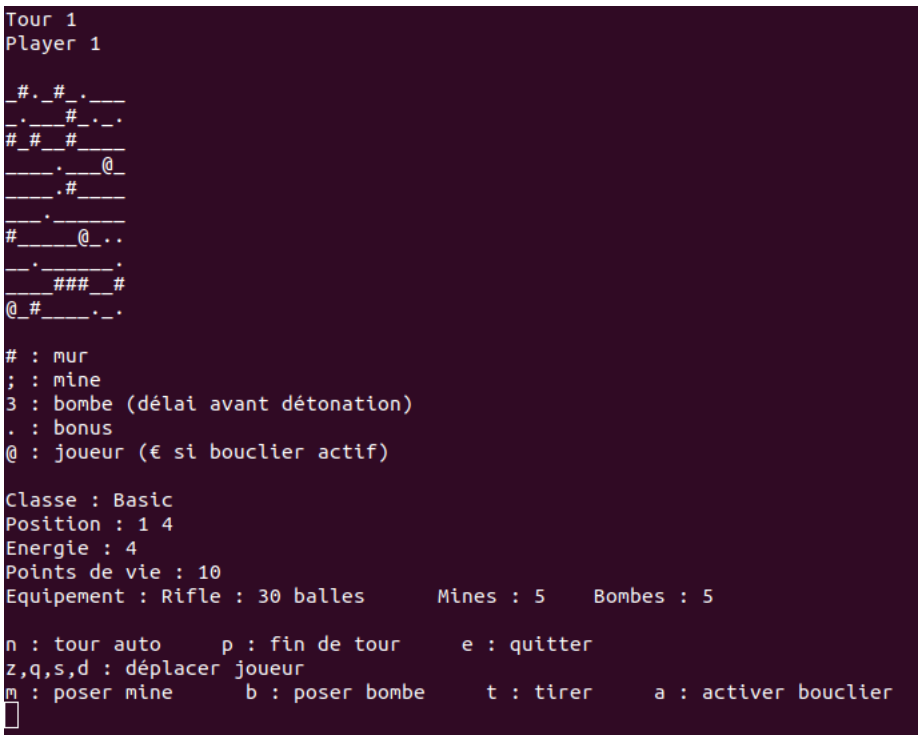

FIGURE 4 – Interface console

de vie, mais moins de munitions et d'explosifs.
Actuellement, les archétypes sont identifiée par une chaîne de caractère en attribut de Player, mais on pourrait également rendre la classe Player abstraite et utiliser la factory pour instancier des classes filles représentant ces archétypes.

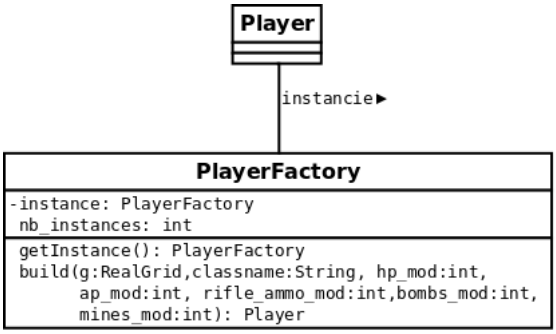

FIGURE 5 – Pattern factory pour les joueurs

2.2 Pattern proxy

Pour simplifier la mise à jour des données lorsque les joueurs agissent, il est préférable de n'avoir qu'une seule grille qui stocke toutes les informations. Cependant, on veut aussi que chaque joueur n'ait qu'une vision partielle de la grille de jeu, ne voyant que les explosifs qu'il a lui-même posés.
Le pattern proxy permet de réaliser ces deux impératifs en même temps. Dans notre application, nous utilisons une classe PlayerGrid qui est liée à la RealGrid unique de la partie, et à un des joueurs. Chaque joueur a donc un attribut PlayerGrid.
La classe PlayerGrid propose des méthodes de RealGrid modifiées pour mentir au joueur.

Ainsi, là où la méthode *getTileAt()* de RealGrid renvoie simplement la case se trouvant aux coordonnées demandées, la même méthode de PlayerGrid vérifie d'abord si le joueur a le droit de connaître la nature exacte de la case, et renvoie une case vide si ce n'est pas le cas. La méthode *toString()* fonctionne de la même façon et permet de produire des représentations de la grille adaptées à chaque joueur.

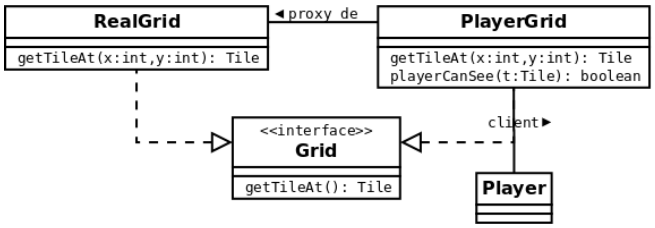

FIGURE 6 – Pattern proxy pour les vues joueur

2.3 Pattern strategy

Nous utilisons le pattern Strategy deux fois dans l'application : pour la génération de la grille et pour le comportement de l'IA.
Ce pattern permet de modifier radicalement le résultat d'une méthode sans pour autant modifier le coeur du code.
Ainsi, la génération de grille est faite en utilisant la méthode *generate()* d'une implémentation de l'interface GridStrategy. Si on veut changer la façon dont on remplit la grille, il suffit d'ajouter une nouvelle implémentation et de faire appel à un simple setter de RealGrid.

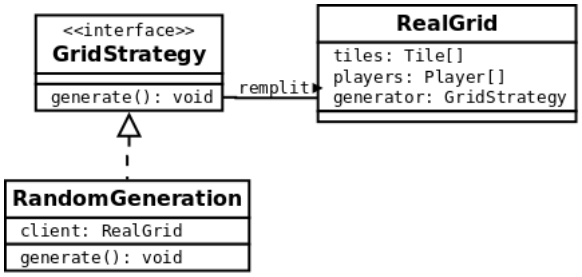

FIGURE 7 – Pattern strategy pour la génération de grille

3 L'interface graphique

3.1 La classe ImagesLoader

La classe ImagesLoader est une classe s'occupant de gérer le chargement des images et de facilité leur utilisation au sein du package *graphics*. Nous allons quelque peu détailler son fonctionnement.

3.1.1 Découpage du tileset

Les images sont découpées au sein d'une même image d'ensemble : *le titeset*. Le découpage de ce tileset se fait du coin en haut à gauche, jusqu'au coin inférieur droit. Cette ordre a son importance puisque la position de l'image dans la liste correspond à l'index récupéré dans le XML moins un.

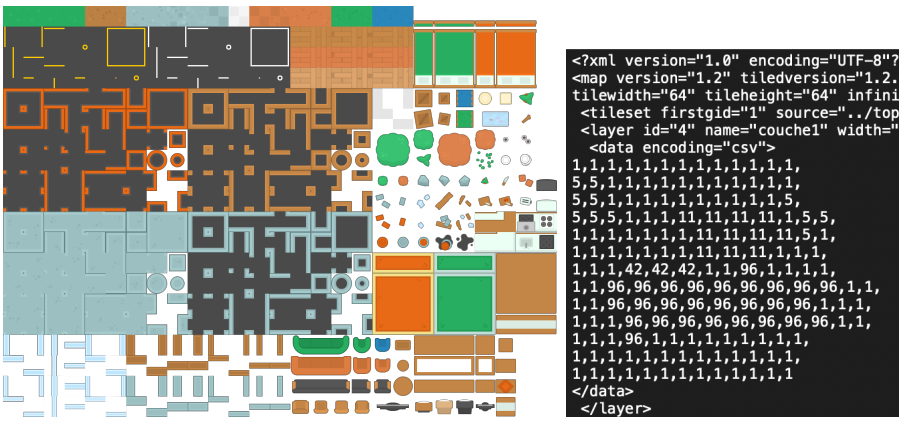

FIGURE 8 – Mise en parallèle du XML et du tileset

Comme on peut le voir sur la figure ci-dessus, les indices 1 correspondent à l'image 0 du tileset qui est l'image en haut à gauche de ce dernier.

3.1.2 Utilisation

Cette classe a pour but de charger les images nécessaires à l'affichage du jeu. Ces ressources sont créées comme étant *static* pour être accessible de n'importe où sans passer par des getters. Le chargement des images nécessite tout de même un appel en début de programme de la méthode loadImages(). Ce fichier regroupe les images :

- Des entités d'environnement du jeu (ArrayList)
- Des personnages (HashMap<Integer, ArrayList>)
- Du bouclier
- Des mines, et des bombes
- Des bonus

Ces images sont utilisées pour créer les objets de classe *Tile*. En effet, un *Tile* est créée avec ses coordonnées x et y et une image étant sa représentation graphique.

3.2 La classe View

La classe view est la classe permettant l'affichage du model en vue graphique. Le code possède également une classe ViewConsole qui réalise un affichage console en permettant de jouer.

3.2.1 Constructeur et principe de fonctionnement

Lorsque la vue est créée, on renseigne pour le constructeur :

- Le model à écouter
- Le joueur propriétaire de la vue
- La liste des entités
- la JFrame contenant la vue : l'observer

3.2.2 Fonctionnement, affichage et actualisation.

Lors de l'affichage, la vue parcourt la liste d'entités dans un ordre défini. Cet ordre représente notamment les couches (layers) contenues dans le fichier XML permettant de charger le niveau. L'ordre de cette liste est très important puisque l'on vient superposer les images les unes aux autres pour ajouter du détail, comme par exemple les fauteuils, tables etc.

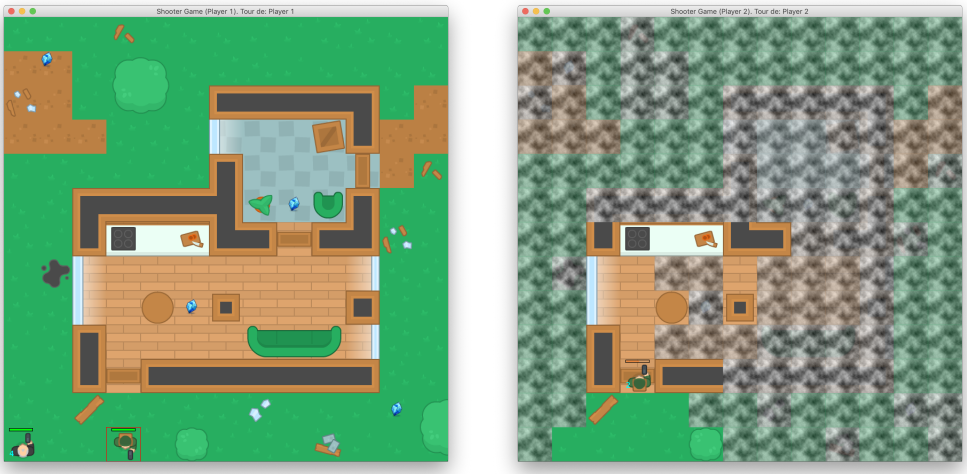

FIGURE 9 – L'interface avec et sans brouillard

Lorsque le model change, l'appel de la méthode *stateChange()* provoque une actualisation de toutes les vues écoutant le model.

3.2.3 Un élément optionnel : l'animation de tir

Lorsqu'un joueur tir, une animation se déclenche permettant de visualiser le parcours de sa balle. Cette animation est réalisée par le biais d'un Thread actualisant deux valeurs x et y tant que la portée maximale n'a pas été atteinte. A chaque actualisation, on prévient la vue pour qu'elle puisse se repeindre. La classe permettant cette animation est la classe *ThreadPlay*.

3.3 La classe GUI

La classe GUI est la classe permettant de créer une fenêtre accueillant la vue d'un joueur.

3.3.1 Constructeur et principe du fonctionnement

Lors de sa création cette classe reçoit en argument :
— Le modèle
— Le joueur à qui la vue devra appartenir
La vue est ensuite créée au sein du constructeur.

3.3.2 La jouabilité

L'application est jouable au clique et au clavier. En effet pour chaque action le joueur doit d'abord cliquer sur son personnage pour faire afficher un menu déroulant de ses actions possibles. Toutefois, si le joueur souhaite effectuer plusieurs déplacements à la suite.

3.3.3 Entrées claviers

Pour lire les entrées claviers, on doit dans un premier temps invoquer la méthode *setFocusable(true)* puis la méthode *requestFocus()*. Puis ajouter un keyListener a la fenêtre.

3.3.4 Entrées souris

La majorité des actions se font grâce aux cliques. Pour cela, on ajoute un *mouseListener* à la fenêtre. Ensuite on agit en conséquence lorsque l'on reçoit un clique.

3.4 Player

3.4.1 line of sight : visiblesTiles

Cette fonction est originellement utilisée comme test pour la ligne de vue des joueurs. Son principe est la base du suivant. Il test uniquement les directions cardinales dans quatre boucles *while* qui s'arrêtent si elles rencontrent un mur ou si elles sont arrivées à la limite de taille de vue (*visionSize*).

3.4.2 line of sight : testView

Dans *testView()* nous testons si la case indiqué est visible par le joueur, pour cela, nous considérons une grille temporaire composée de 0, nous allons ensuite calculer la distance en case qui sépare nos deux points, puis les vecteurs, nous décomposeront ce vecteur en plusieurs que nous appliquerons 1 par 1, a chaque application, nous approchons du point testé, et a chaque avancé, nous ajoutons +1 a la grille, dans la case équivalent à la où nous nous situons, cela permet de connaître les cases parcourues (voir schéma) si dans la boucle des cases parcourues, nous observons une Tile qui a sont attribut *isWalkable* à *faux*, alors la fonction *testView* est arretée. Dans le cas contraire, si elle fini la boucle, alors nous sommes bien arrivé à la case testée, nous pouvons donc l'ajouter dans la liste des cases visibles.

3.4.3 line of sight : visibleTiles

La fonction *visiblesTiles()* quand à elle fait deux choses, en premier lieu elle appelle *testView* pour chaque case, ensuite, elle teste tout les murs afin de savoir si une case adjacente est visible, si c'est le cas alors le mur sélectionné le deviens aussi, cela permet au joueur de comprendre ce qui bloque son champs de vision.

3.4.4 line of sight : notes

Dans la gestion de la ligne de vue, nous utilisons finalement les deux fonctions renvoyant les Tuiles visibles, *visibleTiles* et *visiblesTiles* pour éviter toute possibilité de bug dans les zones proches dûes à la gestion des lignes, celle-ci pouvant sembler anormale pour l'humain, mais pour autant mathématiquement exacte.

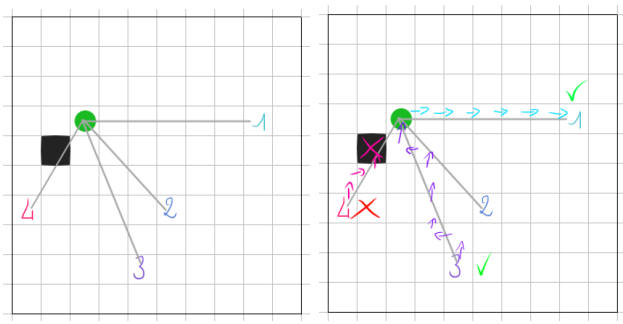

FIGURE 10 – exemple de fonctionnement de plusieurs *testView*