ALGORITHME D'ARIANE

13/05/19

Anne-Sophie Besnard
Romain Lechartier
Groupe 1

Table des matières

Algorithme d’Ariane

INTRODUCTION

L’algorithme d’Ariane est en réalité un labyrinthe. Le but ici est de pouvoir dirigé notre entrée (ici Thésée) à la sortie. Pour cela, le projet a besoin de deux algorithmes différents ainsi que de deux comportements différents.

Les algorithmes sont :

- Déterministe, Thésée se base alors sur ses connaissances de ses précédents déplacements pour savoir où aller. A noter qu’il n’a pas connaissance de la position de la sortie ni des murs.
- Aléatoire, c'est à dire que chaque direction (Nord, Sud, Est, Ouest) a autant de chance d’être choisie que les autres. Cela signifie également que l’algorithme n’est pas optimal et peut être très lent. De plus afin de déterminer une moyenne de déplacement que l’algorithme effectue pour que Thésée trouve la sortie, il doit l’effectuer 100 fois.

Ensuite on trouve deux comportements :

- Manuel, l’utilisateur peut alors voir en détail le comportement de l’algorithme en appuyant sur une touche afin d’avancer l’algorithme pas à pas.
- Automatique, la fenêtre contenant la grille ne s’affiche alors plus et l’algorithme effectue ses déplacements et affiche une fenêtre avec la moyenne de déplacement pour l’algorithme aléatoire qui s’est effectué 100 fois et seulement le nombre de déplacement pour l’algorithme déterministe.

FONCTIONNALITÉ DU PROGRAMME

Menu

Boutons et fenêtre

Dans le but d’avoir une fenêtre avec une taille uniforme pour tous nos menus nous avons fait une classe Fenetre qui hérite de JFrame.

Fenetre() se compose de 4 principales méthodes, les menus menu1() et menu2() ainsi que la saisie() et saisirand() pour les gestions des tailles des grilles,

respectivement, vide et aléatoire et qui crée alors un JTextField afin que l'utilisateur puisse écrire la taille et un bouton pour valider sa réponse.

Afin de faire nos boutons, nous avons fait une classe Bouton qui hérite de la classe JButton afin de pouvoir y ajouter des identifiants en int afin de faciliter leurs utilisations avec des listeners. Par exemple le Listener qui gère les menus du début est Observateur et afin de déterminer nos conditions au clique de la souris, on récupère l'identifiant du bouton précédemment défini en tant que paramètre.

Par exemple : Bouton jouer = new Bouton(5) ;

De plus, les différents menus sont créés en méthode dans la classe Fenetre(), afin de garder la même instance de fenêtre, nous avons « vidé » les panels avec la méthode clearContent() qui contient les méthodes getContentPane().removeAll(). Ce qui permet d'avoir l'ancienne fenêtre vide puis la remplir avec un autre menu en méthode après.

Ceci est la première fenêtre que l'utilisateur voit lorsqu'il lance le programme, elle lui permet soit de créer une grille lui-même (vide ou aléatoire), soit de charger une grille qui existe déjà (avec l'extension .lab).

Si l'utilisateur a choisi Nouvelle grille précédemment, il arrive sur cette fenêtre-ci.

Choix de la grille et taille

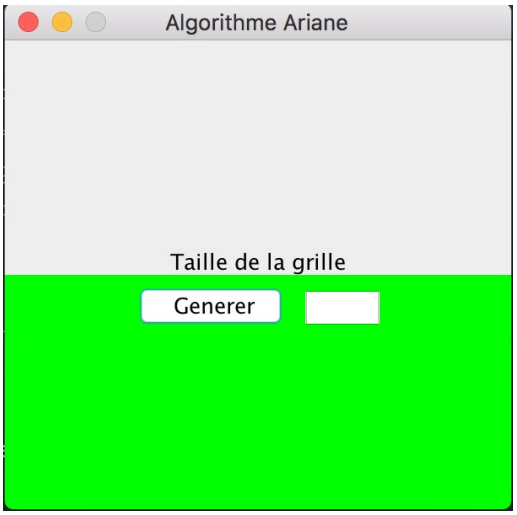
Récupération de la taille

Après avoir choisi la grille vide ou aléatoire, l'utilisateur est invité à donner une taille à la grille entre 5 et 60.

Cette dernière est générée par la classe ControleTaille(). Elle prend une Fenetre en paramètre ainsi qu'un JTextField.

On récupère l'identifiant du bouton sur lequel on a cliqué, le bouton 9 correspond à la grille vide, le bouton 10 à la grille aléatoire.

La taille est récupérée lorsqu'on clique sur générer, on convertit alors le texte contenu dans le JTextField en int avec la méthode Integer.ParseInt(), et on appelle la classe Grille pour la grille vide ou GrilleRand pour la grille aléatoire où l'on met en paramètre la taille obtenue.



L'affichage de saisi() et saisirand(), l'utilisateur met un nombre dans le JTextField et appuie sur générer, ce qui affiche la grille qu'il a précédemment sélectionné.

Afin de gérer les erreurs, notamment si l'utilisateur n'entre aucune taille dans le champ réservé, on utilise la structure try et catch. Le catch affiche alors un pop-up qui indique que la taille n'est pas valide, ainsi, soit l'utilisateur n'a entré aucun caractère, soit c'est un caractère non valide ou la taille ne correspond pas aux bornes.

Le pop-up est généré avec JOptionPane et correspond à un Warning message.

Grille aléatoire

Création de la grille aléatoire

Pour chaque élément tel que Mur ou Chemin, nous avons fait une classe Element dont Thésée, Mur, Chemin et Sortie héritent afin de récupérer la couleur, mais aussi de pouvoir utiliser instanceof pour savoir si la cellule sélectionnée correspond à un élément de la grille. Par exemple, lors de la simulation de déplacement de Thésée, si la prochaine cellule où se rend Thésée est une cell instanceof Mur alors Thésée ne pourra pas s'y rendre.

La grille se compose de cellules qui viennent de la classe Cell qui hérite de JPanel. Elles sont composées entre autres d'un index afin de numéroter les cellules de la grille. Ces cellules héritent de Jpanel, ainsi elles peuvent utiliser la couleur qui correspond à chaque élément pour les afficher dans la grille. Pour cela il y a la méthode getPropriete() qui récupère les propriétés de l'élément pour la placer dans la cellule Cell.

Tout d'abord, la grille est par défaut remplie de l'élément Chemin puis à l'aide de la fonction Random().nextBoolean() qui renvoie aléatoirement un booléen, chaque cellule qui compose la grille est aléatoirement soit un élément Mur soit un Chemin.

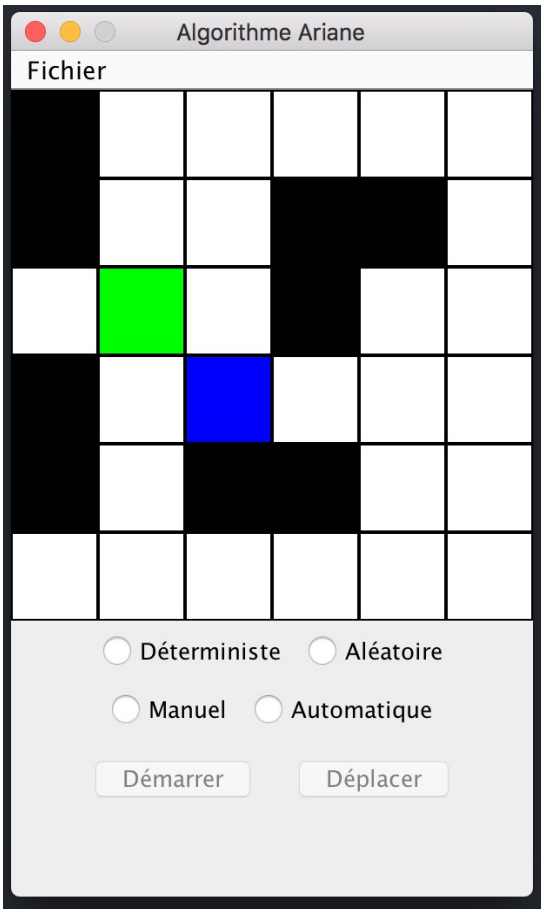
En raison de l'énoncé mal compris à priori, nous en avons déduit que le chemin de Thésée à la sortie était forcément accessible. Ainsi dans la classe GrilleRand, nous avons fait des méthodes afin de créer un chemin aléatoire en partant de Thésée, où chaque chemin visité sont marqués d'un booleen visited = true. A la fin de la simulation du chemin, la sortie est placée aléatoirement parmi les chemins qui portent le booleen visited = true, ce qui permet d'être sûr que la sortie reste accessible à Thésée par au moins un chemin.

La classe GrilleRand se compose aussi de méthode menuBarre() qui contient la JMenuBarre et de ses items, mais aussi de la méthode jouer () où se trouve les

JRadioButtons avec la sélection des algorithmes et du comportement mais aussi des boutons jouer et déplacer. Les Boutons jouer et déplacer sont créer à setEnable(false) afin de les rendre inutilisable si l'utilisateur n'a pas sélectionné le choix de l'algorithme ainsi que la méthode de simulation. Une fois les deux sélectionnés, les boutons se ne sont plus grisés.

Les JRadioButtons sont gérer avec le listener Choix, qui permet de savoir avec l'aide des méthodes getAlgo() et getMethode() quel algorithme et quelle simulation est choisie par l'utilisateur et permet alors d'y rattacher les bonnes méthodes correspondant à l'algorithme et au comportement qui se trouve dans la classe Algo(). Ces deux classes sont détaillés dans la section UTILISATION de GRILLEINTERFACE.

Le bouton jouer et déplacer sont gérées par le listener ValidationChoix qui lance l'algorithme choisis car les algorithmes sont composés de timers, cependant cela sera détaillé plus bas dans le titre sélection.



Ici un exemple de grille aléatoire, avec en vert l'Element Sortie, en bleu Thésée, en noir les Murs et en blanc Chemins.

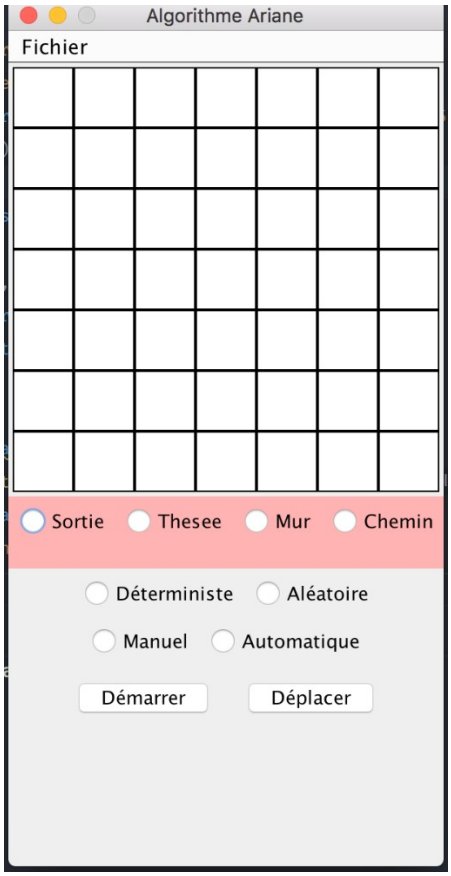
Option de rafraîchissement de la grille

Il est possible de rafraîchir la grille aléatoire générée en allant dans fichier et sélectionner « Rafraîchir ». Avec l'aide de la méthode getTaille() de la classe GrilleRand(), on récupère la taille précédemment sélectionnée par l'utilisateur et dans GrilleInterface = grille ; on instancie une nouvelle grille aléatoire, new GrilleRand(getTaille(), getTaille()) ;

Grille manuelle

Création de la grille manuelle

La grille se compose, comme la grille aléatoire, de cellule contenant l'élément Chemin par défaut. L'image ci-dessous illustre la grille manuelle vide, donc lors de sa création.



La fenêtre a les mêmes fonctionnalités que l'autre grille, c'est à dire, la sélection de l'algorithme, du comportement et de deux boutons : Démarrer et déplacer.

Ce qui diffère ce sont les JRadioButtons. Nous avons fait une classe DefaultBtn qui hérite de JRadioButtons, ce dernier a en argument dans son constructeur un Element (ainsi soit Thésée, soit Mur ...), une action issue de l'interface Action et un listener Selection et contient une méthode setEleminAction() ce qui permet de stocker l'élément du bouton cliqué, donc par exemple l'élément Thésée, pour ensuite affecter les bonnes propriété sur la cellule qui sera cliquée en utilisant la méthode saveElem qui est détaillée dans l'interface Action.

Ainsi il existe des Classes ChemBtn, TheseBtn, ... afin d'y appliquer l'Element spécifique pour chaque.

Le listener Selection récupère le DefaultBtn en cours et y applique la méthode setEleminAction().

INTERFACE ACTION

L'interface permet de faire l'intermédiaire entre le listener Selection et le listener Coloration, qui est chargé d'appliquer les propriétés de l'élément sélectionné dans la cellule cliquée par l'utilisateur.

Elle possède une méthode saveElem(Element e) qui permet de sauvegarder l'élément en cours pour l'utiliser dans le listener Coloration.

getElem() qui retourne l'élément en cours sélectionné par le bouton et ainsi par le listener Selection.

theseeLa(boolean bool) qui permet de mettre à true ou false si l'élément Thésée a déjà été placé dans la grille car la limite est de 1 par grille.

sortieLa(boolean bool), même utilité que la méthode au-dessus mais concerne ici la Sortie.

isThéséeLa() et isSortieLa() retourne la valeur true ou false afin de savoir si Thésée et la Sortie ont déjà été placée.

LISTENER COLORATION

Ce listener est un MouseActionListener et permet de « colorer » la cellule avec l'Element contenu dans le DefaultBtn sélectionné. Pour cela, le listener doit récupérer dans Element la propriété stockée dans le bouton avec l'aide de la méthode getElem(). Il faut aussi récupérer les propriétés de la cellule (issue de la classe Cell) sélectionnée avec getPropriete afin de savoir si Thésée était présent mais a été effacé par l'utilisateur qui a mis un Mur à sa place, par exemple.

On teste avec des instanceof si les éléments Thésée et Sortie sont placés en mettant à jour les valeurs de la méthode theseeLa et sortieLa.

Si on clique sur une cellule qui contenait déjà un élément Thésée ou sortie, on remet à false les deux méthodes ci-dessus car ces dernières ont été effacée de la grille.

Enfin, sur chaque cellule choisie par l'utilisateur on applique la méthode setPropriete(Element e) qui place dans la cellule les propriétés de l'Element en cours. Donc son Element (Thésée, Mur, ...) et sa couleur avec getCouleur() issue des classes héritant de Element.

Option pour effacer la grille

L'option pour effacer la grille marche exactement comme pour l'option de rafraîchissement de la grille aléatoire. Il possède le même listener et la méthode d'instance via la GrilleInterface marche de la même manière avec les getTaille().

Interface Grille

Utilisation

La création d'une interface GrilleInterface a été nécessaire étant donné que nous n'avions pas de classe Grille à part entière où la grille vide et aléatoire héritent.

L'interface permet alors de représenter les classes Grille et GrilleRand afin de faciliter l'utilisation de listener et des algorithmes.

Elle est composée de plusieurs méthodes avec divers objectifs :

- Int getTaille() : qui permet d'avoir la taille entrée par l'utilisateur
- Liste<Cell> getCells()
- String getAlgo : permet de récupérer en chaîne de caractère le type d'algorithme choisis. L'utilisation de String était nécessaire car nous utilisons un ActionListener dans la listener Choix, expliqué après.
- String getMethode : permet de récupérer la méthode de simulation de l'algorithme
- setAlgo(String s) : permet de set l'algorithme choisie
- setMethode(String s) : Permet de set la méthode choisie
- cacherFenetre() : cette méthode appelle .dispose() afin de fermer la fenêtre qui contient la grille ainsi que les boutons dans le cas de la méthode automatique
- getTheseeX() et getTheseeY() : permet de récupérer la position x et y de Thésée afin de l'utiliser dans la sauvegarde.
- getSortieX() et get SortieY() : permet de récupérer la position x et y de la Sortie afin de l'utiliser dans la sauvegarde.

- getEtat(int i)

CLASSE POSITION

Cette classe a été faites afin de pouvoir, en fonction des coordonnées de Thésée, savoir où il allait se trouver en fonction de la direction choisie.

Dans son constructeur il y a int x et int y ainsi que des getter et setter pour ces variables. On a également placé l'énumeration des directions, donc l'enum Direction. Elle prend en paramètre un x et y mais instancie également Position dans son constructeur afin de pouvoir utiliser les directions.

Cela signifie que si la direction Nord est choisie, Thésée devra calculer sa position actuelle en fonction de la direction pour avoir sa future position. C'est pourquoi il y a une méthode getNextPosition(Direction direction) qui ajoute les coordonnées actuelles de Thésée et les coordonnées de la direction afin de retourner la prochaine position dans laquelle Thésée va se trouver.

CLASSE ALGO

La classe Algo est composé de beaucoup de méthodes afin de rendre utilisable les algorithmes déterministe et aléatoire. Il est découpé ainsi pour permettre la réutilisation de certaines méthodes, ou du moins de leur squelette pour ensuite l'adapter en fonction du code.

Les méthodes principales sont de type exitMaze() et elles utilisent un timer afin de faire avancer la simulation dans le temps. En utilisant une boucle for() nous sommes arrivées à des erreurs de stackoverflow. Le timer a permis de résoudre le souci mais je pense qu'il nuit à la rapidité du programme et aux performances de la machine.

Après avoir utilisé la méthode getRandomDirection() (expliquée dans la section algorithme aléatoire) pour déterminer le chemin à prendre, l'algorithme doit vérifier si la direction choisie est valide ou non. Pour cela il y a la méthode deplacementValide, qui prend en argument la position de Thésée, contenu dans une Cell et une direction (issue de Position.Direction). Pour cela il suffit de vérifier à true ou false si, selon la direction choisie et la position de Thésée, la case suivante n'est pas de l'instance Mur.

LISTENER CHOIX

C'est un ActionListener, qui avec la méthode actionPerformed(actionevent e) va récupérer une String, qui correspond aux noms des boutons pour ensuite la comparer aux String passées en constante, comme ALEA, DETER pour les algorithmes, AUTO et MANUEL pour le comportement.

Le listener met ensuite dans setAlgo et setMethode l'algorithme et la méthode sélectionnées pour pouvoir l'utiliser dans la classe Algo avec getAlgo() et getMethode().

On ajoute le listener sur les JRadioButtons créer dans les classes Grille, GrilleRand, GrilleInterface.

LISTENER **VALIDATIONCHOIX**

Ce listener prend une Grilleinterface en paramètre, et son constructeur instancie Algo(grille), ce qui permet de lancer un nouvel algorithme.

Ce listener est un MouseListener, ainsi le listener est ajouté sur les boutons Démarrer et Déplacer, montré précédemment sur les grilles vides et aléatoires. On récupère leur id avec getId() afin de le connaître, 15 étant le bouton Démarrer et 16 Déplacer.

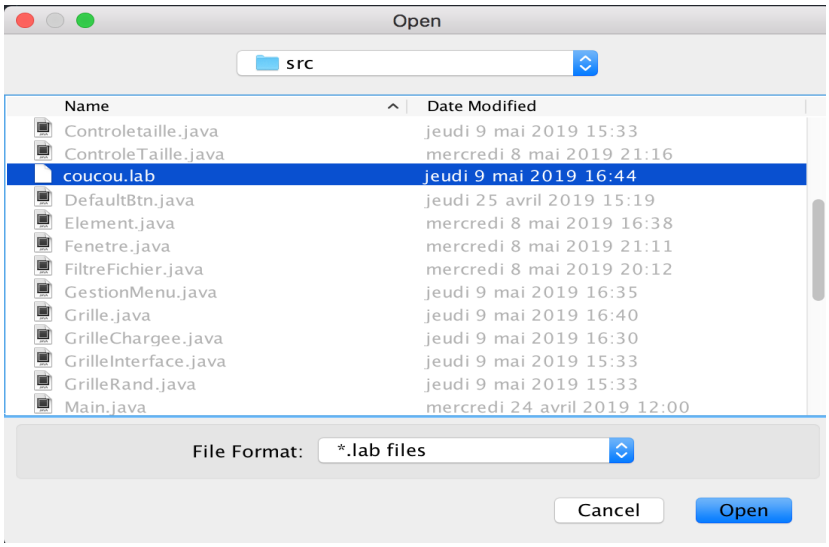
Afin d'empêcher l'utilisateur d'appuyer sur le bouton Démarrer continuellement lors de la simulation du programme en mode manuel, la méthode isEnable() et getEnable() est utilisée afin de désactiver le bouton Démarrer une fois que l'utilisateur a appuyé une fois dessus et ainsi démarrer la simulation.

A noter, cependant, que l'algorithme peut également être lancé directement par Déplacer, du fait de la similitude des méthodes qu'elles appellent. Mais il ne pourra être utilisé que pour la simulation manuelle et ne permet de de démarrer la simulation en mode automatique.

Le bouton déplacer appelle spécifiquement la méthode exitMazeManuel.start(), c'est pourquoi il n'est utilisable que pour la simulation manuelle.

Le bouton démarrer, utilise seulement la méthode start().

Chargement



Le chargement d'une grille s'effectue par le biais de la classe GrilleChargee.

Choix du fichier à lire

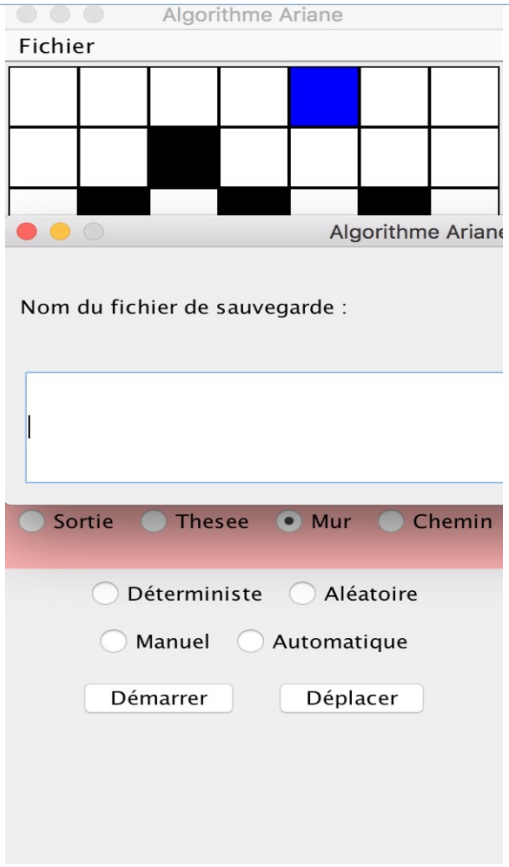
Dans cette classe, on appelle un objet de la classe ChoixFichier (qui hérite de la classe JfileChooser). Cet objet ouvre une fenêtre dans le répertoire courant afin de choisir le fichier que l'on veut récupérer. Un objet de la classe FiltreFichier (qui hérite de la classe FileFilter) est utilisé pour n'afficher que les fichiers contenant l'extension voulu .lab. En effet, on utilise la méthode lastIndexOf() pour déterminer la dernière apparition d'un « . » et on récupère, à l'aide la méthode substring() l'extension.

Lecture du fichier de sauvegarde

On obtient ainsi le chemin absolu vers le fichier qu'on transforme en String pour pouvoir appeler un FileInputStream avec ce fichier qui sert à établir le flux de lecture. On construit ensuite un DataInputStream à partir de ce flux afin de récolter les données souhaitées. On lira ensuite les données byte par byte à l'aide de la méthode readByte() dans l'ordre = taille de la grille, coordonnée X de Thésée, coordonnée Y de Thésée, coordonnée X de la sortie et enfin coordonnée Y de la sortie. Ensuite, les octets suivant sont lus byte par byte pour correspondre aux états de la grille dont la taille a été déterminée comme indiqué ci-supra. Tant que le fichier de lecture n'a pas été entièrement lu (ce qui est déterminé par la méthode available()), on lit des bytes. Les bytes ainsi lus sont transformés en int en utilisant un opérateur logique « and » avec le nombre décimal 0xFF afin que le 1er bit représentant le nombre ne soit pas un 1 (auquel cas, il y aurait un problème lors de la conversion en String ci-après). Ce nombre est ensuite converti en String temporaire puisqu'elle peut nécessiter l'ajout de 0 si elle n'a pas strictement la longueur d'un byte (soit 8). La nouvelle String ainsi obtenue est converti en un tableau de caractères pour représenter les états de la grille colonne par colonnes. Enfin, ce tableau est réarrangé par une opération simple (l'état de la 2ème case de la 1ère ligne est maintenant l'état de la 1ère case de la 2ème ligne et inversement, par exemple) pour être utilisé lors de la construction de la grille.

La grille correspondant à ces données sera ensuite directement créée à l'aide d'un GridLayout et la classe Cell à laquelle on donnera directement ses propriétés.

Sauvegarde



Choix du fichier de sauvegarde

La sauvegarde est déclenchée à l'aide d'un ActionListener qui cible l'option « Sauvegarder » de la barre de menu. Cet ActionListener est un objet de la classe GestionMenu qui créé une fenêtre dite pop-up, dans laquelle est présent un JtextField dans lequel l'utilisateur doit entrer le nom du fichier dans lequel il souhaite enregistrer les données correspondant à la grille. Un ActionListener se déclenche à partir de ce champ de texte, c'est l'ObservateurSauvegarde.

Écriture dans le fichier de sauvegarde

Tout d'abord, au nom du fichier désiré est ajoutée l'extension .lab. Ensuite, un flux d'écriture est créé à l'aide de FileOutputStream. Ce flux est utilisé par un DataOutputStream pour stocker les données à écrire. Les données sont écrites byte par byte à l'aide de la méthode writeByte() dans l'ordre : taille de la grille, coordonnée X de Thésée, coordonnée Y de Thésée, coordonnée X de la sortie et enfin coordonnée Y de la sortie. Ensuite, on récupère les états de la grille par la méthode propre à la grille getTaille() ligne par ligne et 8 par 8 (taille d'un byte) sous la forme d'une String. Ces échantillons de 8 états sont ensuite transformés en byte binaire à l'aide de la méthode parseInt(x,2) et sont écrits dans le fichier. Lors de la dernière écriture d'états, il se peut que le nombre d'états restants à écrire ne soit pas exactement égale à la taille d'un byte. On rajoute donc des 0 à la String représentant les états jusqu'à obtenir une String de 8 caractères avant de la convertir.

Choix de l'algorithme et du comportement

Algorithme déterministe

L'algorithme déterministe est un parcours en profondeur. Il fait appel à une pile et à une file. La pile sert à stocker les cases/cellules traitées (et ainsi les chemins pris) alors que la file sert à stocker les cases déjà visitées (et ainsi stocker les cases déjà visitées). La pile va donc permettre de revenir en arrière lorsque Thésée est bloqué et prendre un chemin différent alors que la file va permettre de ne pas tourner en rond en revenant sur ses pas par hasard. Ce système permet de parcourir le labyrinthe de manière scrupuleuse afin que la sortie soit trouvée en un minimum de temps.

Algorithme aléatoire

La méthode getRandomDirection permet de, façon aléatoire de prendre une direction aléatoire parmi les Directions contenu dans l'énumération. Le random, maths.random() * (4))+1 permet d'obtenir un chiffre aléatoire de 1 à 4. Ainsi 1 correspond au Nord, 2 au Sud,…

Ensuite on utilise la méthode déplacementAlea, permet que si le déplacement de Thésée est valide, ce dernier se déplace à la nouvelle position, donc on récupère les propriétés de Thésée avec getPropriete() pour ensuite setPropriété dans la cellule suivante. De plus, afin de ne pas voir le chemin précédent de Thésée, on instancie un new Chemin dans la case où se trouvait Thésée avant le déplacement.

Comportement automatique

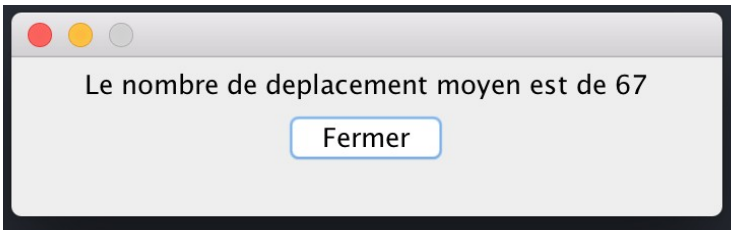
Dans la suite, on donnera le cheminement concernant le comportement automatique de l'algorithme aléatoire mais le même s'applique à l'algorithme déterministe.

C'est la méthode exitMazeAuto() qui gère le comportement de la simulation en mode automatique. Ce dernier a besoin d'un timer qu'on a mis à 0.01seconde afin que la simulation en arrière-plan aille plus vite, car les performances sont réduites avec son utilisation. Au début de la méthode on récupère les positions de Thésée et de la Sortie avec l'aide de la méthode searchThesee et searchSortie qui boucle pour vérifier l'état de toutes les cellules de la grille afin de les sauvegarder dans une variable PositionSortie et PositionThesee.

A chaque étape du timer (quand 0.01 seconde sont écoulées), on utilise la méthode searchThésée pour rafraîchir sa position à chaque étape et à avancer dans la

simulation. Lorsque la position de Thésée est égale à celle de la sortie, on incrémente aléa jusqu'à un maximum de 100. Tant qu'elle n'est pas égale à 100, on replace Thésée et la sortie à leur position de départ dans la grille, donc en utilisant les variables PositionSortie et PositionThesee, et on recommence la simulation tout augmentant la variable nbDeplacement avec deplacementPourMoyenne afin de calculer le nombre de déplacement moyen de Thésée à la sortie dans la simulation.

Puis un pop-up s'affiche pour donner le nombre de déplacement moyen créer par la méthode popupFin() qui est un JFrame contenant un JLabel et un bouton qui permet de quitter le programme avec System.exit(0).

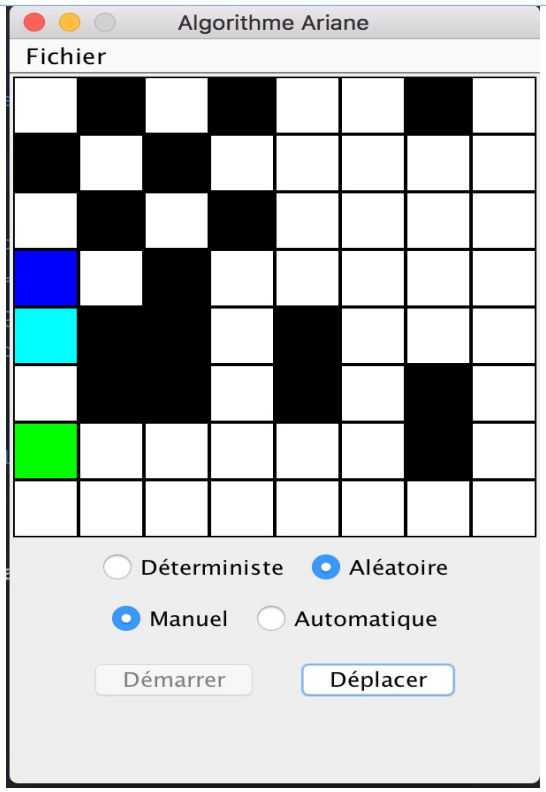


Comportement manuel

Dans la suite, on donnera le cheminement concernant le comportement manuel de l'algorithme aléatoire mais le même s'applique à l'algorithme déterministe.

Cette fois, c'est la méthode exitMazeManuel() qu'on utilise. Le timer est toujours utilisé mais est mis en pause. Afin d'avancer dans la simulation il faut alors appuyer sur les boutons démarrer puis déplacer. Le bouton déplacer appelle la méthode exitMazeManuel() et start() afin de le faire avancer.

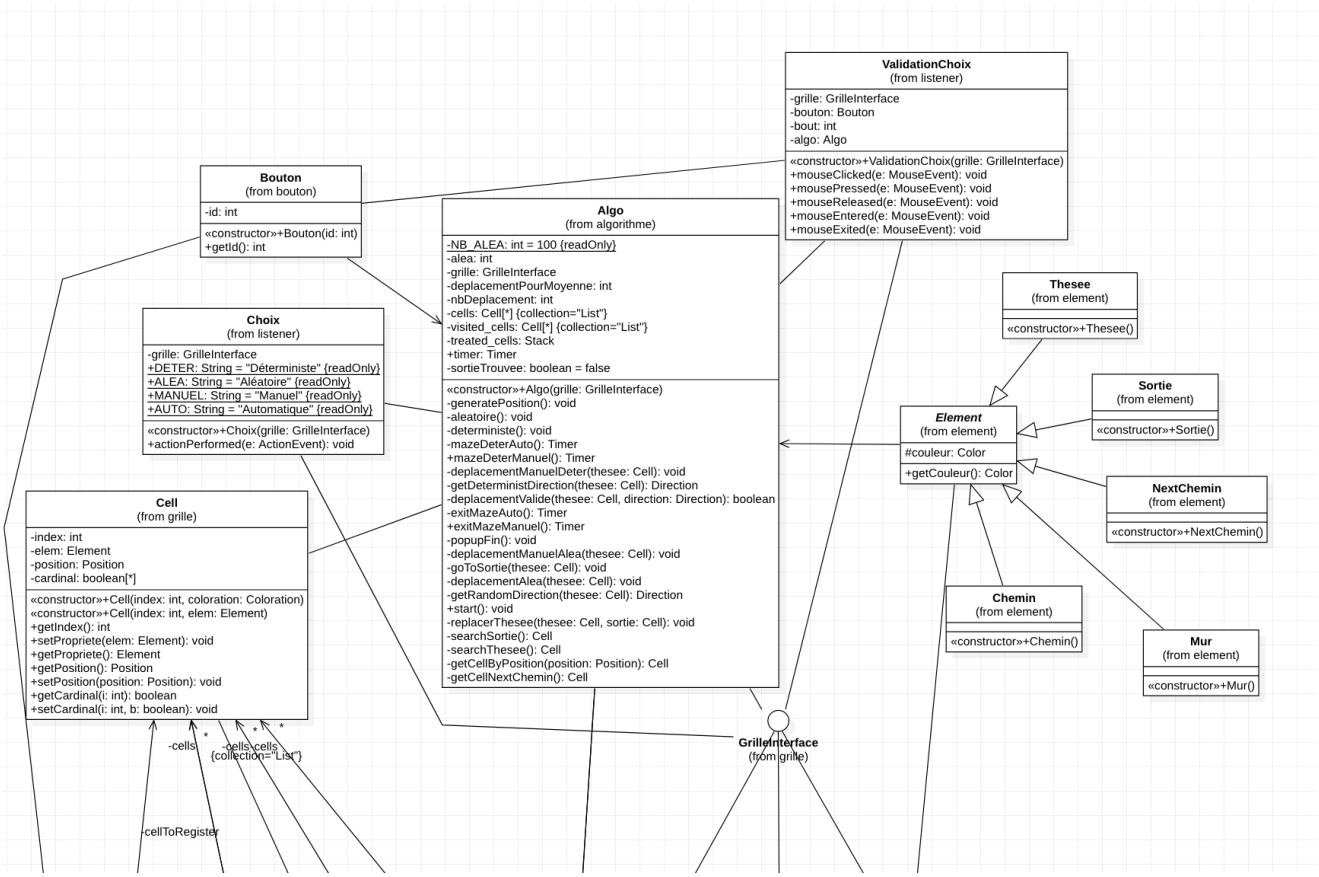
Pour permettre à l'utilisateur de voir où Thésée va se rendre lors de la prochaine étape de la simulation, la méthode déplacementManuelAlea est utilisée, à l'aide d'un element NextChemin et de sa couleur cyan. On utilise la méthode getCellNextChemin afin de récupérer spécifiquement la cellule contenant NextChemin, à chaque étape on vérifie si la prochaine position n'est pas la sortie, on retourne true si c'est la sortie.

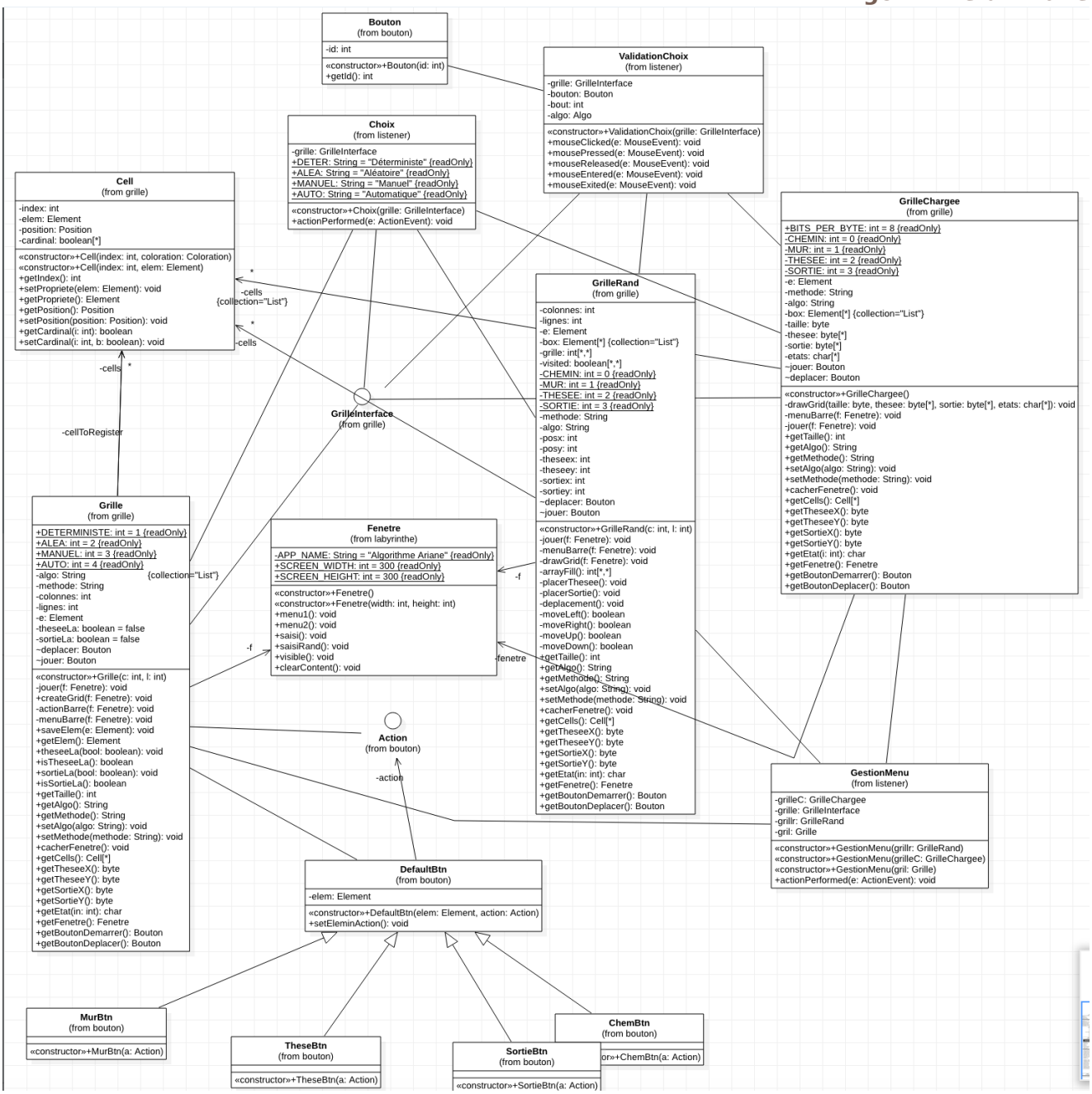


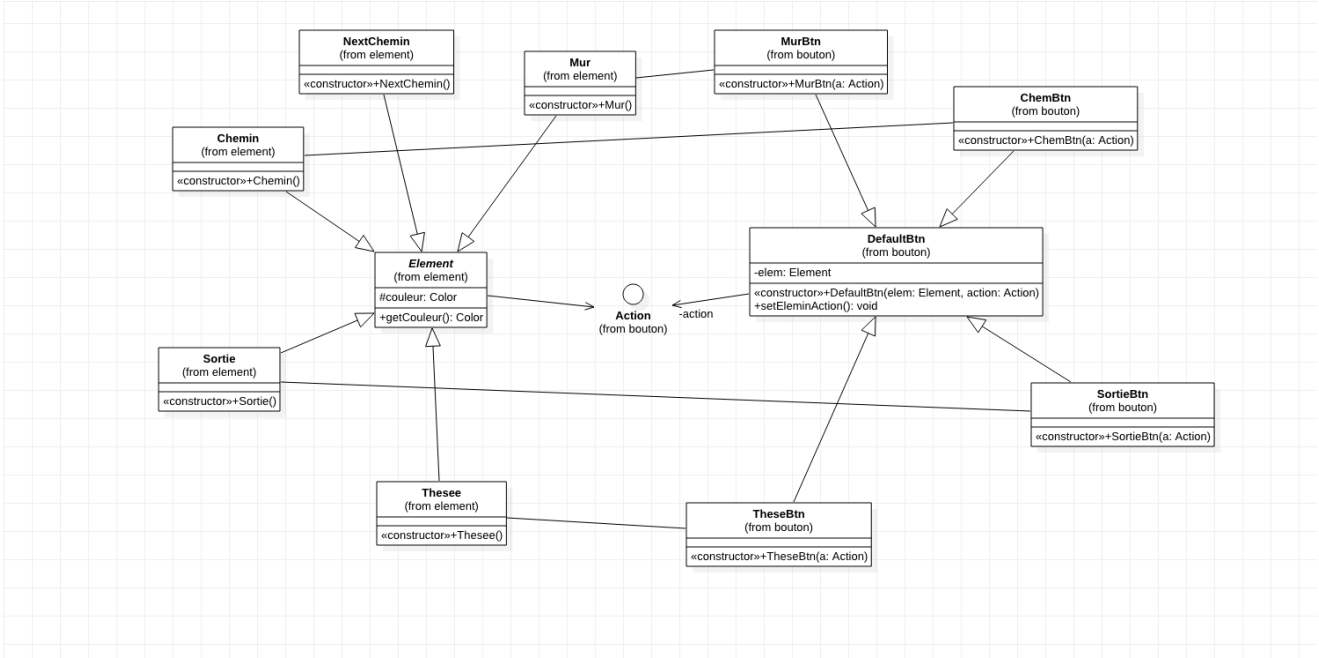
Contrairement au pop-up de fin générer à la fin d'un comportement automatique, le pop-up concernant le comportement manuel est issu du JOptionPane et est un information_message.

Ce choix de prendre un pop-up simple qui ne ferme pas le programme a été fait car contrairement à la simulation automatique, la grille est encore affichée dans le mode manuel, ainsi l'utilisateur peut relancer une grille aléatoire et ainsi retester l'algorithme de son choix et le comportement de son choix. Il peut également sauvegarder la grille.

PRÉSENTATION DE LA STRUCTURE DU PROGRAMME

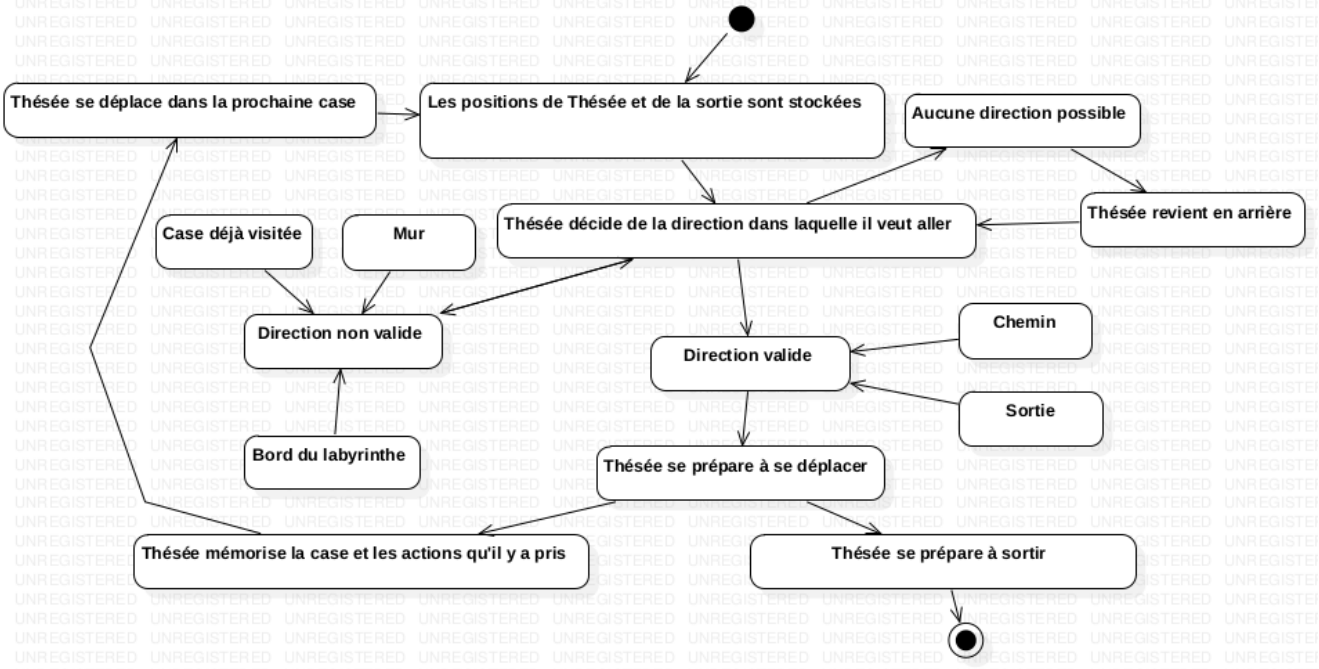



Bouton
(from bouton)
-id: int
«constructor»+Bouton(id: int)
+getId(): int
ValidationChoix
(from listener)
-grille: GrilleInterface
-bouton: Bouton
-bout: int
-algo: Algo
«constructor»+ValidationChoix(grille: GrilleInterface)
+mouseClicked(e: MouseEvent): void
+mousePressed(e: MouseEvent): void
+mouseReleased(e: MouseEvent): void
+mouseEntered(e: MouseEvent): void
+mouseExited(e: MouseEvent): void
Choix
(from listener)
-grille: GrilleInterface
+DETER: String = "Déterministe" {readOnly}
+ALEA: String = "Aléatoire" {readOnly}
+MANUEL: String = "Manuel" {readOnly}
+AUTO: String = "Automatique" {readOnly}
«constructor»+Choix(grille: GrilleInterface)
+actionPerformed(e: ActionEvent): void
Cell
(from grille)
-index: int
-elem: Element
-position: Position
-cardinal: boolean[*]
«constructor»+Cell(index: int, coloration: Coloration)
«constructor»+Cell(index: int, elem: Element)
+getIndex(): int
+setPropriete(elem: Element): void
+getPropriete(): Element
+getPosition(): Position
+setPosition(position: Position): void
+getCardinal(i: int): boolean
+setCardinal(i: int, b: boolean): void
GrilleChargee
(from grille)
+BITS_PER_BYTE: int = 8 {readOnly}
-CHEMIN: int = 0 {readOnly}
-MUR: int = 1 {readOnly}
-THESEE: int = 2 {readOnly}
-SORTIE: int = 3 {readOnly}
-e: Element
-methode: String
-algo: String
-box: Element[*] {collection="List"}
-taille: byte
-thesee: byte[*]
-sortie: byte[*]
-etats: char[*]
~jouer: Bouton
~deplacer: Bouton
«constructor»+GrilleChargee()
-drawGrid(taille: byte, thesee: byte[*], sortie: byte[*], etats: char[*]): void
-menuBarre(f: Fenetre): void
-jouer(f: Fenetre): void
+getTaille(): int
+getAlgo(): String
+getMethode(): String
+setAlgo(algo: String): void
+setMethode(methode: String): void
+cacherFenetre(): void
+getCells(): Cell[*]
+getTheseeX(): byte
+getTheseeY(): byte
+getSortieX(): byte
+getSortieY(): byte
+getEtat(i: int): char
+getFenetre(): Fenetre
+getBoutonDemarrer(): Bouton
+getBoutonDeplacer(): Bouton
GrilleRand
(from grille)
-colonnes: int
-lignes: int
-e: Element
-box: Element[*] {collection="List"}
-grille: int[*,*]
-visited: boolean[*,*]
-CHEMIN: int = 0 {readOnly}
-MUR: int = 1 {readOnly}
-THESEE: int = 2 {readOnly}
-SORTIE: int = 3 {readOnly}
-methode: String
-algo: String
-posx: int
-posy: int
-theseex: int
-theseey: int
-sortiex: int
-sortiey: int
~deplacer: Bouton
~jouer: Bouton
«constructor»+GrilleRand(c: int, l: int)
-jouer(f: Fenetre): void
-menuBarre(f: Fenetre): void
-drawGrid(f: Fenetre): void
-arrayFill(): int[*,*]
-placerThesee(): void
-placerSortie(): void
-deplacement(): void
-moveLeft(): boolean
-moveRight(): boolean
-moveUp(): boolean
-moveDown(): boolean
+getTaille(): int
+getAlgo(): String
+getMethode(): String
+setAlgo(algo: String): void
+setMethode(methode: String): void
+cacherFenetre(): void
+getCells(): Cell[*]
+getTheseeX(): byte
+getTheseeY(): byte
+getSortieX(): byte
+getSortieY(): byte
+getEtat(in: int): char
+getFenetre(): Fenetre
+getBoutonDemarrer(): Bouton
+getBoutonDeplacer(): Bouton
GrilleInterface
(from grille)
Grille
(from grille)
+DETERMINISTE: int = 1 {readOnly}
+ALEA: int = 2 {readOnly}
+MANUEL: int = 3 {readOnly}
+AUTO: int = 4 {readOnly}
-algo: String
-methode: String
-colonnes: int
-lignes: int
-e: Element
-theseeLa: boolean = false
-sortieLa: boolean = false
~deplacer: Bouton
~jouer: Bouton
«constructor»+Grille(c: int, l: int)
-jouer(f: Fenetre): void
+createGrid(f: Fenetre): void
-actionBarre(f: Fenetre): void
-menuBarre(f: Fenetre): void
+saveElem(e: Element): void
+getElem(): Element
+theseeLa(bool: boolean): void
+isTheseeLa(): boolean
+sortieLa(bool: boolean): void
+isSortieLa(): boolean
+getTaille(): int
+getAlgo(): String
+getMethode(): String
+setAlgo(algo: String): void
+setMethode(methode: String): void
+cacherFenetre(): void
+getCells(): Cell[*]
+getTheseeX(): byte
+getTheseeY(): byte
+getSortieX(): byte
+getSortieY(): byte
+getEtat(in: int): char
+getFenetre(): Fenetre
+getBoutonDemarrer(): Bouton
+getBoutonDeplacer(): Bouton
Fenetre
(from labyrinthe)
-APP_NAME: String = "Algorithme Ariane" {readOnly}
+SCREEN_WIDTH: int = 300 {readOnly}
+SCREEN_HEIGHT: int = 300 {readOnly}
«constructor»+Fenetre()
«constructor»+Fenetre(width: int, height: int)
+menu1(): void
+menu2(): void
+saisi(): void
+saisiRand(): void
+visible(): void
+clearContent(): void
Action
(from bouton)
GestionMenu
(from listener)
-grilleC: GrilleChargee
-grille: GrilleInterface
-grillr: GrilleRand
-gril: Grille
«constructor»+GestionMenu(grillr: GrilleRand)
«constructor»+GestionMenu(grilleC: GrilleChargee)
«constructor»+GestionMenu(gril: Grille)
+actionPerformed(e: ActionEvent): void
DefaultBtn
(from bouton)
-elem: Element
«constructor»+DefaultBtn(elem: Element, action: Action)
+setEleminAction(): void
MurBtn
(from bouton)
«constructor»+MurBtn(a: Action)
TheseBtn
(from bouton)
«constructor»+TheseBtn(a: Action)
SortieBtn
(from bouton)
«constructor»+SortieBtn(a: Action)
ChemBtn
(from bouton)
or»+ChemBtn(a: Action)
-cells
{collection="List"}
-cellToRegister
-f
-fenetre
-action



Détail du diagramme des classes des Element correspondant aux boutons

EXPOSITION DE L'ALGORITHME DÉTERMINISTE



Ainsi, lorsque l'algorithme déterministe est appelé, Thésée et la sortie sont récupérés à l'aide des méthodes searchThesee() et searchSortie() au sein de la méthode mazeDeterAuto() ou mazeDeterManuel(). Jusqu'à ce que la sortie ait été trouvée (s'il y en a une), Thésée va chercher à se déplacer à l'aide de la méthode deplacementDeter(). Cette méthode se compose de 2 étapes : la recherche d'une nouvelle direction et le déplacement.

LA RECHERCHE DE LA NOUVELLE DIRECTION

Lors de cette étape, l'algorithme détermine dans quelle direction Thésée doit se diriger à l'aide de la méthode getDeterministDirection(). Cette direction est déterminée en fonction de la position de Thésée par rapport aux bords du labyrinthe, de la connaissance de Thésée des états des cases jouxtant celle où il se situe (grâce aux méthodes getCardinal() et setCardinal()) et de sa connaissance des cases visitées (la file visited_cells). Thésée évaluera les possibilités d'aller dans une case précise toujours selon la même séquence : Nord, Ouest, Sud, Est. Une fois la direction choisie, la case correspondant à cette direction est désignée pour être la prochaine case de Thésée à l'aide de la méthode getCellByPosition. Si aucune direction ne peut être choisie (3 murs autour de Thésée et le chemin d'où il vient par exemple), Thésée revient en arrière à l'aide de la méthode pop() appliquée à la pile, et réitère le même procédé. Si Thésée est revenu au point de départ en revenant sur ses pas, Thésée arrête de chercher la sortie car elle n'existe pas.

LE DÉPLACEMENT

Thésée se place dans la case déterminée si elle existe, puis la nouvelle case est ajoutée à la pile et à la file. On a aussi une incrémentation du nombre de déplacement pour donner à la fin le nombre de déplacement effectués par Thésée pour atteindre la sortie.

CONCLUSION PERSONNELLE

Conclusion de Anne-Sophie Besnard

Pour conclure, je retiens surtout qu'il est assez difficile de travailler en groupe sur du java. Étant donné qu'on est assez libre dans l'utilisation et la création des classes, des interfaces, avec l'héritage et l'implémentation, on peut facilement se perdre dans le code de quelqu'un d'autre car la création des classes n'aurait pas forcément été la même.

De plus, nous n'avons pas respecté le fait que la grille soit totalement aléatoire et que la sortie puisse être inaccessible pour Thésee. Ayant compris cela un peu tard, nous n'avons pas voulu changer cela pour éviter que le programme ne marche plus et que nous n'ayons pas le temps de rendre correctement le projet.

C'est pourquoi, dans la classe GrilleRand nous avons des méthodes qui simulent aléatoirement un chemin et on place la sortie aléatoirement sur ces chemins visités. Ainsi il y a toujours au moins un chemin accessible entre Thésée et la sortie au prix d'une grille générer pas aussi aléatoirement qu'elle aurait dû l'être. En effet, en fonction du random qui gère le nombre de déplacement du chemin possible et du nombre de Mur et Chemin à la base dans la grille (aussi aléatoirement), il est possible d'avoir aucun Mur dans la grille.

De plus, la création du Makefile avec les dépendances a été laborieux. A cause du nombre d'erreur et l'impossibilité d'arriver à un résultat correct, j'ai dû me résoudre à faire une Makefile sans les dépendances mais qui marchent. Cependant il se trouve dans le meme dossier que le Main.

Cependant, ce projet m'a permis d'apprendre à bien séparer mon code en méthode, ce qui nous a permis de rendre plus adaptable notre code.

Nous avons commencé par créer l'algorithme aléatoire avec le déplacement automatique puis manuel, ainsi notre code était découpé en plusieurs méthodes spécifiques à l'algorithme aléatoire mais ce dernier étant assez découpé, nous avons pu baser notre algorithme déterministe ainsi que ses déplacements sur la même structure que les méthodes qui gère l'autre algorithme.

J'ai également appris l'utilisation concrète des interfaces. Ayant créée la classe Grille (qui contient la grille manuelle/vide) et la classe GrilleRand séparément et Romain la classe GrilleChargee, afin de faciliter l'utilisation des algorithmes dans les grilles, j'ai créé une interface GrilleInterface qui permet de faire l'intermédiaire entre la classe Algo et les trois grilles.

Conclusion de Romain Lechartier

En conclusion, ce projet a été pour moi l'occasion de comprendre la difficulté de travailler en binôme, encore plus que lors du projet en C.
En effet, nous n'avons pas réalisé un Diagramme de Classes dès le début de cet exercice et nous n'avons pas non plus décider ensemble des définitions de types nécessaires. De ce fait, la fusion de nos travaux a été très complexe puisque j'ai du implémenter mon code au sein en respectant ses choix, et ainsi, modifier de nombreuses parties de mon code ou bien ajouter des données et méthodes dans le sien.

De plus, j'ai eu des difficultés à créer une classe pour l'observateur destiné à agir sur le timer qui nous sert à contrôler le déplacement de Thésée, j'ai donc du le laisser en tant que classe anonyme.
Pour moi, la difficulté n'a vraiment pas été l'algorithme déterministe, ou la lecture de fichier, ou encore la sauvegarde, c'est bien la mise en commun du travail. Aussi, le temps nécessaire à la génération de la javadoc a été une surprise pour moi, c'est un travail de longue haleine.
Enfin, je dirais que j'ai commencé à me mettre sérieusement au travail beaucoup trop tard ce qui a retardé notre avancement dans le projet.
Ce projet m'a donc aidé dans mon apprentissage en me montrant l'importance de la communication entre membres d'un différent projet, ainsi que le temps nécessaire à l'optimisation d'un programme et de sa documentation.