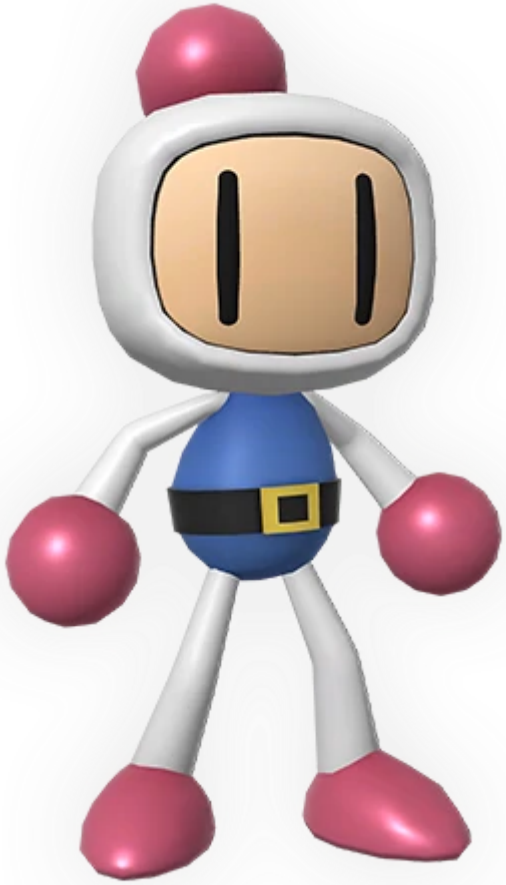


RAPPORT PROJET BOMBERMAN

CHAUDEURDY CORENTIN
LARDEUX Aloïs

université angers

Table des matières

I - Introduction

Premièrement, vous pouvez trouver les sources et l'historique de notre travail sur le dépot github suivant: **https://github.com/AloisL/bomberman**
Vous trouverez sur ce dépôt: les sources du projet, ce document ainsi qu'un diagramme de classe de l'architecture du projet sous forme d'un fichier class_diagram.png.

Deuxièmement afin de faciliter le développement du jeu, nous avons décidé d'utiliser la technologie Maven: cela nous permet notamment d'ajouter des dépendances sans avoir à devoir les ajouter aux sources du projet.
Même si dans les faits, nous avons uniquement utilisé cette fonctionnalité pour ajouter un système de logs, cela a pour conséquence la nécessité disposer de maven 4 sur la machine et d'importer le projet en tant que projet maven si l'on souhaite ouvrir notre projet dans un ide.

De plus, ayant toujours testé le jeu depuis intellij, nous n'avions pas pensé que le jeu devrait être lancé en temps qu'application indépendante et par conséquent il est possible qu'il soit nécessaire de lancer le projet avec intellij si il ne fonctionne pas avec Eclipse. La version community gratuite de intellij étant suffisante.

II - Lancement et commandes du jeu

Le joueur dispose de 3 vies.

Afin de lancer le jeu il faut:

1. Sélectionner un plateau de jeu à l'aide du menu déroulant
2. Choisir sa vitesse de jeu (il est conseillé de laisser à 1) à l'aide de la règle
3. Initialiser le jeu en appuyant sur le bouton actualiser bleu
4. Appuyer sur play, le jeu commence

Le bouton pause et réinitialiser permettent de relancer une partie.

Les touches utilisées pour le déplacement du joueurs sont les suivantes:

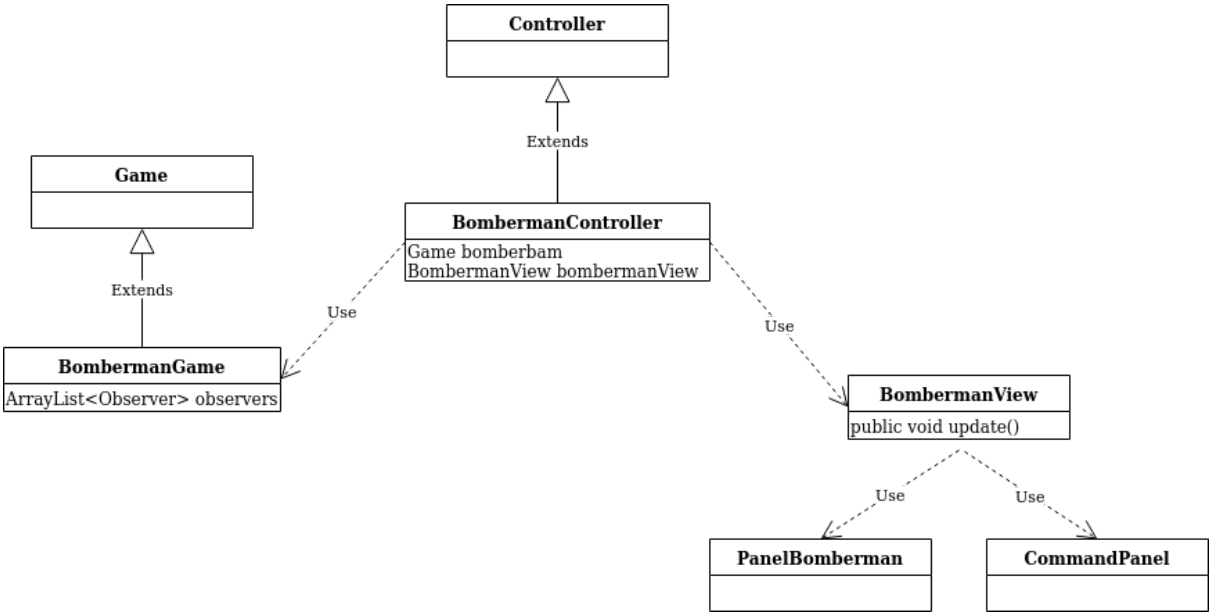
- Maintenir les flèches directionnelles pour effectuer le déplacement du joueur
- Appuyer sur espace pour poser une bombe

III - Patterns

II.1 - MVC et Observateur

Lors de la phase initiale du développement du jeu, il a été demandé d'implémenter le design pattern MVC pour la liaison entre la vue et le jeu. Nous en avons profité pour ajouter le design pattern Observateur nous permettant la mise à jour de l'ihm à chaque mise à jour du jeu, principalement lors de l'initialisation et lors des tours de jeu.

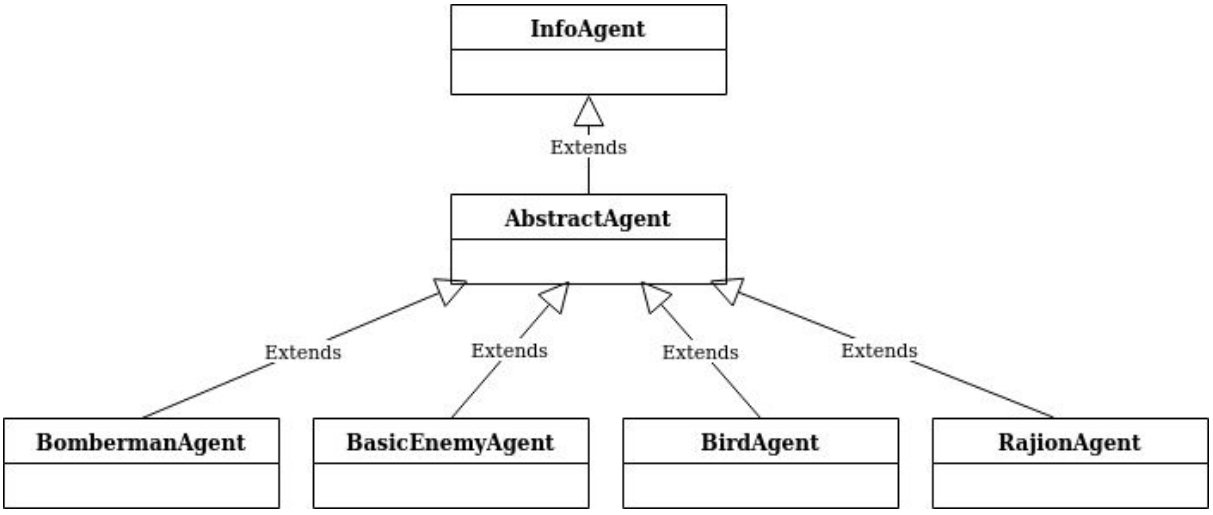
Nous avons par conséquent procédé en suivant le diagram de classes suivant:



Nous pouvons constater ici que le pattern MVC est respecté: la vue et le jeu ne communiquent pas directement ensemble. Lorsque la vue nécessite de communiquer avec le jeu des informations telles que les entrées clavier ou les action des boutons, elle communique les informations au controller qui va ensuite s'occuper de traduire les informations au jeu. De ce fait le controller possède en attributs un BombermanGame ainsi qu'une BombermanView.
Nous avons par contre jugé bon de légèrement transgresser au pattern MVC dans l'implémentation du pattern observateur. En effet, bien que le jeu (observable) possède une liste d'observateurs, nous avons décidé d'ajouter au sein de cette liste non pas le controller mais directement la vue.
Ce choix est motivé par le fait qu'ajouter le controller en tant qu'observateur aurait eu pour conséquence la simple encapsulation de la méthode update() de la vue dans la méthode update() du controller. Ainsi, lors de la mise à jour du jeu, le controller aurait vu sa méthode update() appelée qui elle même aurait ensuite uniquement appelé la méthode update() de la vue. Nous trouvions cela redondant et avons donc décidé de placer la vue en tant qu'observer. Ainsi lorsque le jeu est mis à jour, il "communique" avec la vue au travers de l'appel à la fonction notifyall() indiquant que le jeu est mis à jour.

II.2 - Patron de méthode (Agents)

Dans le but de faciliter la gestion des agents du jeu, nous avons décidé d'utiliser l'architecture suivante:



Cette architecture a plusieurs avantages: premièrement, l'utilisation du pattern patron de méthode pour la gestion des agents du jeu permet de traiter toutes les informations communes à tous les agents du jeu au sein d'une même classe abstraite AbstractAgent.
Et deuxièmement, le fait que la classe AbstractAgent hérite de la classe InfoAgent permet de manipuler une liste d'AbstractAgent comme une liste d'InfoAgent et ainsi créer une sorte de "rétrocompatbilité" avec les sources fournies au lancement du projet qui a facilité le développement du jeu et augmenté la clarté du code.

II.3 - Factory

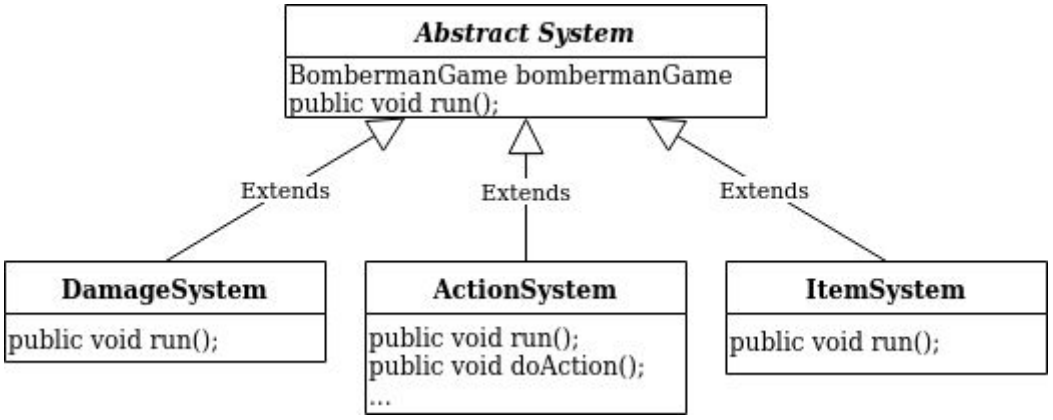
Puisque la structure des agents est articulée autour du pattern patron de méthode décrit dans le point précédent, nous avons donc implémenté le pattern Factory pour l'instantiation des agents lors de l'initialisation du jeu. Cela se traduit par la création d'une classe AgentFactory possédant une seule méthode statique:
public static AbstractAgent newAgent(char type, int posX, int posY, ...);
Cette méthode permet l'instantiation des agents spécifique du jeu en passant simplement le type de l'agent en paramètre de cette méthode.
Cette factory est donc appelée à l'initialisation du jeu afin de peupler une liste d'agent spécifiques, et ce, à l'aide d'une liste d'InfoAgent contenant tous les agents à créer pour un niveau ainsi que leur type sous forme d'un char.

II.4 - Patron de méthode (Sous-Systèmes)

Lors du développement du jeu, nous nous sommes rendus compte qu'un certain nombre d'éléments se retrouvaient systématiquement appelés à tous les tours du jeu. Nous avons appelé ces éléments des "sous-systèmes" et les avons isolés de cette façon:

- Un système gérant les déplacement des agents (ActionSystem)
- Un système gérant les dommages des appliqués aux agents (DamageSystem)
- Un système permettant la gestion des objets (ItemSystem)

L'implémentation de ces sous-systèmes est décrite par le schéma suivant représentant à nouveau un patron de méthode:



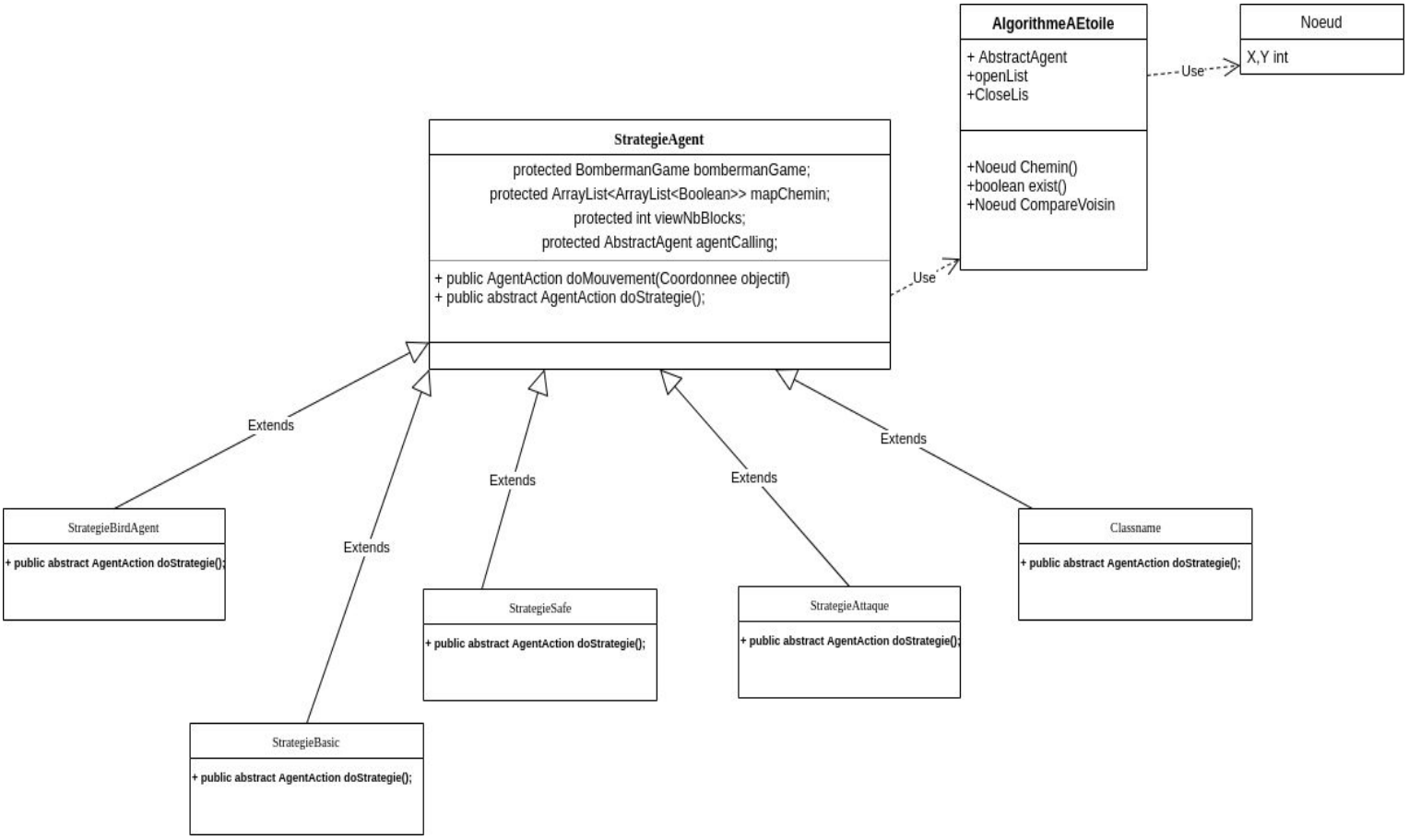
L'intérêt du pattern patron de méthode réside dans le fait que ces sous-systèmes doivent nécessairement redéfinir la méthode run(), méthode appelée systématiquement à tous les tours du jeu et contenant l'ensemble des actions à effectuer en lien avec le sous-système durant un tour de jeu.
Ainsi l'appel au méthode run() des trois sous systèmes dans un tour suffit à effectuer l'ensemble des déplacements, dégats et manipulations sur les objets (cf: BombermanGame.takeTurn()).
De plus cette conception à l'avantage d'offrir l'accès à toutes les données du jeu aux sous-systèmes au travers des attributs de AbstractSystem.
A noter que bien que la méthode run() doive nécessairement être redéfinie, d'autres méthodes peuvent être utilisées de manière indépendante si nécessaire comme dans le cas de l'intelligence artificielle qui utilise d'autre méthodes que run() dans le sous-sytème ActionSystem.

II.5 - Commande

Dans le cas de la classe ActionSystem gérant les déplacement des IA, nous avons décidé d'utiliser une implémentation du pattern commande pour effectuer les diverses actions des agents. Ainsi, lorsque qu'un agent doit se déplacer dans le jeu, il suffit d'appeler la méthode public void doAction(AbstractAgent agent, AgentAction action) de ActionSystem.
Cette méthode effectue le déplacement d'un agent en prenant en paramètre lle dit agent et son action, la logique est donc cachée derrière un système indépendant de l'agent.

IV - Intelligence Artificielle

Pour attribuer une intelligence artificielle aux personnages non joueur nous avons utilisé le design pattern Stratégie avec l'algorithme A*, voici son schéma UML:



Malheureusement ayant mal réalisés la récursivité d'une de nos méthodes, il arrive que l'algorithme se trouve dans une boucle infinie.
Nous avons donc mis en commentaires certaines parties pour que le projet soit exécutable.

La Stratégies safe n'est utilisable que pour les terrains ou les IA n'utilisent pas de bombes.
De même, la stratégie StrategieAttaque qui n'est pas utilisée comme nous l'aurions voulu puisqu'elle ne permet pas aux IA de casser les murs afin de se frayer un chemin..
L'algorithme A* sert principalement à rechercher un chemin d'un point A à un point B.

Pour le design Pattern Stratégie, nous avons utilisé une classe abstraite façon patron de méthode ce qui permet de factoriser une grande partie du code.
Les 3 stratégies :

- StrategieAttaque
- StrategieSafe
- StrategieBasic

La stratégie Stratégie Basique est utilisé par tous les types d'agents excepté le Bird qui utilise la stratégie StrategieBirdAgent.
Les 3 stratégies étaient supposées être utilisées de pair pour former une IA à plusieurs facette qui aurait pu être combinée avec le design pattern État.

Au moment de la création du pattern Stratégie nous nous sommes demandé si il n'était pas plus simple d'utiliser une classe "Coordonnee" représentant une coordonnée dans le plan du jeu. Mais l'ayant implémenté un peut tardivement dans le projet, nous ne l'avons pas utilisé dans toute les classes qui auraient qui pourrait en tirer partie.

L'algorithme A* utilise normalement 3 types de valeurs sous la formule F(x)=g(x)+h(x):
g(x) : le coût (ou nombre déplacement ou encore la distance depuis le noeud de base)
h(x) : l'heuristique (ou la distance depuis la case jusqu'à l'objectif).

Cependant, nous n'avons pas réellement compris l'utilité des 3 valeurs dans le cas ou l'on utilise pas les diagonales, nous avons donc uniquement basé la recherche de chemin le plus court sur l'heuristique.

Nous n'avons pas cherché à utiliser le "perceptron" fournit en cours pour une raison: il utilise le principe des réseaux de neurones. Ayant eu comme projet de concrétisation disciplinaire "Le deep Learning et L'euroMillion", nous avons choisi d'en apprendre un peu plus sur les algorithmes différents que sont les algorithmes de type "PathFinding" que l'on peut retrouver dans les jeux vidéos.

V - Conclusion

Voilà qui conclut notre rapport portant sur le projet bomberman. Nous avons essayé de prendre le plus de recul possible dans sa rédactions et certains de nos choix nous ont mené à de nombreux contre-temps.
Certains ajouts que nous pensions facilement implémenter ne sont pas dans la version finale comme par exemple le mode 2 joueurs qui est pourtant compatible avec la conception du jeu que nous proposons notamment grâce à la gestion des agents joueurs mais qui n'est pas implémenté faute de la mauvaise gestion de notre temps.
De plus, bien que git nous ai permis de gagner du temps à bien des égards, de nombreux soucis de merge duent aux nombreux changement dans l'architecture au cours du développement ont contribué à cette mauvaise gestion du temps.
Nous vous remercions de votre patience et sommes désolés du temps supplémentaire que la rédaction du rapport nous a pris.