Projet Design Pattern - Snake multijoueur

Olivier Goudet

September 20, 2022

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 qui comprend un ou plusieurs agents *snake* ainsi que différents items.

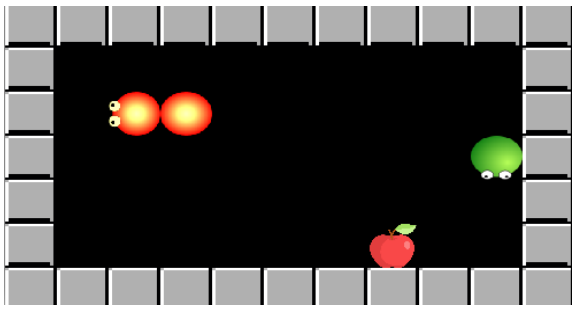

Figure 1: Plateau du jeu du Snake multijoueur

Règles du jeu

Les règles du jeu Snake multijoueur que l'on propose d'implémenter dans ce projet sont les suivantes :

- L'environnement est constitué d'un plateau constitué de cases libres et éventuellement de cases de murs indestructibles. Une vue du plateau est présentée sur la figure 1.
- un agent *snake* est toujours composé d'un tête. Il peut aussi avoir des éléments de corps qui le suivent.
- Chaque tête d'agent *snake* sans corps se déplace d'une case à chaque tour dans la direction de son choix.
- Chaque tête d'agent *snake* avec un corps se déplace d'une case à chaque tour avec une direction qui ne peut jamais être l'inverse de sa direction actuelle.
- Si un agent sort du plateau, il revient de l'autre côté car le plateau est sous la forme d'un tore (cette situation intervient s'il n'y a pas de murs sur les côtés ou bien si un agent à la possibilité de traverser les murs sans être éliminé).
- Si la tête d'un agent *snake* se retrouve sur la position d'un mur ou bien de son propre corps, il est directement éliminé.
- Si la tête d'un agent *snake* se retrouve sur la position d'un autre *snake* (tête ou corps) et que la taille de son corps est supérieure ou égale à celle de l'autre *snake*, il le mange et le fait disparaître.
- Il peut arriver que deux têtes de *snake* de même taille se rencontre sur la même case. Dans ce cas, les deux *snakes* sont simultanément éliminés.
- Si la tête d'un agent *snake* se retrouve sur une pomme, son corps grandit d'une unité.
- A chaque fois qu'une pomme est mangée par un agent, il y une probabilité p_{item} qu'un objet bonus apparaisse de façon aléatoire sur le plateau à une position libre.
- Un agent *snake* peut obtenir cet objet bonus en se déplaçant dessus. Il y a trois objet bonus différents :
  1. la balle jaune, rendant le *snake* malade pendant 20 tours de jeu. Lorsque le *snake* est malade, il ne peut plus manger de pommes ni d'items.
  2. la balle violette, rendant le *snake* invincible pendant 20 tours de jeu. Lorsque le *snake* est invincible, il ne peut pas être mangé par un autre *snake* et n'est pas éliminé s'il rencontre un mur.
  3. la boîte mystère, qui fait un effet aléatoire, correspondant à l'effet de la balle jaune ou de la balle violette.
- Le jeu s'arrête si tous les agents *snake* sont éliminés du plateau.

Objectifs du projet

L'objectif de ce projet est d'implémenter un jeu de Snake 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 pourra implémenter plusieurs modes de jeu en coopératif ou en affrontement avec un ou deux joueurs humain ainsi que des intelligences artificielles simples.

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. Création des premiers éléments de l'interface graphique.

Séance 2 Implémentation d'une architecture Modèle-Vue-Contrôleur (MVC) pour gérer l'affichage graphique et les commandes de l'utilisateur. Initialisation de l'environnement du jeu Snake et premier affichage du jeu.

Séance 3 Implémentation des règles du jeu de base du Snake ainsi que des premiers comportements.

Séance 4 Fin de l'implémentation des règles du jeu du Snake avec les règles spéciales relatives aux items. Ajout d'un mode interactif.

Séance 5 Implémentation de stratégies scriptées plus évoluées pour les agents, amélioration de l'interface graphique (par exemple pour compter les points ou bien changer de niveau), ajout de nouveaux items ou de nouvelles règles...

1 Séance 1

1.1 Création de l'architecture d'un jeu séquentiel

Il s'agit tout d'abord de créer une architecture générale d'un jeu séquentiel à l'aide du pattern *Patron de méthode* (voir cours séance 1). 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 *SnakeGame*.

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 tours du jeu et effectue un seul tour du jeu en appelant la méthode abstraite *void takeTurn()* si le jeu n'est pas terminé. 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. 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.
   - une méthode concrète *pause()* met en pause le jeu en mettant le flag booléen *isRunning* à *false*.
   - une méthode concrète *void run()* lance le jeu en pas à pas avec la méthode *step()* tant que le flag booléen *isRunning* reste à *true*.

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* (librairie java.lang.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*.
   - 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 correspond 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.

1.2 Création des premiers éléments de l'interface graphique

On souhaite maintenant créer les premiers éléments de l'interface graphique pour ce jeu type de jeu séquentiel.

Pour l'interface graphique, il s'agira de réaliser deux classes :

- une classe *ViewSimpleGame* qui permet l'affichage du jeu *SimpleGame*. Au début, 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.

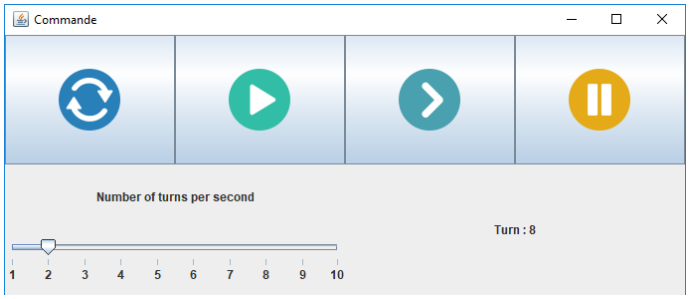

Figure 3: Interface de commande du jeu

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 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);
```

Il faudra préalablement importer les librairies *import javax.swing.*;* et *import java.awt.*;*.

Pour rendre la fenêtre visible, il faudra ajouter aussi la ligne *jFrame.setVisible(true);*.

Création de l'affichage du jeu

Pour la partie affichage du jeu, créez simplement pour l'instant un panneau qui contient un JLabel avec un petit texte. On affichera plus tard à cet endroit 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, ...).

Remarque :

- On peut centrer ce texte avec l'option JLabel.CENTER précisée dans le constructeur du JLabel.
- On peut changer la police avec la méthode *setFont()* du JLabel.

Affichage des commandes du jeu

Dans le constructeur de la classe *ViewCommand*, créez les éléments graphiques pour les commandes utilisateur. 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 (placés dans un dossier nommé icons et situé à la racine du projet), on peut utiliser la commande suivante :

```
Icon restartIcon = new ImageIcon("icons/icon_restart.png");
JButton restartButton = new JButton(restartIcon);
```

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

Pour le slider, on pourra le régler pour accepter des vitesses entre 1 et 10, et utiliser les méthodes *setMajorTickSpacing*, *setPaintTicks* et *setPaintLabels* pour afficher les choix de vitesse possibles.

2 Séance 2

2.1 Création du Modèle-Vue-Contrôleur

Voir la cours de la séance 2 sur le MVC. Dans notre cas, qui est la vue et qui est le modèle ? Comment faire en sorte que les deux fenêtres graphiques développées lors de la première séance réagissent à des modifications de l'état du jeu ?

2.1.1 Mise en place du design pattern Observateur

Mettez en place le design pattern Observateur de façon à ce qu'à chaque fois que le jeu change d'état, les différents éléments graphiques soient mis à jour. On affichera pour l'instant simplement le nombre de tours dans le jeu sur chacun de ces composants graphiques.

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.

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.
3. utiliser l'API *propertyChangeListener*.

Remarque : 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).

Faites un test pour vérifier si les différentes interfaces se mettent bien à jour au bon moment quand il y a un changement dans le jeu.

2.1.2 Création du contrôleur pour l'interface graphique

On va ajouter un objet "contrôleur" qui va faire le lien entre la vue et le jeu. Dans le modèle MVC, le "contrôleur" est la stratégie pour la vue.

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.

De façon a rester le plus générique possible et à factoriser du code commun aux contrôleurs des différents jeux (*SimpleGame*, puis plus tard *SnakeGame*), on peut réaliser une classe abstraite de contrôleur avec les fonctionnalités de base que doit assurer un contrôleur pour ce type de jeux.

Créer tout d'abord une classe abstraite de contrôleur *AbstractController* :

- Cette classe *AbstractController* doit avoir un attribut de type *Game*.
- Ajouter des méthodes dans *AbstractController*: *void restart()* (arrêt et réinitialisation), *void step()* (passage manuel d'une étape), *void play()* (passage automatique des étapes), *void pause()* (interruption du passage automatique des étapes), *void setSpeed(double speed)* (réglage de la vitesse du jeu). A chaque fois qu'une méthode est appelée on effectuera le ou les actions correspondantes sur le jeu *Game*.
- Attention, dans la méthode *setSpeed*, il y a une conversion à faire pour régler le temps de pause entre chaque tour dans le jeu.

Créez une ensuite classe concrète *ControllerSimpleGame* qui étend *AbstractController*. Cette classe doit avoir un constructeur qui permet de créer puis d'initialiser son attribut *Game* avec un jeu concret de type *SimpleGame*, mais aussi de créer les interfaces graphiques de type *ViewSimpleGame* et *ViewCommand*. Pour cette dernière, le contrôleur peut se passer en attribut car c'est une "stratégie" pour *ViewCommand*.

2.1.3 Ajout des écouteurs sur les différents composants de l'interface de commande

Ajouter ensuite des écouteurs d'actions sur les différents composants de l'interface graphique de *ViewCommand*.

Voici le code par exemple pour réaliser une action quand l'utilisateur clique sur le JButton *restartButton*.

```
restartButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evenement) {
                ...
        }
});
```

Lorsqu'une action est effectuée par l'utilisateur, on appellera ici son attribut de type *AbstractController* qui sait comment gérer les actions des utilisateurs et les communiquer au modèle. C'est la stratégie pour la vue.

2.2 Activation et désactivations des boutons en fonctions des actions effectuées par l'utilisateur

Une fonctionnalité intéressante peut être d'ajuster les choix possibles des boutons pour l'utilisateur en fonction du choix qu'il vient de faire.

Par exemple, au début seront activés tous les boutons sauf le bouton "restart" et le bouton "pause". Dès que l'utilisateur clique sur le bouton "play", on peut désactiver les boutons "play" et "step" (car le jeu est déjà lancé). Par contre à ce moment, on active les boutons "restart" et "pause", car ce sont de nouveaux choix proposés à l'utilisateur. On fera la même chose pour la gestion des autres boutons.

Quel design pattern peut être utile pour gérer ça ?

Faites un diagramme de conception, parlez en avec l'encadrant de la séance et mettez en oeuvre la solution retenue.

2.3 Test de l'architecture MVC

Créer maintenant une classe *Test* avec une méthode *main()* qui instancie 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.

Organisez votre projet en différents packages.

2.4 Intégration des premiers éléments du jeu Snake

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

2.5 Matériel fourni

Dans la section *Projet Snake* 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. Il faut placer ce dossier à la racine de votre projet eclipse, au même niveau que votre dossier "src".
- Dans le répertoire "images" 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*, *ColorSnake*, *ItemType* correspondent à des types prédéfinis que vous devrez utiliser dans ce programme de façon à uniformiser les codes entre les différents groupes. Par exemple, chaque stratégie d'un agent devra renvoyer un type *AgentAction* : MOVE_UP, MOVE_DOWN, MOVE_LEFT...
  2. Les classes *FeaturesSnake*, et *FeaturesItem* correspondent aux informations nécessaires à communiquer à l'objet *PanelSnakeGame* pour l'affichage :
     - *FeaturesSnake* prend en paramètre une liste de positions (x,y) de l'agent (la première position de cette liste correspond à la tête), le type *AgentAction* de la dernière action effectuée (pour avoir un affichage qui correspond à la direction choisie par l'agent), le type *ColorSnake* qui correspond à sa couleur, ainsi que les flags booleans *isInvincible* et *isSick* qui modifient l'affichage pour tenir compte du fait qu'il ait changé d'état (après avoir récupéré un item).
     - *FeaturesItem* prend en paramètre la position d'un item (x et y) ainsi que son type *ItemType*. Il y a pour l'instant 4 type d'items différents :
       (a) APPLE : pomme qui augmente de 1 la taille du *snake*.
       (b) SICK_BALL : boule jaune qui rend le *snake* malade pendant 20 tours de jeu.
       (c) INVINCIBILITY_BALL : boule violette qui rend le *snake* invincible pendant 20 tours de jeu.
       (d) BOX : boîte mystère qui a un effet aléatoire entre la maladie et l'invincibilité.
  3. *InputMap.java* vous fournit une classe qui charge l'ensemble du plateau de jeu de départ dans un objet *InputMap*. Le constructeur de cette classe prend comme argument le nom du layout du plateau à charger. *InputMap* dispose de différentes méthodes qui permettent de récupérer notamment la liste des agents qui sont sur la carte au début (méthode *getStart_ snakes()*), la liste des items présents sur la carte (méthode *getStart_ items)*), ainsi qu'un tableau des murs (méthode *get_ walls()*).
  4. Le fichier *PanelSnakeGame.java* permet de créer le panneau du jeu. Il étend la classe JPanel et prend en paramètre dans son constructeur différents champs qui permettent de spécifier l'affichage de départ du jeu. Il dispose d'une méthode *updateInfoGame* qui permet de mettre à jour les informations sur les agents, les items 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.

2.6 Intégration des premiers éléments du jeu Snake

- Importer les nouvelles classes dans votre projet :
  - Toutes les classes *AgentAction*, *ColorSnake*, *ItemType*, *FeaturesSnake*, *FeaturesItem* et *Position* peuvent être mises dans un package à part nommé "utils" pour ne pas encombrer vos packages existants.
  - La classe *PanelSnakeGame* est plutôt à mettre dans le package "view".
  - La classe *InputMap* dans le package "model".
- Mettez à jour l'architecture MCV déjà implémentée avec cette fois-ci les classes du jeu Snake : *SnakeGame*, *ControllerSnakeGame* et *ViewSnakeGame*.
- A partir d'un chemin de layout spécifié en dur pour l'instant, le constructeur de *ControllerSnakeGame* pourra se charger de la récupération de la carte dans un objet InputMap, de la création du jeu Snake avec cet objet objet InputMap en paramètre, de la création du panneau du jeu *PanelSnakeGame* puis de la création de l'interface graphique *ViewSnakeGame* avec ce panneau de jeu.
- On pourra dimensionner la taille de la fenêtre de jeu en fonction de la taille de la map.
- Faites un premier test d'affichage à partir des différents layouts.

3 Séance 3

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.

Réalisations à effectuer pour obtenir une première simulation de jeu

1. Créez la classe Snake qui permet d'instancier un agent *snake*. On distinguera cette classe Snake de la classe *FeaturesSnake* fournie qui sert juste à récupérer des information de base du layout et à communiquer des informations pour l'affichage graphique. Cette classe Snake contiendra d'autre méthodes pour gérer les comportements de l'agent, ses mouvements, etc.

2. 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 ?

3. Il est recommandé de créer une listes d'agents *snake* ainsi qu'une liste d'items dans le jeu.

4. 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'élimination des agents *snake*). Quel Design Pattern peut être utile pour implémenter les comportements des agents de façon à rendre le programme facilement extensibles lorsque l'on souhaitera implémenter plus tard de nouveaux comportements ?
   Pour gérer les déplacement, 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* qui met à jour la nouvelle position de l'agent si c'est un déplacement. Gérer les cas où l'agent sort du plateau. Dans ce cas, il faudra le faire revenir de l'autre côté.

5. Implémentez la méthode *taketurn* qui effectue une action pour chaque agent.

6. 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.

7. Testez le bon déroulé de cette première version du jeu.

4 Séance 4

4.1 Implémentation des règles de jeu plus avancées

1. Implémentez les règles éliminations des snakes présentés dans la description du jeu.
2. Implémentez les règles avec les objets spéciaux.
3. Implémentez les conditions de fin du jeu.
4. Gérer le fait qu'on puisse réinitialiser le jeu à sa position de départ lorsque le jeu est terminé.

5 Séance 5

Stratégies et conceptions avancées

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

Plusieurs orientations sont possibles :

- Ajouter un menu déroulant dans l'interface qui permettra de choisir le layout à charger, pour cela regardez la classe *JFileChooser* de Java.
- Implémentez une stratégie qui est en mode interactif (pour un ou deux joueurs). C'est l'utilisateur qui choisit avec le clavier l'action du snake à effectuer dans le tour.
- Améliorer l'interface et la gestion du jeu (compte des points, du nombre de vies des *snakes*), gestion des niveaux, etc...
- Ajouter de "un peu d'IA" dans les agents *snake* (stratégie scriptée plus évoluée, algorithme A*, ...).
- ...