MASTER 2 - Semestre 1 - 2008

Architecture logicielle
Bomberman

SELLIER Xavier
PHOMMARINH Phetsana
RADANIELINA Andria Johary

Chargé de TD : D. Cassou

Avant de commencer le rapport du projet, voici l'usage du jeu.

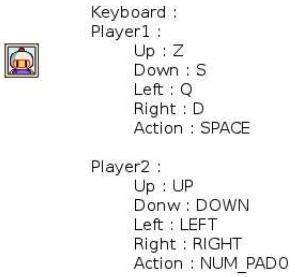

Fig. 1 – Touches du clavier

1 Bomberman

Le jeu que nous allons essayer d'implémenter est le Bomberman, le poseur de bombes. Le but de ce jeu est d'éliminer son adversaire avec les bombes à notre disposition ou user de ruse pour que l'ennemi tombe sous ses propres bombes.

Tout d'abord, listons les entités du jeu dont nous aurons besoin. Ceci est une liste non exhaustive qui représente les entités et leurs caractéristiques que l'on peut trouver en général dans les Bomberman.

1.1 Composition minimal

1.1.1 Joueur

Entité centrale du jeu, le joueur qui est représenté par un avatar qui peut se déplacer sur la carte et avoir certaines actions dont le posage de bombes. Un joueur en déplacement est bloqué par les murs qui composent le jeu.

1.1.2 Bombe

L'autre pièce maîtresse du jeu, la bombe une fois posée, explose après un certains laps de temps et on a une expansion de flamme. Elle peut posséder une facteur de puissance.

1.1.3 Flamme

Une entité qui est assez abstraite, elle est la représentation d'une explosion de bombe. Suivant l'implémentation choisie, on pourrait voir la flamme comme une entité invisible, le tout étant qu'il faille qu'elle intéragisse avec l'environnement après une explosion de bombe.

1.2 Niveau

Tout jeu possède des niveaux, ceux des Bomberman sont caractérisés par les deux types de murs suivant.

1.2.1 Murs indestructibles

Ces murs ne peuvent être traversés. Elle délimite le niveau du jeu et sont insensibles aux explosions des bombes.

1.2.2 Murs destructibles

Ensuite nous avons les murs destructibles qui s'éffritent sous les vagues des bombes.

1.3 Objets

Les Bomberman de maintenant possèdent des objets qui ajoutent de nouvelles capacités à l'avatar du joueur mais ce n'est pas obligatoire, un bomberman pourrait très bien ne pas comporter de bonus, le minimum étant qu'il puisse poser des bombes explosives.

Voici, une liste des principaux objets que contient un Bomberman.

1.3.1 Bombe

Un Bomberman peut généralement poser une seule bombe à la fois mais cet objet lui permet d'augmenter sa capacité maximale.

1.3.2 Flamme

Un autre objet bonus classique des Bomberman est la flamme qui permet d'augmenter la puissance d'explosion d'une bombe.

1.3.3 Autres

Ensuite, nous pouvons trouver divers objets qui ajoutent des nouvelles capacités au joueur ou tout simplement augmentent ou décroient les caractéristiques de son avatar.

On pourra par exemple citer les bottes de vitesse qui augmentent la vitesse de déplacement du joueur, les gants de fer qui lui permettent de pousser les bombes. Dans les objets qui apportent un malus, on a celui qui donne un contrôle aléatoire de son avatar ou ceux qui baissent le nombre total de bombes. Il existe une multitude d'objets bonus qui sont laissés libre à l'imagination du joueur.

1.4 Intéractions

Essayons de lister les intéractions dont nous aurons besoins afin de faire marcher notre jeu. On peut voir une première représentation des différentes intéractions entre les entités sur la figure 2.

2 Architecture

Pour faire notre Bomberman, nous avons à notre disposition un « framework » de jeu.

	Bomberman	Bombe	Flamme	Murs indestructibles	Murs destructibles	Objets bonus
Bomberman	Un joueur peut ou pas traverser un autre joueur suivant le choix de l'implémentation	La bombe bloque le joueur une fois posée	La flamme tue le joueur	Ce mur bloque le déplacement du joueur	Ce mur bloque le déplacement du joueur	Les objets affecte les caractéristiques du joueur
Bombe		Un bombe ne peut être posée sur une autre bombe	La flamme active l'explosion de la bombe	La bombe ne peut être posée sur ce type d'entité	La bombe ne peut être posée sur ce type d'entité	La Bombe peut être posée sur un objet bonus
Flamme			La flamme n'a pas d'effet sur une autre flamme	La flamme n'affecte pas ce type de mur	La flamme détruit le mur	La flamme détruit les objets bonus
Murs indestructibles						
Murs destructibles						Une fois détruit ce type de mur peut faire apparaitre des objets bonus
Objets bonus						

FIG. 2 – Schéma d'intéraction entre les entités

2.0.1 Framework

Le « framework » se décompose en plusieurs parties, chacune gérant une activité. Nous allons le voir avec l'exemple du jeu Pacman fourni.

Légende :
- Noir : Classes du « framework »
- Bleu : Classes de l'API Java
- Rouge : Classes spécifiques au jeu

Gestion du jeu Cette partie est le lien avec l'utilisateur, elle permet de lancer le jeu.

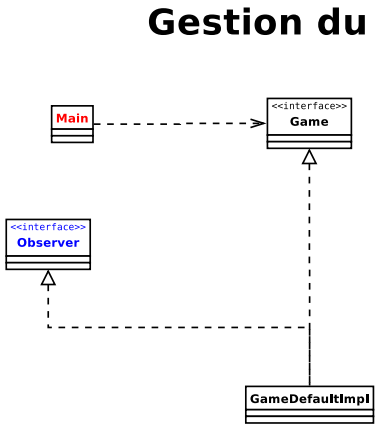

FIG. 3 – Gestion du jeu

Gestion du niveau La composition d'un niveau de jeu est gérée par ces classes. Elles définissent entre autres les entités présentes, comment sont placées les entités par rapport aux autres sur la carte.

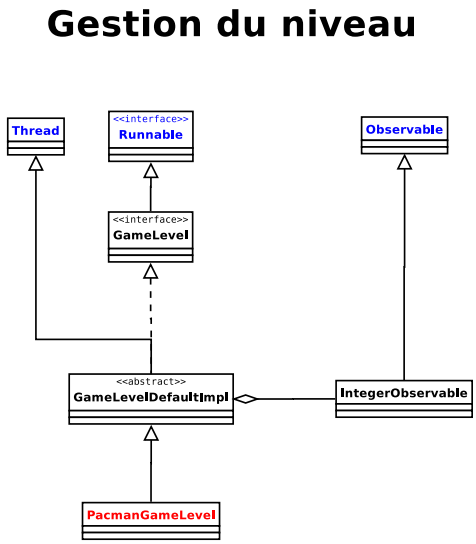

FIG. 4 – Gestion d'un niveau

Gestion de l'environnement d'un niveau Là où la gestion du niveau s'occupait du placement des entités, cette partie s'occupe de tout ce qu'il y a derrière, c'est à dire qu'elle se charge de permettre une intéraction entre les entités du jeu, que les entités sont bien transmis à la partie qui gère l'affichage, etc. . .

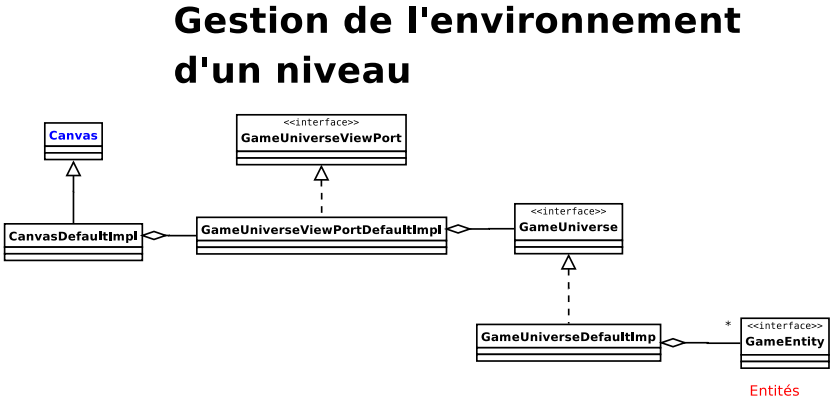

FIG. 5 – Gestion de l'environnement d'un niveau

Gestion des déplacements Nous avons ensuite, une partie qui gère la façon dont vont se déplacer les entités dans le jeu.

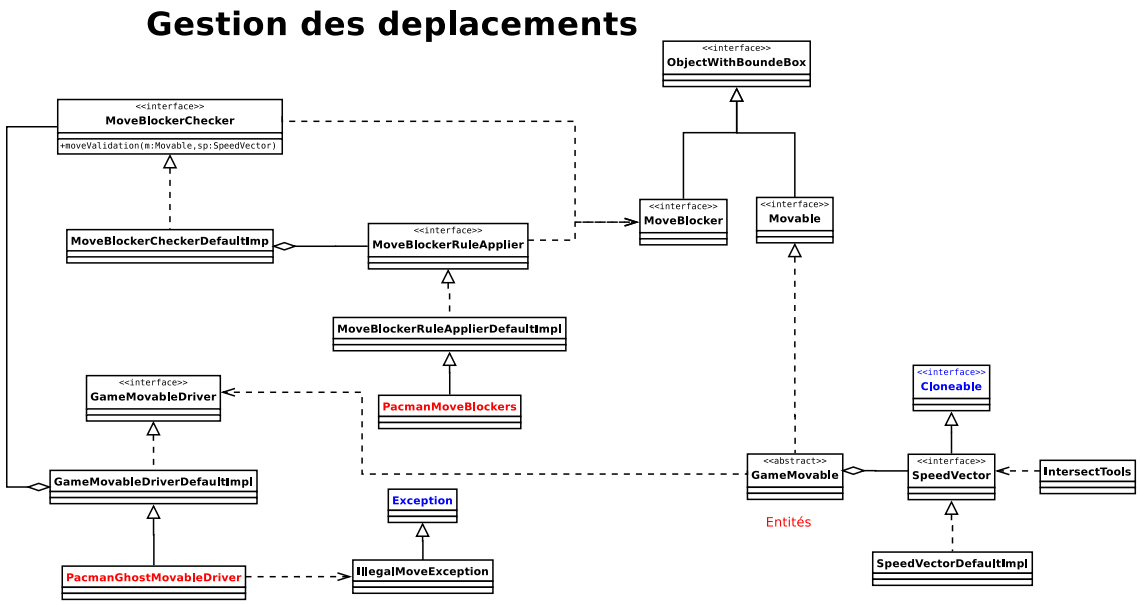

Fig. 6 – Gestion des déplacements

Gestion des stratégies de déplacement Ici, nous avons une forme de spécialisation de la partie précedente. On va définir des stratégies de déplacement à suivre pour les entités qui peuvent être controlées par l'IA ou par le joueur via le clavier.

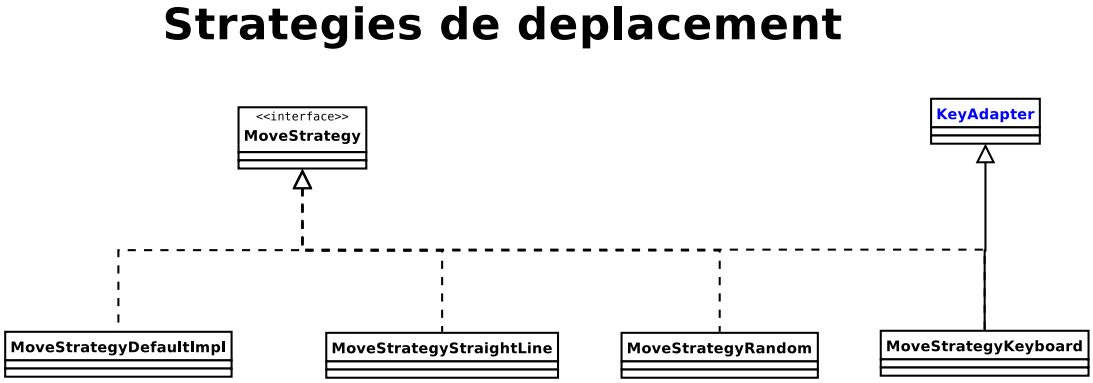

Fig. 7 – Gestion des stratégies de déplacement

Gestion des collisions Voici, la partie qui est spécialisée dans la gestion des collisions. Elle définit les règles de collisions, de ce qu'ils se passent lorsqu'une collision se produit entre deux entités du jeu.

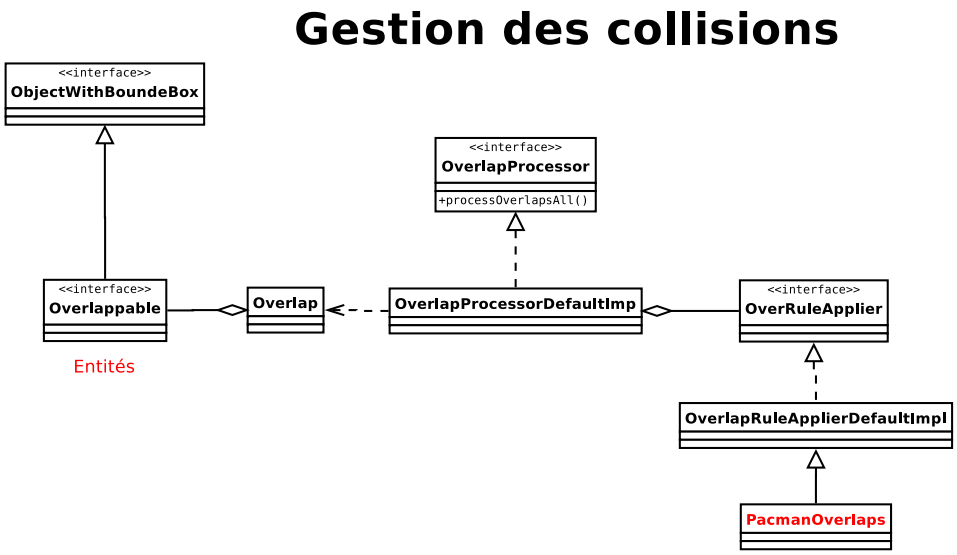

FIG. 8 – Gestion des collisions

Gestions des dessins Pour en terminer avec les classes du « framework », il reste la partie qui se charge de ce qu'on peut voir sur notre écran de jeu, c'est à dire ce qui est déssinable.

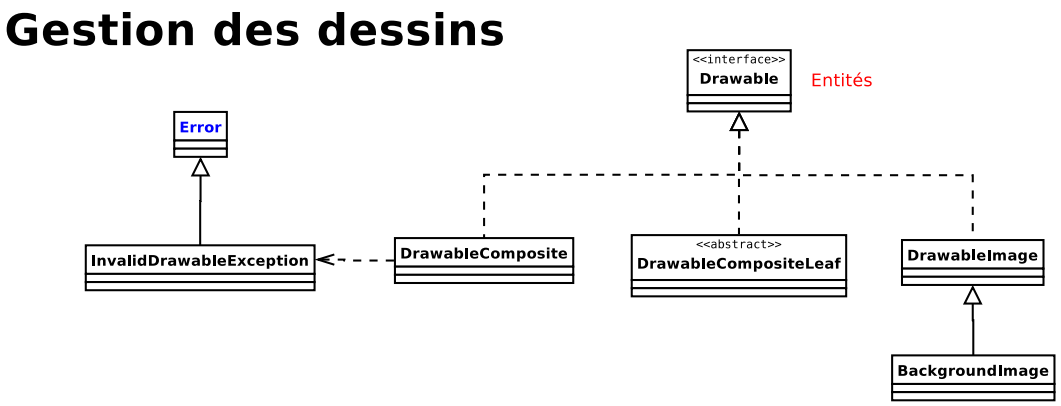

FIG. 9 – Gestion des dessins

Classes spécifiques au jeu Ensuite nous avons les classes qui font que le jeu est le jeu. Ces classes définissent chacune des entités. Voici un exemple sur la figure 10 avec celui du jeu Pacman.

On peut voir sur la figure 11 une architecture globale simplifiée.

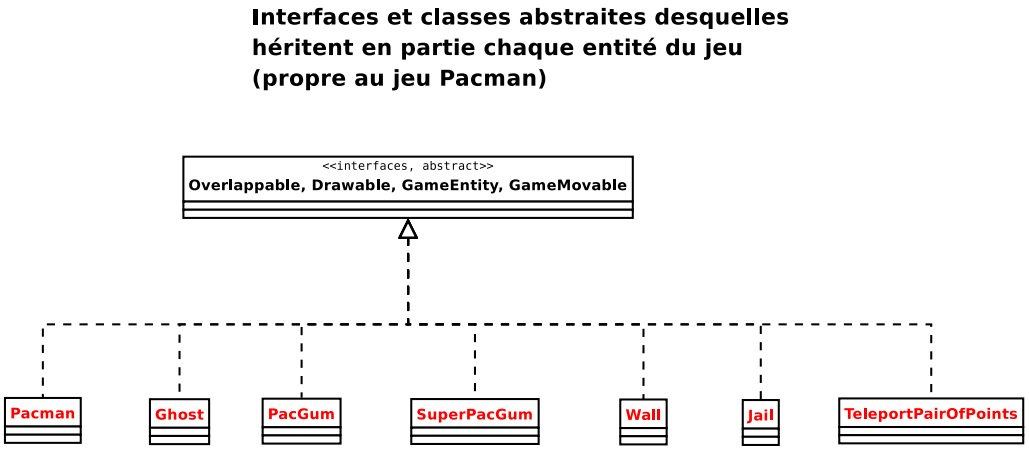

FIG. 10 – Entités du jeu Pacman

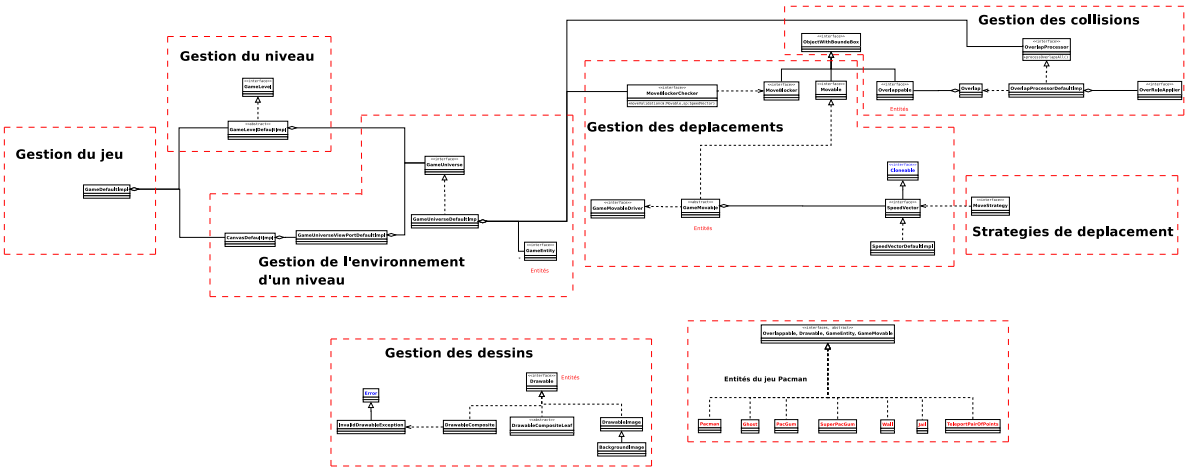

FIG. 11 – Architecture globale simplifiée

3 Implémentation d'un Bomberman

En utilisant ce « framework », nous allons créer le jeu Bomberman.

3.1 Extension du « framework »

Tout d'abord, nous avons choisi de ne pas toucher au code source du « framework » donné. Toutes les modifications nécéssaire se feront sous forme d'extension, le plus souvent par un simple héritage de la classe que l'on veut modifier. Les extensions se trouvent dans les paquetages « bomberman.game » et « bomberman.base » qui extendent respectivement les paquetages « framework.game » et « framework.base ».

Pour un soucis de confort, nous avons décide d'extendre la classe *GameDefaultImpl*. Cette classe gère la taille du panneau Java et la taille de la grille. La méthode dont nous avions

besoin de surcharger était *createGUI*, cette dernière faisait appel à deux méthodes privées que nous avons dû dupliquer dans cette classe. Les modifications faites vont nous permettre de lancer notre jeu sur une grille de taille et une résolution arbitraire.

Ensuite, l'autre modification que nous avons appliqué était la gestion du clavier. Pour se faire, nous avons étendu la classe *MoveStrategyKeyboard*.

Tout d'abord, les modifications dans cette classe va permettre de changer la stratégie du clavier qui sur Pacman était une avance automatique, c'est à dire que lorsqu'on lui indique une direction, l'avatar continue jusqu'à rencontrer un obstacle.

Nous avons au passage utilisé le modèle de conception « Méthode de fabrication » , que l'on peut voir sur la figure 12, afin de gérer les claviers des joueurs. On ne connait pas à l'avance les touches du clavier qui seront associées au déplacement et à l'action mais chaque avatar des joueurs se comporteront de la même façon.

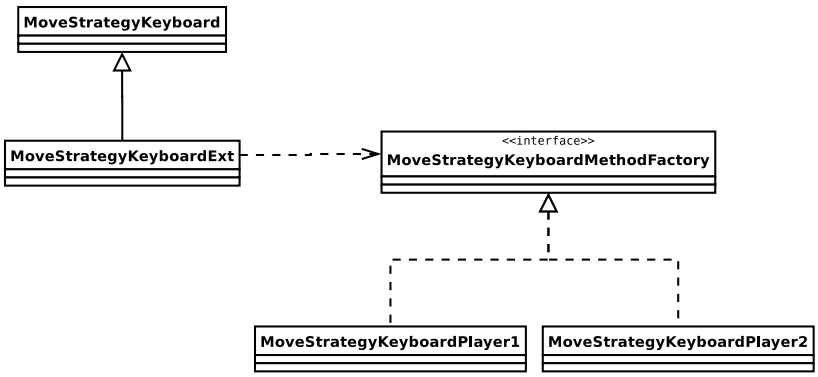

FIG. 12 – Gestion du clavier

3.2 Refactoring

Le code des entités du jeu Pacman possèdait des variables de classes tels que *SPRITE_SIZE* déclarées dans plusieurs classes ce qui n'est pas bon pour la maintenabilité. Nous avons crée une classe *ConstantValues* qui contient les diverses constantes du jeu.

3.3 Paquetages

Sans avoir une reflexion très poussée sur le sujet, plusieurs variables dans les classes des entités du jeu de Pacman était déclarées en « protected », il n'y avait pas vraiment d'encapsulation.

Nous avons crée une forme de hiérarchie que l'on peut voir sur la figure 13. Nous avons regroupé les entités par rôle dans le jeu. Ainsi dans le paquetage *bomberman.entity.item*, nous trouverons tous les objets bonus. On peut noter le paquetage *utility* qui contient la classe qui se charge des chargements des images.

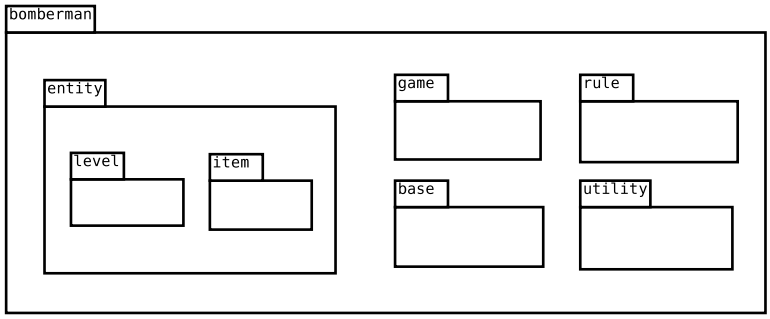

Fig. 13 – Paquetages

3.4 Architecture du jeu

Nous allons présenté une architecture que nous avions implémentée dans un premier temps mais qui n'est pas forcément l'architecture finale. Nous expliquerons les raisons de ces changements.

3.4.1 Timer

Avant de commencer à expliquer notre architecture, présentons la classe *Timer* du paquetage *java.util* fourni par l'API Java. Cette classe nous a été d'un grand secours, elle nous permet entre autres de gérer l'animation des « sprites ».

3.4.2 Entités principales

La figure 14 montre les trois principales entités du jeu avec les principales variables et méthodes.

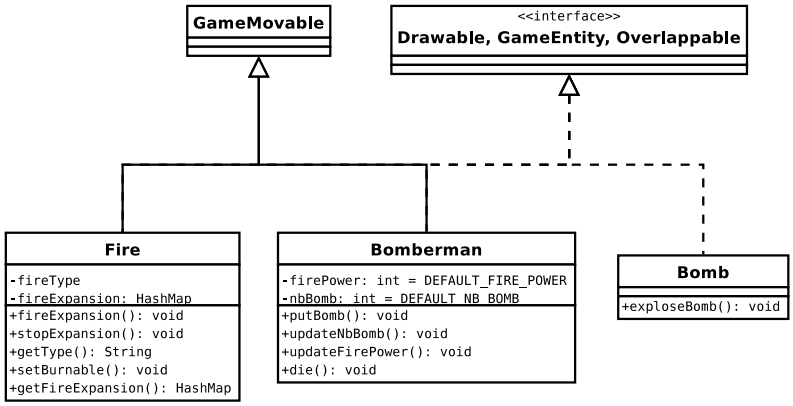

Fig. 14 – Entités principales

Bomberman Les noms des méthodes sont assez explicites pour en rajouter sur ce qu'elles font.

Fire Nous avons considéré cet entité comme une entité « Movable », ce qui va nous permettre de gérer les collisions avec les autres entités.

La principale fonction à noter est *fireExpansion* qui fait le calcul de l'expansion lors d'une explosion.

Bomb Dernière des trois entités, il n'y a pas grand chose à dire sur cette entité si ce n'est que la méthode *exploseBomb* déclenche l'explosion de la bombe.

Pour gérér le compte à rebours de la bombe, nous avons utilisé la classe *Timer*.

3.4.3 Objets

La gestion des objets bonus est représentée par la figure 15. La classe abstraite *AbstractItem* factorise le code de la combustion d'un objet qui se produit au contact de flamme en mettant la variable *isActive* à faux.

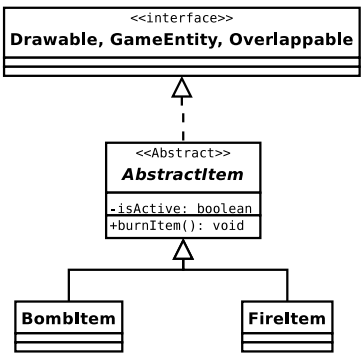

Fig. 15 – Objets bonus principaux

3.4.4 Niveau

Ce groupe contient, les éléments du décor. Nous avons l'entité *BlocAround* qui est en fait une *SuperWall* avec une représentation différente, nous aurions pu éviter de faire une classe pour cet entité et chercher à utiliser un modèle de conception ou tout simplement avoir en argument au constructeur de la classe l'image à afficher. Mais nous avons fait simple et cela nous aide à nous représenter l'entité dans le jeu. Les *BlocAround* sont les murs qui délimitent le jeu sur les extérieurs.

Les entités *BlocAround*, *SuperWall* et *Wall* sont de type *MoveBlocker*, elles ne peuvent être traversées. En plus de cela, l'entité *Wall* est aussi de type *Overlappable*, ce qui permet de gérer la collision avec une flamme.

Pour les besoins de notre jeu, nous avons intégrer l'entité *Floor* qui va nous permettre de calculer l'expansion d'une explosion.

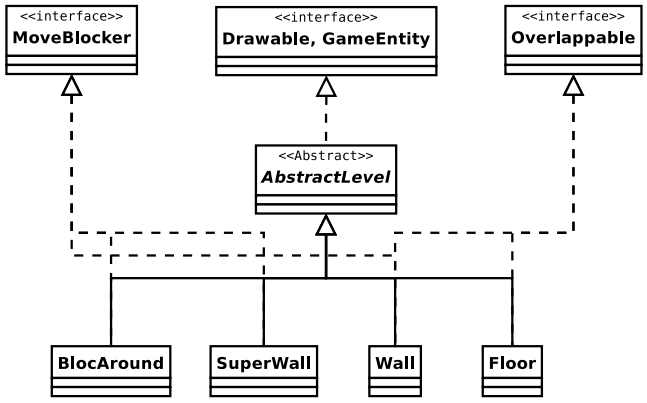

FIG. 16 – Principales entités du décor

3.4.5 Pré-architecture globale

Tout ceci donne l'architecture figure 17.

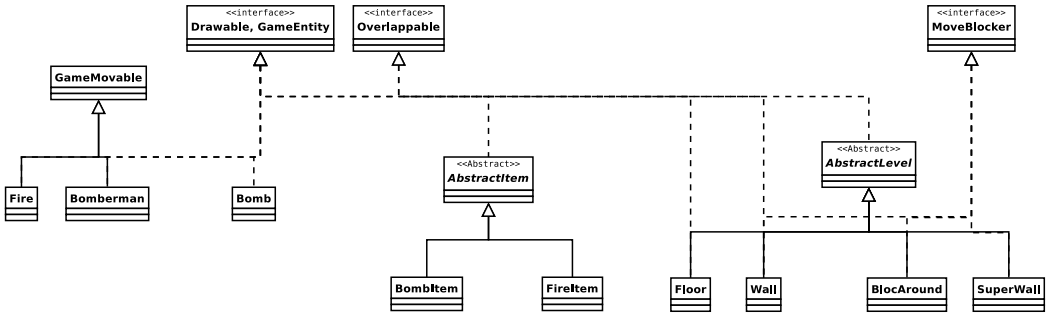

FIG. 17 – Première architecture

4 Difficultés rencontrées

La création d'un jeu avec un « framework » n'est pas toujours évident, il faut essayer de s'imprégner de ce qui se fait et respecter les rôles définis par chaque groupe de gestion.

4.1 Posage de bombes

La première difficulté que nous avons rencontré et qui nous as demandé un gros temps de travail dessus est le posage de bombes. Sur le Pacman, toutes les entités étaient ajoutées à l'univers avant que la partie ne commence réellement par l'usage de la méthode *addGameEntity*.

En essayant de poser des bombes, il arrivait que le jeu levait des exceptions du type :

```
Exception in thread "Thread-2" java.util.ConcurrentModificationException
at java.util.AbstractListItr.checkForComodification(UnknownSource)
atjava.util.AbstractListItr.next(Unknown Source)
at gameframework.game.Game...Impl.allOneStepMoves(Game...Impl.java :44)
at gameframework.game.Game...Impl.run(Game...Impl.java :46)
```

ou encore

```
Exception in thread "Thread-2" java.util.ConcurrentModificationException
at java.util.AbstractList$Itr.checkForComodification(Unknown Source)
at java.util.AbstractList$Itr.next(Unknown Source)
at bomberman.game.Game...ImplExt.paint(Game...ImplExt.java :40)
at gameframework.game.Game...Impl.run(Game...Impl.java :45)
```

Ce genre d'exception est un problème récurrent de la programmation avec les « threads ».

Nous avons un thread qui s'occupe du mise à jour de ce qu'on peut voir dans l'écran, le « framework » utilise un itérateur pour manipuler les entités. Or un itérateur ne permet pas d'accès concurrent, ce qui lève l'exception précédent.

En ayant conscience de cela, on peut voir que la boucle de jeu se trouve dans la classe *GameLevelDefaultImpl* et qu'elle boucle sur principalement trois fonctions :

- *paint*
- *allOneStepMoves*
- *processAllOverlaps*

Le jeu Pacman tournant correctement, on peut supposer que si l'on essaye de faire un *addGameEntity* dans l'une de ces trois fonctions, la stabilité du jeu n'est pas assurée.

Une solution auquelle nous avons pensé est de faire une fonction qui sera appelée à la suite des ces trois fonctions. Cette fonction s'occuperait de l'ajout des entités que l'on souhaite. Pour implémenter notre solution, nous avons utilisé le patron de conception « Observateur ». Nous allons surveiller les appels aux méthodes *addGameEntity* et *removeGameEntity*.
Lorsque l'on souhaite ajouter une entité dans l'univers du jeu, est placé dans une file, une fois les trois précédentes fonctions terminées, on se charge de vider la file, c'est à dire ajouter réellement dans l'univers l'entité.

La classe *GameLevelDefaultImpl* fait partie du « framework » ce qui ne nous permet pas de modification mais la classe *BombermanGameLevel* étend la précédente classe, ce qui va nous permettre de surcharger la méthode qui contient la boucle de jeu.

La figure 18 donne une schématisation de l'implémentation.

Nous sommes conscients que nous forçons le comportement de la bombe à être de type *Movable*. Nous avons voulu factoriser le code de l'ajout et suppresion d'entité au sein de l'univers. La classe abstraite *AbstractEntity* va permettre aux trois entités d'avoir accès à une instance de la classe *EntityOperation* qui est observée par la classe *BombermanGameLevel*,

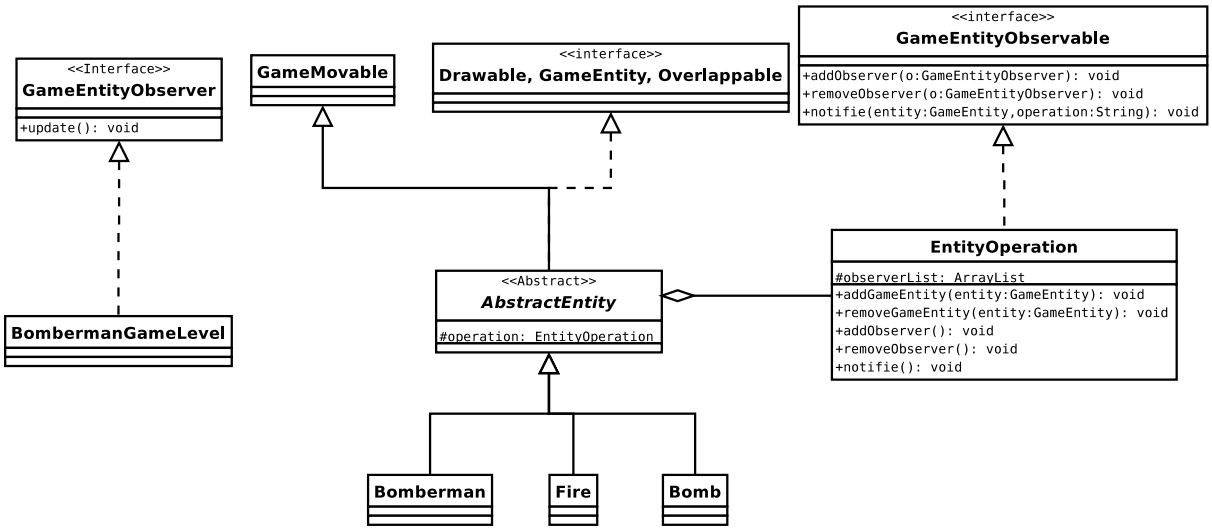

FIG. 18 – Observateur

qui au passage est maintenant aussi un observateur.

Précédemment, lorsque nous avions un « universe.addGameEntity(e) », nous allons maintenant faire appel à la méthode *addGameEntity* de la classe *EntityOperation*. Ce qui donne donc « operation.addGameEntity(e) ».

La classe n'ajoute pas directement à l'univers l'entité, elle se charge de transmettre à l'observateur l'entité qui va la mettre dans une file en attendant consommation de celle-ci.

4.2 Calcul des expansions

Un autre problème qui s'est posé est le calcul des expansions des explosions. Avant tout, présentons notre solution.

Lorsqu'une bombe est posée, elle va exploser après un laps de temps. Cette explosion va générer une première flamme le « centre » qui va ensuite commencer l'expansion sur les quatres points cardinaux. Quatres autres flammes sont alors créées, chaque flamme va tester d'abord si elle peut s'afficher ou dit autrement si elle peut brûler. Ainsi, on n'affichera rien sur les murs dits « indestructibles ».

L'expansion de flamme se fait grâce à l'entité *Floor* qui indique que c'est une passage vide et donc qu'elle peut continuer, tout ceci jusqu'à ce que la puissance de la flamme s'estompe ou que cette dernière soit arrêtée par un mur. Pour cela, on maintient une table d'association qui va indiquer si on peut ou pas continuer l'expansion. Lorsque la flamme rencontrait un mur destructible, elle mettait à jour la table.

Le problème que nous rencontrions est que dans l'ordre de tests des collisions entre entités, la flamme faisait d'abord le test avec l'entité *Floor* avant de le faire sur l'entité *Wall*. Ce qui

fait qu'elle mettait la table d'association un cycle trop tard, donc ne s'arrêtait pas au première obstable mais continuait à « +1 ».

Une solution que nous avons trouvé pour reglér ce problème qui n'en est pas vraiment une et qui s'apparente plus à du bricolage est de passer la classe *Wall* en *Movable* et mettre de mettre la règle de collision correspondante. Ce qui donne un comportement qui n'est pas vraiment pertinent, un mur peut-il se déplacer ? Une solution qui marche mais non satisfaisante en terme d'architecture logicielle.

4.3 Architecture finale

Toutes ces modications nous le diagramme figure 19.

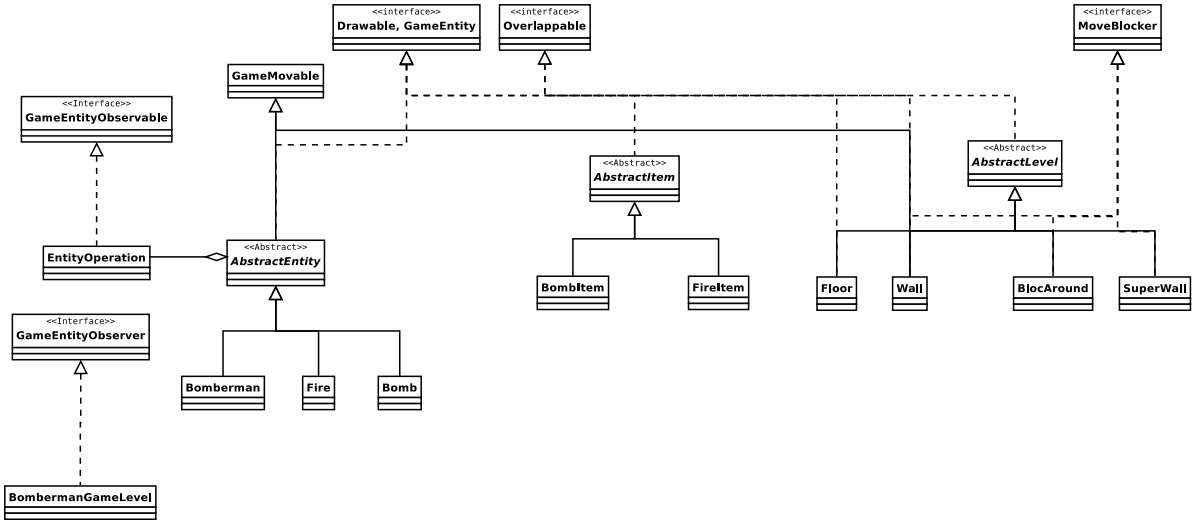

Fig. 19 – Architecture finale

5 Bilan

Dans la finalité, nous avons un jeu qui semble marcher malgré que sur la fin l'architecture proposée ne nous satisfait pas. Une grosse partie du temps consacré à ce projet aura été sur la compréhension de l'exception levée qui est décrite plus haut. Dans un premier temps, nous étions passés par le système des « synchronized » et consorts que propose l'API Java. Cela ne résolvait pas le problème mais réduisait les exceptions.

Comme le jeu n'était pas stable (problème avec les « threads » et itérateurs) et que la solution de calcul des expansions n'a été résolu que peu de temps avant la « deadline ». Nous n'avons pas pu intégrer des tests unitaires afin de valider le comportement du jeu en vue de le maintenir.

5.1 « TODO »

Comme dit plus haut, il faudrait établir une batterie de tests unitaires. Ensuite, il y a un travail à faire dans tout ce qui est des menus et de l'environnement lorsque le joueur gagne une partie, veut mettre en pause, etc...

Au niveau du jeu, nous avons le comportement minimal, on pourrait s'amuser à gérer d'autres objets bonus.