Projet Design Pattern - Bomberman

Olivier Goudet

September 26, 2019

1 Description du sujet

Le jeu de plateau séquentiel que vous allez développer dans ce projet est composé d'un monde de taille limitée sur lequel évolue un certain nombre d'agents. Le déroulé général du jeu est le suivant :

- Au tour $t = 0$ le plateau du jeu est initialisé suivant une configuration définie par l'utilisateur. Les agents sont créés et placés sur le plateau.
- A chaque tour t, chaque agent peut réaliser une action prédéfinie par un ensemble de règles et qui a un impact sur l'environnement. L'ensemble de ces actions vont conduire à un nouvel état du monde au temps $t + 1$.
- Une fois que le nombre maximum de tours est atteint ou bien qu'une condition spécifique de fin de jeu est atteinte, le jeu s'arrête.

Nous allons créer un jeu de plateau de type Bomberman qui comprend un ou plusieurs agents *Bomberman* ainsi que plusieurs types de personnages non joueurs (PNJ).

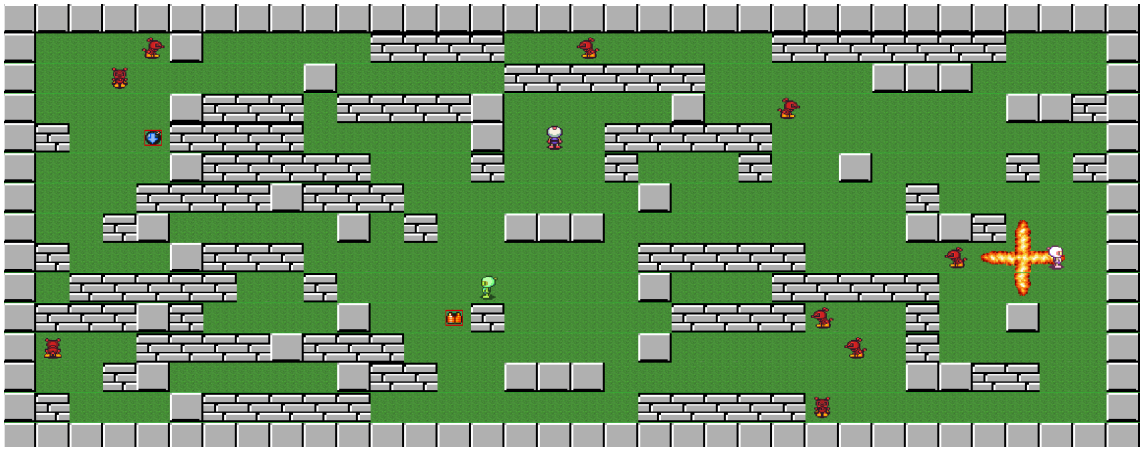

Figure 1: Plateau du jeu de Bomberman

1.1 Règles du jeu

Les règles du jeu Bomberman sont les suivantes :

- L'environnement est constitué d'un labyrinthe avec des cases libres, des cases de murs destructibles et des cases de murs indestructibles. Une vue du plateau est présentée sur la figure 1.
- Chaque agent Bomberman ou agent PNJ se déplace d'une case à chaque tour dans la direction de son choix mais ne peut pas aller sur une case mur.
- Si un agent Bomberman se trouve sur la même case qu'un agent PNJ, il est mangé et disparaît du plateau.
- A son tour de jeu, au lieu de se déplacer, un agent Bomberman peut décider de poser une bombe. Une bombe explose après un nombre de tours T_{bombe}. L'explosion se propage dans les 4 directions sur un nombre de cases C_{bombe}. Tous les ennemis (les autres Bomberman en mode non-coopératif et les PNJ) ainsi que tous les murs destructibles touchés par l'explosion sont retirés du plateau.
- A chaque fois qu'un mur destructible est détruit, il y une probabilité p_{item} qu'un objet bonus apparaisse à la place de ce mur destructible.
- Un agent Bomberman peut obtenir cet objet bonus en se déplaçant dessus. Il le conserve jusqu'à la fin de la partie.
- Le jeu s'arrête si tous les agents Bomberman sont éliminés (victoire des agents PNJ), ou bien si un agent Bomberman est le seul survivant (victoire de cet agent Bomberman).

1.2 Objectifs du projet

L'objectif de ce projet est d'implémenter un jeu de Bomberman interactif avec une interface visuelle de la façon la plus modulable et extensible possible en utilisant les Design Patterns vus en cours. Une fois la base du jeu réalisée, il s'agira d'implémenter les comportements et les stratégies des agents pour remporter la partie. On implémentera plusieurs modes de jeu en coopératif ou en affrontement avec un ou deux joueurs humain ainsi que des intelligences artificielles.

Les premières séances de TP seront très guidées de façon à ce que chacun puisse réaliser une base du jeu. Une fois cette base réalisée, plus de liberté sera accordée pour le développement de l'interface, des commandes et des stratégies des agents.

L'avancement lors des séances de TP sera pris en compte pour la notation finale.

Il y a 5 séances de TP en tout. Les séances de TP vont s'articuler dans les grandes lignes de la façon suivante :

Séance 1 Réalisation d'un patron de conception général d'un jeu de plateau. Implémentation d'un prototype de jeu très simple. Test de l'architecture. Début de l'implémentation d'une architecture Modèle-Vue-Contrôleur (MVC) pour gérer l'affichage graphique et les commandes de l'utilisateur.

Séance 2 Fin de l'implémentation d'une architecture Modèle-Vue-Contrôleur (MVC) pour gérer l'affichage graphique et les commandes de l'utilisateur. Initialisation de l'environnement de jeu Bomberman. Création des agents sur la plateforme. Affichage du jeu.

Séance 3 Implémentation des règles du jeu de Bomberman, des états des agents ainsi que des comportements.

Séance 4 Ajout de stratégies pour les agents. Ajout d'un mode interactif.

Séance 5 Implémentation de stratégies avancées : stratégie de recherche dans des arbres, algorithme A*, perceptron, coopération multi-agent...

2 Création de l'architecture du jeu

Il s'agit tout d'abord de créer une architecture générale d'un jeu séquentiel à l'aide du pattern *Patron de méthode*. Ce patron de méthode sera implémenté de façon concrète par un prototype de jeu très simple dans un premier temps avec une classe *SimpleGame*. On pourra étendre cette même architecture par la suite avec un jeu plus complexe du type Bomberman.

1. Créer une classe abstraite *Game* avec les éléments suivants :
   - un entier *turn* qui permettra de compter le nombre de tours du jeu.
   - un nombre de tours maximum *maxturn* dont la valeur est prédéfinie au moment de la création du jeu via le constructeur de *Game*.
   - un boolean *isRunning* qui permet de savoir si le jeu a été mis en pause ou pas.
   - une méthode concrète *void init()* qui initialise le jeu en remettant le compteur du nombre de tours *turn* à zéro, met *isRunning* à la valeur *true* et appelle la méthode abstraite *void initializeGame()*, non implémentée pour l'instant. Elle sera implémentée par les classes concrètes qui héritent de *Game*.
   - une méthode concrète *void step()* qui incrémente le compteur de tour du jeu et effectue un seul tour du jeu en appelant la méthode abstraite *void takeTurn()* si le jeu n'est pas terminé. Si le jeu est terminé, elle doit mettre le booléen *isRunning* à la valeur *false* puis faire appel à la méthode abstraite *void gameOver()* et qui permettra d'afficher un message de fin du jeu. Pour savoir si le jeu est terminé, on appelera la méthode abstraite *boolean gameContinue()* et on vérifiera si le nombre maximum de tours est atteint.
   - une méthode concrète *void run()* lance le jeu en pas à pas avec la méthode *step()* jusqu'à la fin tant que le jeu n'est pas mis en pause (testé par un flag booléen *isRunning*).
   - une méthode concrète *stop()* met en pause le jeu en mettant le flag booléen *isRunning* à *false*.
2. Créer une classe concrète *SimpleGame* qui hérite de cette classe abstraite *Game*. A chaque fois que quelque chose se passe dans le jeu, faites une sortie console. Cela servira à vérifier que la structure générale du jeu marche bien. Par exemple quand la méthode *takeTurn()* est appelée, on affichera "Tour x du jeu en cours", où x est la valeur courante du compteur de tour *turn*. La méthode *boolean gameContinue()* renvoie pour l'instant simplement la valeur *true* systématiquement.
3. Implémentez une classe *Test* avec une méthode *main* qui permet de créer un objet *SimpleGame*. On appelera ensuite la méthode *init()* de *SimpleGame*, puis on testera les méthodes *step()* et *run()* de *SimpleGame*. Vérifiez que tout fonctionne correctement avec les affichages en sortie console.
4. Comme vous pouvez le constater, le déroulé du jeu est très rapide jusqu'à la fin.

   Vous allez maintenant changer un peu la classe *Game* de façon à ce que la vitesse de déroulement du jeu soit contrôlable :
   - Faites en sorte que la classe *Game* implémente l'interface *Runnable*.
   - Ajoutez un attribut *thread* de type Thread dans la classe *Game*.
   - Ajouter une méthode concrète *launch()* dans la classe *Game* qui met l'attribut *isRunning* à la valeur *true* puis instancie l'attribut *thread* avec un nouvel objet Thread qui contient le jeu courant (*thread = new Thread(this);*) et enfin fait appel à la méthode *start()* de cet objet thread pour lancer le jeu. Remarque : l'appel à cette méthode lancera automatiquement la méthode *run()* de l'objet *Game* de type *Runnable* (cf. exemple du cours - séance 3 - task runner).
   - Ajouter un temps d'arrêt paramétrable dans la méthode *run()* après chaque tour de jeu : *Thread.sleep(time)*, où *time* est un attribut de type long de la classe *Game*. Il corresponds au temps de pause entre chaque tour en millisecondes.
   - Tester à nouveau avec *SimpleGame* que tout fonctionne correctement. Vérifiez que le changement de la valeur de l'attribut *time* a un impact sur la vitesse de déroulement de la simulation.

Remarque : on verra par la suite que le fait d'exécuter le jeu dans un thread présentera d'autres avantages que le simple contrôle du temps de la simulation. Cela permettra notamment à l'utilisateur de garder la main sur l'interface graphique pendant que le jeu s'éxécute en tâche de fond, ainsi que de lancer un grand nombre de simulations de jeu en parallèle pour évaluer différentes stratégies de comportement des agents.

3 Création du Modèle-Vue-Contrôleur

On souhaite maintenant créer une interface graphique pour ce jeu *SimpleGame*, avec un contrôleur qui fait le "pont" entre les éléments de l'interface graphique et le modèle et qui permettra à l'utilisateur d'interagir avec le jeu. Le design devra ensuite être facilement réutilisable pour le jeu plus complexe du Bomberman.

Pour l'interface graphique, il s'agira de réaliser deux classes (le détail de l'implémentation de ces éléments graphiques sera expliqué dans la section 3.2) :

- une classe *ViewSimpleGame* qui permet l'affichage du jeu *SimpleGame*. Pour l'instant l'affichage du jeu consistera simplement à écrire le tour courant du jeu sur un panneau comme sur la Figure 2.

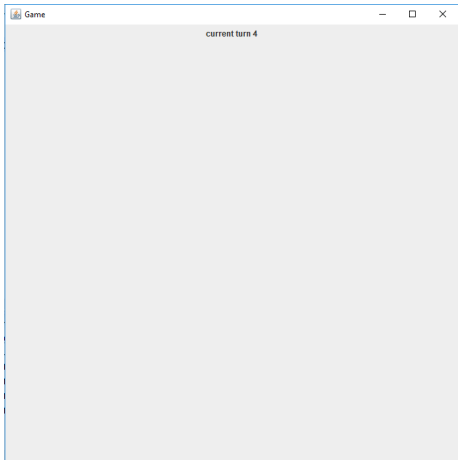

Figure 2: Affichage du jeu

- une classe *ViewCommand* qui affiche les principales commandes pour l'utilisateur (initialisation du jeu, lancement, mise en pause, etc...), ainsi qu'un compteur de tours du jeu, comme présenté sur la figure 3.

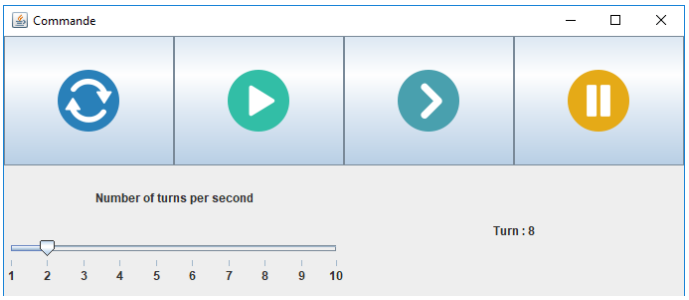

Figure 3: Interface de commande du jeu

Relisez le cours de la séance 2 pour vous rappeler le principe de l'architecture Modèle-Vue-Contrôleur.

Question préliminaire : proposer un rapide diagramme de conception du programme et discutez-en avec votre encadrant. Quel est le Modèle ici ? Qui est la Vue ? Qui sont les observables ? Qui sont les observateurs ?

3.1 Modèle

Faites en sorte que le modèle implémente une interface sujet (ou observable). Il y a deux possibilités d'implémentation :

1. créer vous même une interface d'observable que vous implémentez.
2. utiliser directement la librairie *java.util.Observable* de Java.

A chaque fois que le jeu change d'état, il faut notifier les observateurs. Pour cela, inspirez vous de l'exemple vu en cours. Il s'agit d'effectuer des notifications aux observateurs quand il se passe quelque chose dans le jeu : après l'initialisation, après qu'un tour soit effectué, etc.

Si vous utilisez la librairie *java.util.Observable* de Java, attention à bien appeler la méthode *setChanged()* sur l'observable pour dire qu'il a changé d'état juste avant de notifier les observateurs (voir l'exemple du cours).

3.2 Création de la Vue

Chacune des deux classes *ViewSimpleGame* et *ViewCommand* contiendra un élément de type JFrame qui permettra de placer les différents composants graphiques. Ces deux JFrame peuvent être disposées à deux endroits différents de l'écran. Pour dimensionner et positionner un élément JFrame par rapport au centre de l'écran, on peut utiliser par exemple le code suivant :

```
jFrame = new JFrame();
jFrame.setTitle("Game");
jFrame.setSize(new Dimension(700, 700));
Dimension windowSize = jFrame.getSize();
GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
Point centerPoint = ge.getCenterPoint();
int dx = centerPoint.x - windowSize.width / 2 ;
int dy = centerPoint.y - windowSize.height / 2 - 350;
jFrame.setLocation(dx, dy);
```

De plus, ces deux éléments doivent réagir à des changements qui peuvent se produire dans le jeu :

- l'interface de commande (classe *ViewCommand*) doit afficher un compteur du nombre de tours (voir figure 3 en bas à droite).
- l'interface de visualisation du jeu (classe *ViewSimpleGame*) doit afficher le nouvel état du jeu (pour l'instant il s'agit juste de l'affichage du nombre de tours courant sur un panneau, mais plus tard on dessinera le plateau du jeu Bomberman à chaque tour).

Comment faire en sorte que ces deux éléments réagissent à des modifications de l'état du jeu ?

3.2.1 Affichage des commandes du jeu

Directement dans le constructeur de la classe *ViewCommand*, ou bien par appel à une méthode *createView()* de la classe, créez les éléments graphiques pour les commandes utilisateurs. Un exemple d'affichage est proposée sur la figure 3.

Pour réaliser cette affichage de commande il faut utiliser les classe *JSlider*, *JLabel* et *JButton*. On peut les placer avec des Panels composés de *GridLayout* (positions en quadrillage) : un premier avec un *GridLayout* (2,1) qui permet de couper le panneau global en deux suivant le sens de la hauteur. Un deuxième qui sera situé en haut avec un *GridLayout* (1,4) pour positionner quatre boutons et un deuxième en bas avec un *GridLayout* (1,2) pour positionner le slider et la zone de texte.

Pour ajouter des icônes on peut utiliser la commande suivante :

```
Icon icon_restart = new ImageIcon("icon_restart.png");
initChoice = new JButton(icon_restart);
```

Les fichiers d'icône sont disponibles dans le dossier *Icones* de la section TP du projet sur Moodle.

Il s'agit ensuite d'ajouter des écouteurs d'actions sur les différents composants de l'interface. Voici le code par exemple pour réaliser une action quand l'utilisateur clique sur le JButton *choixInit*.

```
choixInit.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evenement) {
                controleurGame.start();
        }
});
```

Lorsqu'une action est effectuée par l'utilisateur, on appellera ici l'objet appelé "controleurGame" de type *InterfaceControleur* qui sait comment gérer les actions des utilisateurs et les communiquer au modèle (cf. section 3.3 plus loin). C'est la stratégie pour la vue.

3.2.2 Création du panneau de jeu

Pour la partie affichage du jeu, créez simplement pour l'instant un panneau qui affiche sous format texte ce qui se passe dans le jeu (affichage du message qu'un nouveau tour se déroule, affichage d'un message quand le jeu est fini, ...).

3.3 Création du contrôleur pour l'interface graphique

La classe qui va faire le lien entre la vue et le jeu *SimpleGame* est une classe *ControleurSimpleGame*, qui est la stratégie pour la vue. Cette classe implémente l'interface *InterfaceControleur* dont dépend la classe *ViewCommand* de façon a rester le plus générique possible. On pourra facilement implémenter par la suite un autre type de contrôleur pour le jeu de Bomberman.

Ce contrôleur doit permettre de contrôler le jeu quand des actions ont été effectuées par l'utilisateur dans la Vue (par exemple lorsque l'utilisateur clique sur un bouton). Voir l'exemple du cours.

1. Définissez tout d'abord une interface de contrôleur *InterfaceControleur* avec les méthodes *void start()*, *void step()*, *void run()*, *void stop()*, *void setTime(double time)*. Ce sont les fonctionnalités que doivent assurer un contrôleur.
2. Créez une classe *ControleurSimpleGame* qui implémente *InterfaceControleur* :
   - cette classe *ControleurSimpleGame* doit avoir un constructeur avec en paramètre un objet de type *Game*. Ce constructeur doit créer les deux interfaces graphiques *ViewCommand* et *ViewSimpleGame*.
   - Implémentez les méthodes de l'interface *InterfaceControleur*. A chaque fois qu'une méthode est appelée on effectuera l'action correspondante sur le jeu. On ajustera aussi les choix possibles des boutons en conséquence. Par exemple, au début sera activé uniquement le bouton *init*, car c'est la première action à effectuer pour lancer le jeu. Ensuite quand l'utilisateur aura cliqué sur ce bouton *init*, on le désactivera (car il ne sert à rien de cliquer deux fois de suite dessus). On fera la même chose pour la gestion des autres boutons.
   - Ces méthodes du contrôleur sont appelées à chaque fois qu'un utilisateur clique sur un bouton dans l'interface graphique *ViewCommand*.
   - La méthode *setTime(long time)* sera appelée par l'objet de type *ViewCommand* au moment d'une action de l'utilisateur sur le Slider de contrôle du nombre de tours par seconde. Il fera appel une méthode *setTime(long time)* de l'objet *Game* pour régler le temps de pause entre chaque tour du jeu.

3.4 Test de l'architecture MVC

Créer maintenant une classe *Test* avec une méthode *main()* qui instancie un jeu de type *SimpleGame* ainsi qu'un objet de type *ControleurSimpleGame*.

Testez que tout fonctionne correctement au niveau de l'affichage, des différents boutons et du réglage du temps de chaque tour de jeu.

4 Création du jeu Bomberman

On va pouvoir maintenant facilement réutiliser l'architecture du jeu développée jusqu'ici pour créer le jeu Bomberman. Certains éléments vous seront fournis pour ne pas perdre trop de temps, notamment pour la réalisation de l'interface graphique du jeu.

4.1 Matériel fourni

Dans la section Projet Bomberman de Moodle se trouvent différents éléments qui vous seront utiles pour créer le jeu :

- Il y a tout d'abord un répertoire nommé *layouts* qui contient une base de plateaux de jeu au format texte. Vous pourrez bien sûr en créer de nouveaux.
- Dans le répertoire "image" se trouve toutes les images qui seront utiles pour l'affichage graphique. Pour les images soient correctement trouvées par le programme, il faut placer ce dossier à la racine de votre projet eclipse, au même niveau que votre dossier "src".
- Dans le répertoire "Fichiers sources Java" se trouvent des classes déjà implémentées pour l'interface graphique. Normalement vous ne devrez pas modifier le code de ces classes (au moins dans un premier temps pour le début du projet). Tout doit fonctionner comme si c'était une API externe qui gère l'affichage. Les fichiers source Java sont les suivants :

  1. *AgentAction*, *ColorAgent*, *ItemType* et *StateBomb* correspondent à des types prédéfinis que vous devrez utiliser dans ce programme de façon à uniformiser un peu les codes entre les différents groupes. Par exemple, chaque stratégie d'un agent devra renvoyer un type *AgentAction* : MOVE_UP, MOVE_DOWN, MOVE_LEFT, STOP...
  2. Les classes *InfoAgent*, *InfoBomb* et *InfoItem* correspondent aux informations nécessaires à communiquer à l'objet *PanelBomberman* pour l'affichage :
     - *InfoAgent* prend en paramètre la position de l'agent (x et y), le type *AgentAction* de la dernière action effectuée (pour avoir un affichage qui correspond à la direction choisie par l'agent), le type d'agent sous la forme d'un caractère ('B' pour un agent bomberman, 'R' pour agent rajion, 'V' pour un agent bird et 'E' pour un ennemi basique), le type *ColorAgent* pour modifier éventuellement sa couleur, ainsi que les flags booleans *isInvincible* et *isSick* qui modifient l'affichage.
     - *InfoBomb* prend en paramètre la position d'une bombe (x et y), sa portée et son état (la bombe grossit de step1 à step3, jusqu'à l'explosion).
     - *InfoItem* prend en paramètre la position d'un item (x et y) ainsi que son type *ItemType*. On pourra implémenter 6 types d'objet bonus et malus de base :
       (a) FIRE_UP : augmente de 1 la range de la bombe du bomberman.
       (b) FIRE_DOWN : diminue de 1 la range de la bombe du bomberman.
       (c) BOMB_UP : augmente de 1 le nombre de bombes que peut poser en même temps le bomberman sur le plateau.
       (d) BOMB_DOWN : diminue de 1 le nombre de bombes que peut poser en même temps le bomberman sur le plateau (avec un minimum de 1 a priori).
       (e) FIRE_SUIT : rend invincible pendant x tours (flag isInvincible à activer pour l'affichage).
       (f) SKULL : rend malade pendant x tours (flag isSick à activer pour l'affichage). Lorsqu'il est malade l'agent ne peut pas poser de bombes.
  3. *Map.java* vous fournit une classe qui charge l'ensemble du plateau de jeu de départ dans un objet *Map*. Le constructeur de cette classe prend comme argument le nom du layout du plateau à charger. Il y a différentes méthodes qui permettent de récupérer notamment la liste des agents qui sont sur la carte au début (méthode *getStart_ agents()*), un tableau des murs cassables d'origine (méthode *getStart_ brokable_ walls()*) ainsi qu'un tableau des murs non cassables (méthode *get_ walls()*).
  4. Le fichier *PanelBomberman.java* permet de créer le panneau du jeu. Il étend la classe JPanel et prend en paramètre dans son constructeur un objet Map. Il dispose d'une

méthode *setInfoGame* qui permet de mettre à jour les informations sur les agents, les bombes, les objets et les murs afin de rafraîchir l'affichage. Une fois ces éléments communiqués, il faudra faire appel à sa méthode *repaint()* pour mettre à jour le JPanel.

4.2 Réalisations à effectuer pour obtenir une première simulation de jeu

A partir de maintenant les consignes sont volontairement moins précises, c'est à vous de concevoir et implémenter le jeu. Un fil directeur général est cependant proposé :

1. Mettez à jour l'architecture MCV déjà implémentée avec cette fois-ci les classes du jeu Bomberman : *BombermanGame*, *ControleurBombermanGame* et *ViewBombermanGame*.
2. Chargez et affichez le plateau du jeu avec la position de base à partir d'un layout donné.
3. Créez une hiérarchie de classe Agent qui permet de modéliser tous les types d'agent du jeu. On distinguera cette hierarchie des agents du jeu de la classe *InfoAgent* fournie qui sert juste à communiquer des informations pour l'affichage graphique.
4. Implémentez la méthode *initializeGame* pour initialiser le jeu et notamment créer les différents types d'agent à partir de leurs positions initiales sur le plateau. Quel design pattern peut être utile dans ce cas ?
5. Vous pouvez ajouter si vous le souhaitez un menu déroulant dans l'interface qui permettra de choisir le layout à charger, pour cela regardez la classe *JFileChooser* de Java.
6. Implémentez les règles du jeu avec simplement des déplacements aléatoires des agents (ne pas implémenter tout de suite les règles avec les items spéciaux, ni les règles d'explosions de bombes). Pour cela il peut être utile de créer deux méthodes :
   - une méthode *isLegalMove* prend en entrée un agent et une action et renvoie vrai ou faux si l'action est possible sur le plateau.
   - une méthode *moveAgent* prend en entrée un agent et une action et met à jour la nouvelle position de l'agent si c'est un déplacement.
7. Implémentez la méthode *taketurn* qui effectue une action pour chaque agent puis met à jour le plateau (par exemple si un agent a été éliminé).
8. Faites en sorte que la méthode *update* de la Vue mette à jour le panneau du jeu en fonction des nouvelles positions des agents et de l'état du plateau.
9. Implémentez des comportements d'agent différents. Par exemple, l'agent Bird reste immobile, il se réveille quand un agent bomberman arrive dans un certain rayon d'action, puis se déplace au sautant par dessus les murs). Quel Design Pattern peut être utile ?
10. Implémentez les règles restantes avec les objets spéciaux, les bombes et l'élimination des agents.
11. Implémentez les conditions de fin du jeu.
12. On veut maintenant créer des agents bomberman avec des stratégies qui dépendent du fait qu'un item d'invincibilité ait été ramassé ou non. Par exemple quand un agent Bomberman est invincible (ce qui dure un temps limité), il ignore les bombes et a plutôt tendance à jouer agressivement. Quel design pattern vu en cours peut être utile dans ce cas ?
13. Implémentez une stratégie qui est en mode interactif. C'est l'utilisateur qui choisit avec le clavier l'action du bomberman à effectuer dans le tour.
14. Implémentez un mode coopératif ou un mode duel à deux joueurs.

4.3 Stratégies et conceptions avancées

Il est demandé de proposer des extensions au jeu. Le choix est libre mais entrera dans la note finale. Choisissez en binôme le thème qui vous intéresse puis discutez-en avec votre encadrant.

Plusieurs orientations sont possibles :

- Améliorer l'interface et la gestion du jeu (compte des points, du nombre de vies des bombermans), gestion des niveaux, etc...
- Ajouter de l'IA dans les agents bomberman et/ou PNJ (algorithme A*, perceptron, ...).
- ajouter de la coopération multi-agent entre les agents d'une même équipe (par exemple pour ne pas prendre le même chemin et encercler les ennemis).