VISITE ÉTUDIANT-E-S EMVS
The maze game

Hes·so VALAIS WALLIS
Haute Ecole d'Ingénierie
Hochschule für Ingenieurwissenschaften

1ère partie – Afficher et sortir du labyrinthe

Dans cette première partie, vous allez prendre connaissance du code source que nous avons préparée pour vous ainsi que de la structure du code Java. À la fin de cette partie, vous aurez de quoi afficher un labyrinthe sous forme de texte et d'image. Vous pourrez également afficher sur l'image le chemin de n'importe quel point vers la sortie.

Tâche 0 – Importation du projet

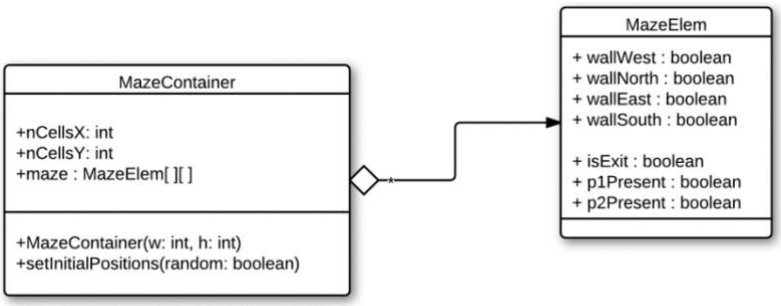
Nous avons préparé pour vous un projet Eclipse contenant les éléments de base du projet. Pour l'importer :

1. Lancez Eclipse puis faites *File->Import->General->Existing Projects into Workspace.*
2. Cliquez l'option *Select archive file*, choisissez le fichier *maze_codestudents_2014.tar* de la clé (*Browse*) et cliquez sur *Finish.* Le projet *EMVS* existe désormais dans votre liste de projets.

Tâche 1 – Dessin du labyrinthe sous forme de texte

Cette première tâche va vous permettre de comprendre comment est codé le labyrinthe dans le projet en vous demandant de l'afficher sous forme de texte dans la console. Cela vous sera utile tout au long de la journée car nous réutiliserons toujours la même structure de donnée.

1. Ouvrez le fichier `TextDisplay.java` qui se trouve dans `maze/display` et observez le fichier.
2. Remarquez qu'il y a un `main` dans le fichier. Comme en C/C++, le `main` constitue un point d'entrée dans le programme.
3. Dans le `main`, un `MazeContainer` est créé. Cette classe fabrique le labyrinthe et vous permet d'y accéder sous forme d'un tableau bi-dimensionnel de `MazeElem`. Dans la classe se trouvent également les informations sur les dimensions du labyrinthe (champs `nCellsX`, `nCellsY`).
4. Ouvrez la classe `MazeElem` et observez-la. Cette dernière classe contient déjà toutes les informations concernant ce qui se trouve dans une case du labyrinthe (murs, joueurs, sortie…).
5. Le diagramme de classes UML, c'est-à-dire comment les classes sont organisées, est le suivant :



6. Lancez le programme correspondant (clic droit sur la classe *TextDisplay* → *Run As* → *Java Application*).
7. Changez le programme de sorte à ce qu'un labyrinthe de taille 5x5 soit affiché.
8. Comme vous le voyez, le labyrinthe n'est pas affiché complètement, seulement les croisements et les murs du bas et de tout à droite sont affichés. Complétez le code aux endroits notés *TODO Task 1* afin d'afficher le labyrinthe correctement comme ci-dessous (`p1` correspond à la position du premier joueur et `e` à la sortie). Votre affichage devrait donner exactement :

```
*---*---*---*---*---*
| p1|               |
*   *   *---*---*   *
|   |   |       |   |
*   *---*   *   *   *
|           |       |
*---*---*---*---*   *
|       |           |
*   *---*   *---*---*
|         e         |
*---*---*---*---*---*
```

Tâche 2 – Dessin du labyrinthe sous forme d'image

Maintenant que nous sommes capables d'afficher correctement un labyrinthe sous forme de texte, il est temps de rajouter un peu de graphisme à notre programme. Une classe fonctionnant de manière semblable à `TextDisplay` vous est fournie dans le fichier `GraphicDisplay`.

1. Ouvrez le fichier `GraphicDisplay`.
2. Observez la méthode `main` de cette classe et exécutez-la.
3. Modifiez le code afin d'afficher un labyrinthe 10x10.
4. Il est possible de changer la taille de chaque petit carré en modifiant un paramètre du constructeur de `GraphicDisplay`. Essayez d'autres valeurs plus grandes et plus petites.
5. Il est possible d'afficher le labyrinthe sans les bordures de la fenêtre. Trouvez comment et exécutez votre programme. Vous obtiendrez quelque chose de similaire à ce qui se trouve à droite sur l'image ci-contre. Vous pouvez décider d'utiliser plutôt cette méthode ou l'autre selon votre préférence. Essayez également de faire apparaître le grillage d'arrière-plan (une variable à changer).

6. Afin de vous assurer que l'affichage est correct, affichez également depuis le `main` une version textuelle du même labyrinthe.

Tâche 3 – Algorithme de propagation de Lee (A*)

Dans cette phase, vous allez implémenter un moyen de trouver automatiquement la sortie du labyrinthe à l'aide d'un algorithme de routage.

1. Ouvrez la classe `AStar.java` qui contient le squelette de la classe du solveur pour le labyrinthe selon la méthode de Lee qui a été présentée au début de la matinée. La phase de propagation de l'algorithme (quand le chemin grandit dans toutes les directions depuis le point de départ) peut être décrite formellement comme suit :

Algorithme 1 Propagation de Lee (algorithme A^*)

Marquer le point de départ avec 1
$m \leftarrow 1$
// *Propagation en vague*
répéter
  Marquer tous les voisins non marqués des points marqués m avec $m+1$
  $m \leftarrow m+1$
jusqu'à destination trouvée **or** plus aucun point ne peut être marqué

1. Pour tester votre méthode, un `main` a également été créé ici. Vous pouvez l'utiliser pour débugger votre code plus facilement.
2. Observez la méthode `solve`. Elle commence par créer une solution vide (qui est en fait un tableau d'entiers). Elle continue en faisant la propagation de Lee puis le backtracking. Le but de tout l'algorithme est de mettre des 1 là où se trouve le chemin de la solution entre le point passé en argument et la sortie. Par exemple, le labyrinthe :

```
*---*---*---*---*
| p1|           |
*   *   *---*   *
|   |   |       |
*   *---*   *   *
|           |   |
*---*---*---*   *
|     e         |
*---*---*---*---*
```

donnera au final

```
1 - 0 - 0 - 0
1 - 0 - 1 - 1
1 - 1 - 1 - 1
0 - 1 - 1 - 1
```

3. Si on enlève le backtracking (en commentant la méthode `backtrace()` dans `solve`), on peut voir les étapes de la propagation ce qui est très pratique pour débugger son code. Dans le cas ci-dessus, cela donnera:

```
01 - 10 - 09 - 08
02 - 11 - 06 - 07
03 - 04 - 05 - 08
00 - 11 - 10 - 09
```

4. Pour pouvoir obtenir ce résultat, vous devez implémenter la méthode `expansion`. Celle-ci est appelée par `solve` tant qu'elle ne retourne pas *true* (c'est-à-dire tant que la propagation de la vague n'a pas atteint la destination). Notez également que la méthode reçoit à chaque appel l'état de *m* qui est le numéro de l'étape en cours. A vous de trouver comment faire (avec notre aide) ! N'hésitez pas à utiliser la méthode `access_solution()` (en comprenant à quoi elle sert…)

Tâche 4 – Afficher la solution sur la fenêtre graphique

Votre implémentation du point précédent vous donne un tableau contenant une solution à partir d'un point donné. Vous avez la possibilité de l'afficher directement sur la représentation graphique du labyrinthe. C'est ce que vous allez réaliser dans cette tâche.

1. En vous inspirant de ce qui a été fait auparavant, affichez depuis le `main` de la classe `AStar` la version graphique du labyrinthe.
2. L'objet de la classe `GraphicDisplay` que vous avez créé possède une méthode `setSolution` qui prend comme argument un tableau d'entiers correspondant à une solution depuis un point. Utilisez cette méthode pour afficher la solution calculée avec l'algorithme A* créé à l'étape précédente. Si vous désirez enlever la solution de l'affichage, utilisez la méthode `clearSolution();`

Tâches optionnelles

1. Changez la manière dont est dessiné le joueur (autre forme comme une flèche montrant le dernier déplacement, couleurs, …)

2. Affichez un dessin différent pour la sortie (par exemple une flèche pointant en direction de l'extérieur).

Partie 2 – Déplacement du joueur et logique du jeu

Vous avez maintenant un labyrinthe que vous pouvez afficher et dont vous pouvez trouver une solution. Le but de cette seconde partie sera de réaliser le déplacement d'un pion dans le labyrinthe ainsi que l'affichage d'un message lorsque la sortie sera atteinte.

Tâche 5 – Déplacement du joueur

La logique du jeu permettant d'afficher des messages lorsque la sortie a été trouvée se trouve dans la classe `MazeGame`. L'idée est relativement simple : le joueur peut déplacer son pion à l'aide des touches du clavier et lorsqu'un déplacement donne sur la sortie (c'est la même classe), le joueur a gagné et un nouveau jeu commence.

1. Prenez connaissance de la classe `MazeGame`.
2. Complétez la méthode `findPlayer(Player p)` qui permet de retourner les coordonnées d'un joueur sur le tableau de jeu. Les coordonnées retournées le sont sous la forme d'un `Point`. Si le joueur n'est pas dans le jeu, il faut retourner *null* qui est une valeur spéciale qui signifie "rien"[1].
3. Complétez la méthode `movePlayer(Direction d)` qui déplacera le pion dans la direction donnée. Notez que vous devez prendre en compte les murs correctement. Pour débugger cette méthode, utilisez-la directement dans le `main` afin de voir si tout se passe comme vous le pensez. Lorsque vous pensez que votre méthode fonctionne, essayez de déplacer votre pion dans différentes directions.
4. Essayez le code que vous avez écrit avec des labyrinthes qui ne sont pas carrés. Pourquoi précisément ce point peut-il potentiellement poser des problèmes ?

Tâche 6 – Gestion du clavier

Quelques mots de théorie – Renversement du flux de contrôle

Etant donné que Java est très fortement orienté-objet, il est possible de séparer clairement les différentes étapes. Ainsi, la gestion des touches se fait dans la classe `KeyboardListener`. Dans cette classe, le *flux de contrôle est renversé*. Cela signifie que ce n'est pas vous qui choisissez quand une méthode sera appelée mais la machine virtuelle (un peu

[1] Ce n'est pas le moyen le plus élégant de faire cela. En effet, que se passe-t-il si vous cherchez la position en X du joueur 2 et qu'il n'est pas là ? Essayer si vous avez le temps. Toutefois, au vu du temps disponible aujourd'hui c'est un moyen raisonnable.

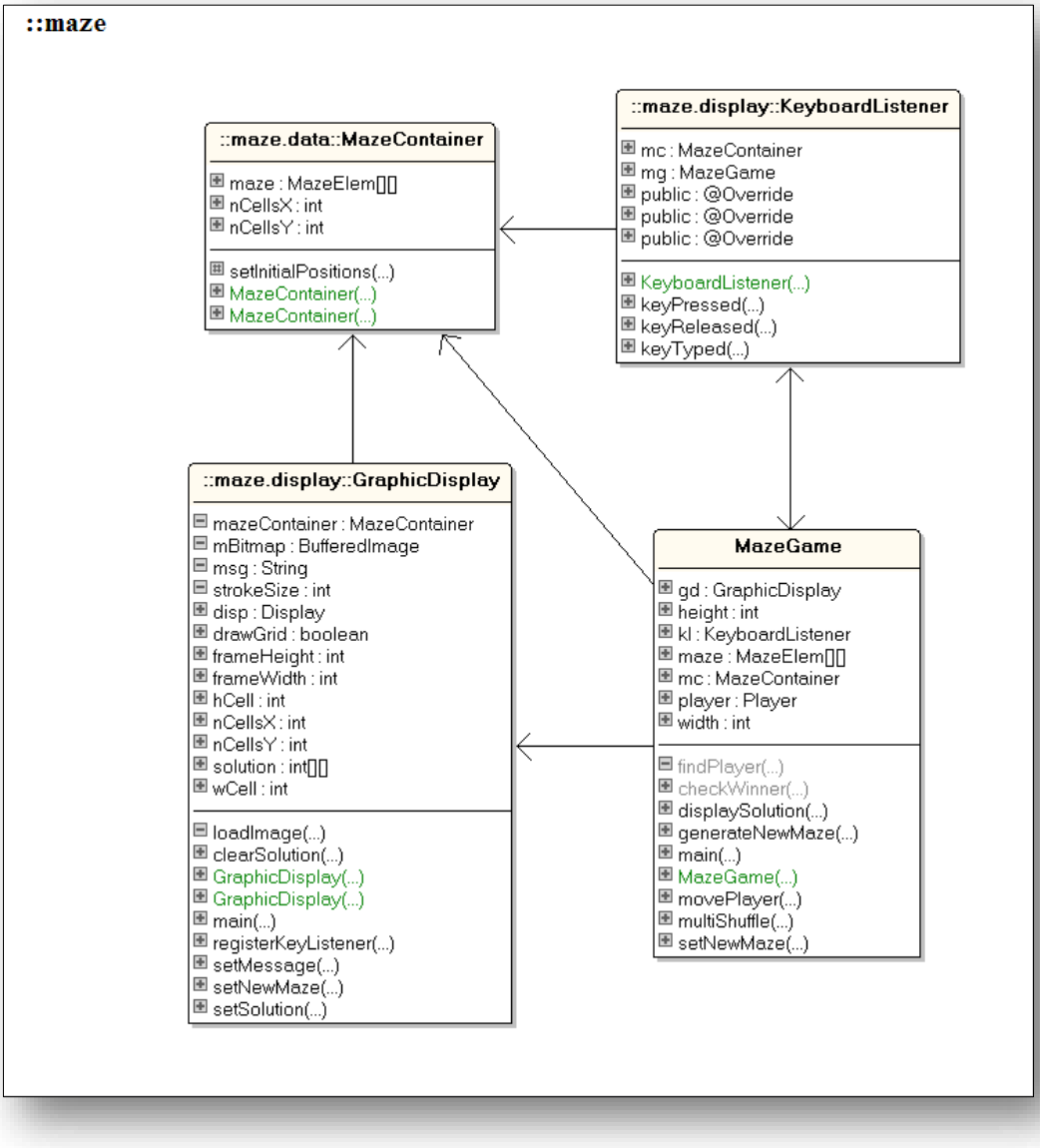
comme les interruptions). Par exemple pour une interface graphique, on ne sait jamais quand l'utilisateur va presser un bouton ou une touche et plutôt que de faire de *l'attente active*, cette manière de faire par inversion de contrôle permet de réagir quand il le faut.

Concrètement, cela signifie que des méthodes seront appelées par la machine virtuelle de Java lorsque l'utilisateur aura pressé une touche. Par exemple, lorsque l'utilisateur presse une touche et la relâche, la méthode `keyPressed` avec quelque chose permettant de récupérer la touche pressée en argument est appelée. Nous vous invitons à regarder le squelette fourni dans la classe `KeyboardListener` pour plus d'informations.

1. Observez la classe, notamment comment il est possible de réagir à des touches spéciales du clavier comme les touches de fonction (HOME, F1-F12…).
2. Remarquez qu'une instance du jeu et une instance du labyrinthe sont présentes dans la classe. A quoi servent-elles selon vous ?
3. Faites en sorte de pouvoir déplacer le joueur à l'aide des touches.
4. Faites en sorte qu'il soit possible d'afficher la solution en surimpression. Pour cela, vous devez lier une touche avec la méthode `displaySolution` qui vous est fournie.

Remarque :

Le diagramme de classes ci-dessous peut vous être utile si vous avez un doute dans les relations entre les classes :

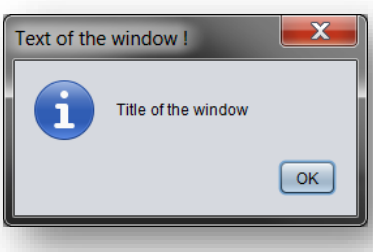


Tâche 7 – Sortie du labyrinthe

Maintenant que le clavier fonctionne correctement, il faut pouvoir afficher un message lorsque l'utilisateur a trouvé la sortie. C'est le but de la méthode `checkWinner` que vous devez implémenter. Pour afficher un message dans une nouvelle fenêtre, vous pouvez utiliser la technique suivante[2] :

```
JOptionPane.showMessageDialog(null,"Title of the window", "Text of the window
!", ,JOptionPane.INFORMATION_MESSAGE);
```

qui donne une fenêtre comme ci-dessous :



Nous vous souhaitons bien du plaisir dans la réalisation de ce projet !

[2] Elle se trouve en commentaire au fond de `MazeGame` comme cela vous pouvez la copier directement.