Informatique - Rapport - Surizarinku

Quentin CHAN-WAI-NAM - Groupe 1

27 mars 2015

Table des matières

1 Présentation et analyse

1.1 Présentation du sujet choisi

L'objectif est d'écrire un programme Java permettant de jouer de manière conviviale au Surizarinku, aussi appelé Slitherlink ou encore Dilemne de points.

1.2 Description du jeu

Disposant d'une grille plus ou moins grande, le but est de déterminer un chemin continu, vertical ou horizontal, formant une seule boucle et faisant le tour des cases et respectant le chiffre 0, 1, 2 ou 3 indiquant le nombre de côtés pleins de la case considérée.

On suppose qu'il n'y a pas de croisement, le chiffre 4 est donc exclu.

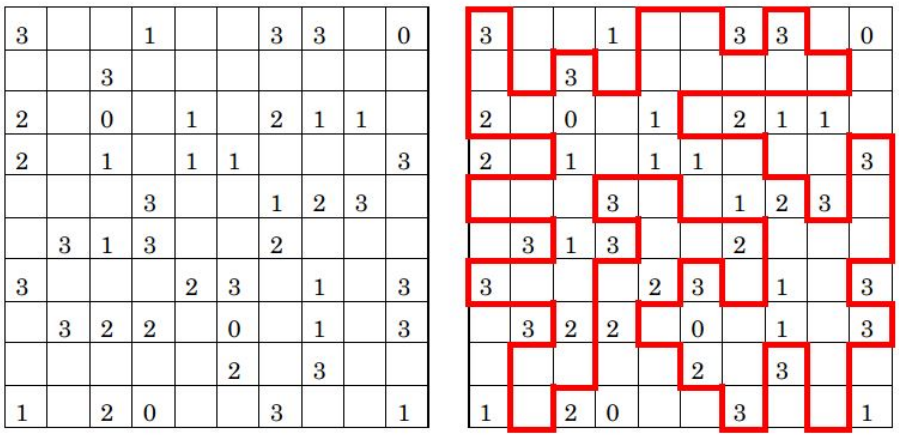


FIGURE 1 – Un exemple de grille de Surizarinku

1.3 Résultat

On a abouti à un programme permettant de jouer au jeu avec des grilles 10×10, avec certaines fonctionnalités :

1. Proposer une interface graphiquement praticable avec des couleurs et des menus
2. Proposer un jeu à la souris et au clavier : souris pour sélectionner des noeuds, clavier pour se déplacer. Cf section 2 pour détails
3. Pouvoir vérifier qu'une solution proposée par l'utilisateur ne comporte pas d'erreur et clore le jeu s'il a trouvé une solution qui convient
4. Proposer un système permettant de garder en mémoire les actions réalisées par le joueur, pour qu'il puisse "revenir en arrière" aussi loin qu'il le veuille ; et de même revenir en avant cas échéant
5. Permettre de charger des grilles depuis un fichier
6. Permettre la création par étapes et la sauvegarde de grilles : dessiner un chemin plus ou moins compliqué, puis sélectionner les cases où l'on souhaite mettre un chiffre, ce dernier étant calculé automatiquement.

2 Analyse de la solution

2.1 Description du fonctionnement

Le programme propose deux modes de fonctionnement :

Le mode Jeu

- Le joueur charge une grille existante d'après un fichier. Les numéros dans les cases sont fixés.
- Il peut tracer un chemin passant par les côtés des cases.
- Il dispose de fonctionnalités permettant d'annuler ou refaire une action, de montrer une solution puis de la masquer, de vérifier la justesse d'un chemin puis de masquer ces vérifications, d'effacer le chemin qu'il a construit.

Le mode Création

- Le joueur crée sa propre grille de jeu. Il part d'une grille vide (ou d'une grille existante).
- Il peut cliquer au centre d'une cellule pour indiquer s'il souhaite que cette cellule soit remplie d'un nombre ou non. Le nombre de côtés indiqué est automatiquement mis à jour en fonction du chemin que l'utilisateur trace.
- Il dispose encore des fonctions permettant d'annuler ou refaire une action.
- Enfin, il peut enfin enregistrer la grille dans un fichier du nom qu'il souhaite.

Mécanismes de fonctionnement d'une grille

Le joueur choisit un "noeud" (intersection de lignes dans le tableau) où commencer. Ce noeud devient "actif".

Le joueur peut ainsi utiliser les **flèches du clavier** : haut, bas, gauche, droite, pour relier le point courant au point situé en haut, en bas, à gauche ou à droite. Passer une nouvelle fois sur un chemin déjà tracé fait l'action inverse (et donc l'efface).

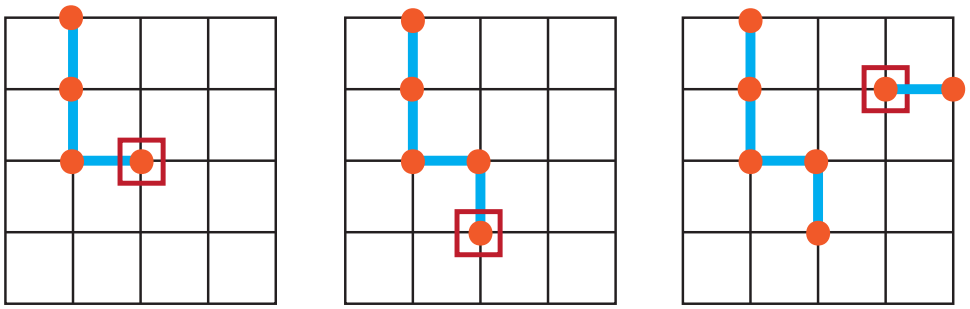


FIGURE 2 – En bleu, côtés pleins ; en orange, points utilisés ; encadré en rouge, point actif

2.2 Structure du programme

Tout le code du programme est personnel et a été réalisé seul.

Les données de la grille de jeu

La base du programme est constituée de la **classe Grille**, qui comporte le terrain de jeu à proprement parler. Il est constitué de plusieurs tableaux contenant :

- Le nombre de côtés pleins d'une cellule de la grille
- La donnée du côté bas de la cellule
- La donnée du côté droit de la cellule

Pour pouvoir couvrir tous les côtés des cellules de la grille, et notamment les bords haut et gauche de la grille, on doit "élargir" la grille de deux bandes supplémentaires de cases, en haut et à gauche.

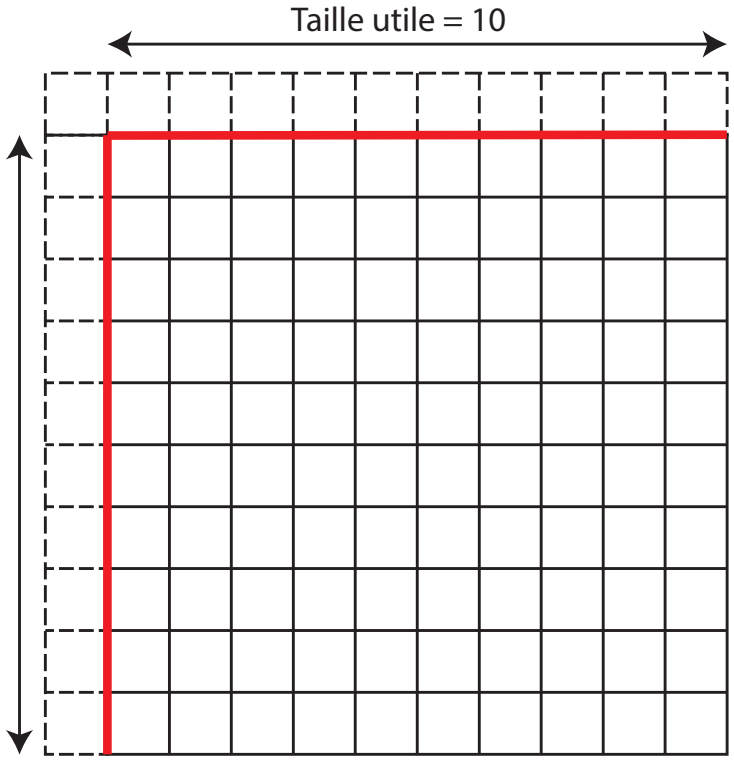


FIGURE 3 – Pour stocker les données des côtés en rouge (et du "point actif" de la classe Session), on rajoute des cellules inertes

Les espaces de dessin et les données du jeu actif

Les classes qui intègrent la Grille sont la classe **Session** et ses classes filles **Session_jeu et Session_creation**. C'est dans ces classes que sont stockés tous les éléments de dessin : on dessine la grille, le chemin tracé. L'utilisateur interagit ici avec les éléments de la grille. On stocke notamment

- La donnée du point actif : il s'agit du coin inférieur droit d'une cellule de la grille.
- Les piles contenant les actions effectuées et annulées, qui permettent à l'utilisateur de revenir en arrière.

La fenêtre de jeu, les menus et options

Enfin, la classe **Surizarinku** est la classe qui contient tous les boutons, et méthodes pour lire des grilles, écrire les grilles, afficher ou non des solutions, changer de mode de jeu, ainsi que la JFrame utilisée pour l'IHM.

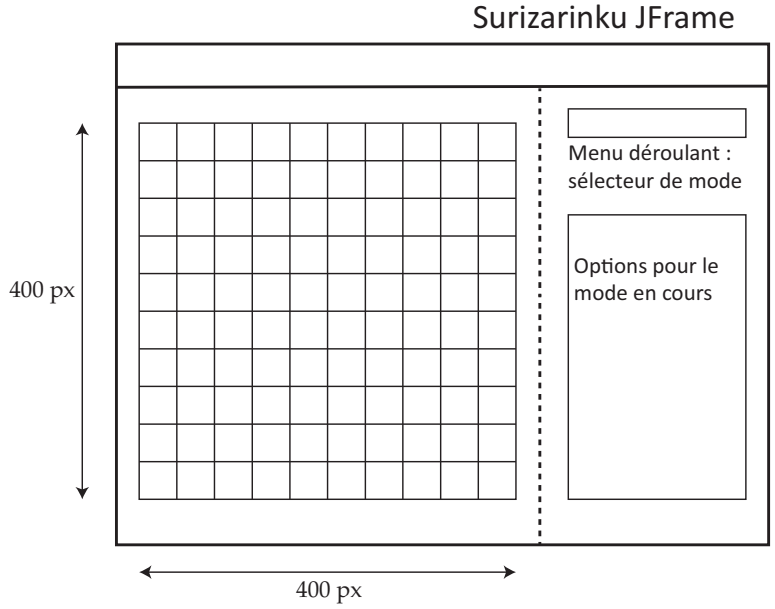


FIGURE 4 – Croquis de la disposition de l'IHM

Elle implémente les fonctions suivantes :
- Changer de mode entre Jeu et Création
- Annuler/refaire action
- Dans le mode jeu, lire une grille
- Vérifier si la grille et correcte (et mettre fin au jeu cas échéant)
- Effacer le tracé du joueur
- Dans le mode création, remplir de manière aléatoire les indications dans les cases selon une proportion déterminée de cases remplies
- Enregistrer une grille

Enregistrement et lecture de grilles en fichier texte

La liste de grilles est lue dans le fichier texte LISTEGRILLES.txt. Chaque grille est enregistrée dans un format qui indique le nom du fichier, la date et l'heure de création et comporte :
- La donnée du chiffre indiqué dans chaque cellule non inerte
- La donnée des côtés bas des cellules pour le chemin "solution"
- La donnée des côtés droits des cellules pour le chemin "solution"

Pour le premier point, on a un tableau de chiffres 0, 1, 2, 3 ou 9, ce dernier signifiant qu'aucune indication n'est donnée au joueur pour cette case (case blanche). Pour les deux autres points, on a 0 ou 1 qui sont booléens : le côté est plein si 1, sinon 0.

Certes, il est en l'état possible de tricher, en allant regarder la solution, qui est en clair dans le fichier. Il serait possible de modifier cela en écrivant la solution dans un fichier binaire crypté, fonctionnalité qui n'est pas implémentée ici.

3 Description des classes et algorithmique

3.1 Classes auxiliaires

3.1.1 Classe Action

```
public class Action {
   int x;
   int y;
   Direction d;
   enum Direction {
      H, B, G, D;
   }
   public Action(int coordX, int coordY, Direction directD) {
      x = coordX;
      y = coordY;
      d = directD;
   }
}
```

Cette classe sert à décrire entièrement une action du joueur : coordonnées du point de départ et direction de déplacement.

3.1.2 Classe Dates

```
public class Dates {
   // Choix de la langue francaise
   static Locale locale = Locale.getDefault();
   static Date actuelle = new Date();

   // Definition du format utilise pour les dates
   static DateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");

   // Donne la date
   public static String date() {
      String dat = dateFormat.format(actuelle);
      return dat;
   }
}
```

Cette classe permet de donner la date et l'heure en format voulu.

3.2 Classe Surizarinku : Fenêtre de jeu

```
public class Surizarinku extends JFrame implements ActionListener {
   private Session_jeu sessionJeu;
   private Session_creation sessionCreation;
```

Cette classe est une JFrame qui sert à l'ensemble du jeu. Elle contient tous les boutons du jeu, et tous les JPanel. En particulier :

- la Session_jeu sessionJeu
- la Session_creation sessionCreation

qui correspondent aux espaces de dessin des grilles correspondant aux deux modes de fonctionnement.

```
   /** Constructeur */
   public Surizarinku(Session_jeu sj, Session_creation sc) {

   /** Met à jour la liste des grilles depuis le fichier ad-hoc */
   private void LireListeGrilles() {

   /** Renvoie une grille lue depuis le fichier correspondant */
   private Grille LireGrille(String s) {

   /** Enregistre la grille de la Session_creation dans un fichier portant le
   * nom dans le champ texte. */
   private void EnregistrerGrille(String s) {

   /** Supprime la ligne s dans le fichier de la liste des grilles */
   private void SupprimerGrille(String s) {

   public void actionPerformed(ActionEvent evt) {

   public void itemStateChanged(ItemEvent evt) {

   /** Bascule vers la session de jeu */
   private void AfficheSessionJeu() {

   /** Bascule vers la session création */
   private void AfficheSessionCreation() {

   }
```

3.3 Classe Session et dérivés : environnements jeu et de dessin

La classe Session comporte : les données de la session active (position du curseur du joueur, piles des actions effectuées/annulées, et tous les élements de dessin.

Les classes filles Session_jeu et Session_creation apportent en plus des écouteurs clavier et souris particuliers à chacun des usages jeu ou mode création.

3.3.1 Classe Session

```
public class Session extends JPanel implements ActionListener {

    /** CARACT TERRAIN DE JEU */
    Grille grille;

    /** Indicateur Solution Active */
    protected boolean solutionEstActive = false;

    // Coordonnées du point actif
    protected String coordonnees = new String("0");
    protected int pointActifX = 0;
    protected int pointActifY = 0;
    protected int keyEventNum = 0;

    // Caractéristiques graphiques
    protected final int MARGE = 50;

    // Piles
    protected Stack<Action> pileActEffectuees;
    protected Stack<Action> pileActAnnulees;
```

Seuls les attributs non privés sont listés. Les attributs les plus importants étant :

- La Grille grille, qui contient les données relatives au remplissage de la grille et du chemin tracé
- Le boolean solutionEstActive qui permet de déterminer s'il faut faire apparaître la solution lorsque l'on peint
- Les Stack<Action> pileActEffectuees et pileActAnnulees qui contiennent toutes les actions effectuées ou annnulées

```
    // Constructeurs
    public Session(Grille j) {
    public Session(int lar, int hau) {

    // Dessin
    public void paint(Graphics g) {

    // Accès aux dimensions de la grille
    protected int RenvoyerLargeur() {
    protected int RenvoyerHauteur() {

    // Remise à zéro
```

```
    protected void Reset() {

    // Méthodes de la fonction annuler action/refaire action
    protected void AnnulerAction() {
    protected void RefaireAction() {

    // Change la visibilité de la solution : active ou non
    protected void ToggleSolutionEstActive() {

    }
```

Les méthodes importantes de Session sont les méthodes AnnulerAction et RefaireAction qui utilisent les piles pour annuler/refaire la dernière action de l'utilisateur. La méthode ToggleSolutionEstActive déclenche ou non le dessin du chemin "solution" enregistré dans la grille.

3.3.2 Classe Session_jeu

```
public class Session_jeu extends Session {
    public Session_jeu(Grille j) {
    public Session_jeu(int lar, int hau) {
    private class EcouteurSouris extends MouseAdapter {
    private class EcouteurClavier extends KeyAdapter {
}
```

Cette classe hérite de Session et ajoute en sus des écouteurs souris et clavier particuliers au mode jeu (gestion des clics, où et leurs effets, gestion des touches).

3.3.3 Classe Session_creation

```
public class Session_creation extends Session {
    public Session_creation(Grille j) {
    public Session_creation(int lar, int hau) {
    public void paint(Graphics g) {
        grille.RafraichirCotes();
        super.paint(g);
    }
    public void ResetNbCotesPleins() {
    private class EcouteurSouris extends MouseAdapter {
    private class EcouteurClavier extends KeyAdapter {

}
```

La méthode paint redéfinie rafraîchit le nombre écrit dans la case en fonction des côtés pleins. On a une méthode pour vider tous les nombres dans les cases (qui sont définitifs en mode jeu), ce qui constitue la différence majeure. Les écouteurs sont adaptés (on peut par exemple cliquer sur une case pour en faire apparaître l'indication).

3.4 Classe Grille

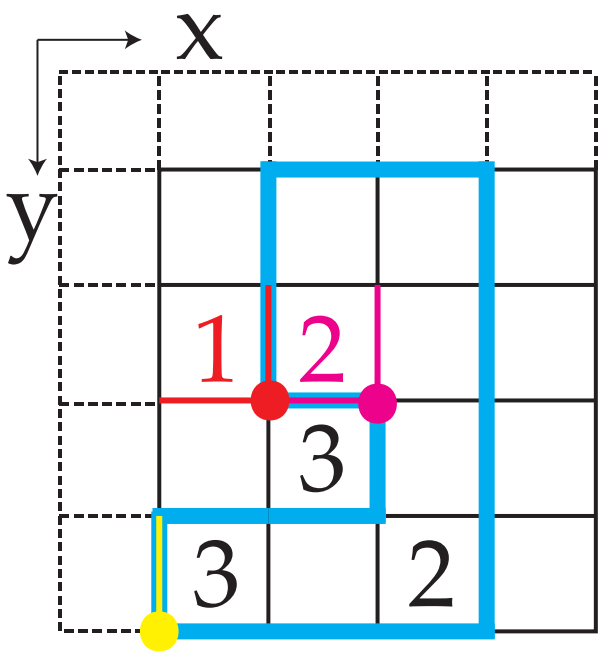


FIGURE 5 – Exemple 4×4 ; dans les tableaux, les coordonnées sont les mêmes pour les côtés rouges, ainsi que pour le point rouge ; de même pour les côtés et le point violets. Les cellules en pointillé ne sont pas affichées, mais contiennent les données des noeuds au bord et des frontières haute et gauche : par exemple, tout en bas à gauche, le côté jaune et le noeud jaune.

```
public class Grille {

   /** Largeur de la grille */
   protected final int TAILLEL;
   /** Hauteur de la grille */
   protected final int TAILLEH;

   /** Tableau contenant les nombres affichés dans les cases */
   protected int[][] nbCotesPleins;
   /** Tableau contenant la donnée du côté bas de la case */
   protected boolean[][] coteBEstPlein;
   /** Tableau contenant la donnée du côté droit de la case */
   protected boolean[][] coteDEstPlein;
   /** Tableau indiquant si les noeuds sont faux */
   protected boolean[][] cheminEstFaux;
   /** Tableau indiquant si les cellules sont fausses */
   protected boolean[][] celluleEstFausse;

   /** Tableau final contenant la solution, côtés bas */
   protected final boolean[][] SOLUTION_COTEB_EST_PLEIN;
   /** Tableau final contenant la solution, côtés droit */
```

```
protected final boolean[][] SOLUTION_COTED_EST_PLEIN;
```

Le type énuméré Diff est un indicateur de difficulté. Il est implémenté dans Grille et non implémenté dans les classes plus hautes ; on pourrait l'utiliser pour proposer des grilles de niveaux différents selon les choix de l'utilisateur.

- Le tableau nbCotesPleins contient des entiers 0, 1, 2, 3 ou 9 correspondant aux chiffres dans les cellules.
- coteBEstPlein et coteDEstPlein sont des tableaux de booléens indiquant si le chemin tracé par l'utilisateur passe ou non par le côté bas/droit de la cellule de coordonnées correspondantes.
- cheminEstFaux et celluleEstFausse sont des tableaux remplis par les méthodes de la classe Grille et lus par les méthodes de la classe Session pour dessiner les endroits problématiques dans la solution proposée par les utilisateurs.
- SOLUTION_COTEB_EST_PLEIN et SOLUTION_COTED_EST_PLEIN sont les tableaux contenant le chemin "solution"

```
// Parmi les constructeurs, celui qui sera utilisé par les autres classes est :
public Grille(int tL, int tH) {

/**
 * Vérifie que le chemin proposé par l'utilisateur est cohérent,
 * c'est-à-dire qu'il n'y a pas de croisement et que c'est bien une boucle.
 * Cela revient en fait à vérifier que pour chaque noeud, il y a exactement
 * zéro ou deux attributs parmi les cotés haut, bas, gauche, droit marqués
 * ``true''. Si une erreur est détectée, on met en évidence le noeud posant
 * problème (via le tableau cheminEstFaux). Si les chemins sont fermés sur
 * eux-mêmes, on vérifie s'il n'y a qu'un seul chemin. Pour cela, on choisit
 * un point de départ puis on suit le chemin qui passe par ce point jusqu'à
 * revenir au point de départ. Par la suite, on vérifie qu'il n'y a aucun
 * autre tracé enregistré.
 */
protected boolean CheminEstFaux() {

/**
 * Vérifie que le chemin proposé par l'utilisateur vérifie bien la
 * contrainte du nombre de côtés adjacents aux cases contenant un chiffre
 * différent de 9. Pour cela, on prend les cellules une par une. Si elle
 * contient un chiffre différent de 9, on compte les côtés pleins (via la
 * méthode RenvoyerCotes. Si une erreur est détectée, on met en évidence la
 * cellule posant problème (via l'attribut celluleEstFausse).
 */
protected boolean CotesSontFaux() {

/**
 * Vide les tableaux relatifs aux vérifications des côtés et des chiffres,
 * pour que la grille soit dessinée "normalement".
 */
protected void EffacerVerifier() {

/** Change l'état du côté droit de la case concernée i,j */
protected void ChangeCoteD(int i, int j) {
```

```
/** Change l'état du côté bas de la case concernée i,j */
protected void ChangeCoteB(int i, int j) {

/** Renvoie le nombre de côtés pleins bordant la case i,j de la grille */
protected int RenvoyerCotes(int i, int j) {

/** Rafraîchit le nombre de côtés pleins de toutes les cases de la grille */
protected void RafraichirCotes() {

/**
 * Détermine aléatoirement (probabilité d) si la chaque case du tableau
 * donnera une restriction au chemin de l'utilisateur
 */
protected void RemplirNbCotesAlea(double d) {

/**
 * Bascule le chemin solution enregistré vers le chemin tracé par
 * l'utilisateur.
 */
protected void TracerCheminSolution() {

/**
 * Remet à zéro l'état des côtés des cases
 */
protected void ResetCotesPleins() {
```

3.5 Vérification d'une solution : principe

Il n'y a pas unicité de solution à une grille de Surizarinku la plupart du temps. Pour valider une solution et mettre fin au jeu, le programme doit donc s'assurer de la cohérence de cette dernière indépendamment de la solution stockée avec la grille.

Comme cette partie du code est dense, son fonctionnement est détaillé ci-après.

Vérification du nombre de côtés pleins par case

Dans un premier temps, on vérifie si les numéros dans les cases sont bien respectés par le chemin. Si ce n'est pas le cas, on colorie le numéro en orange.

C'est l'objet de la méthode **CotesSontFaux** de la classe Grille (page 26), qui renvoie un booléen.

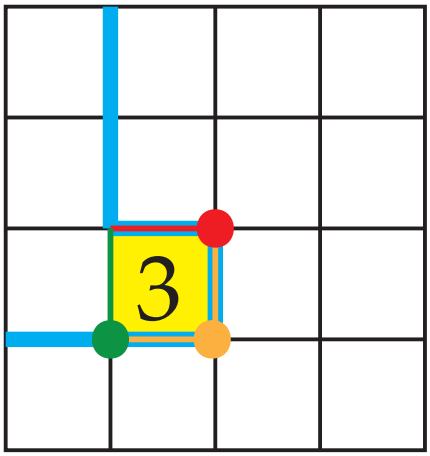


FIGURE 6 – Vérification du nombre de côtés pleins. En jaune, la case concernée. On regarde les côtés adjacents : les côtés beiges appartiennent à la case dont le coin inférieur droit est beige, donc la case active. De même, le côté rouge appartient à la case supérieure et le côté gauche à la case gauche.

On regarde chaque case ; l'algorithme est simple car il n'y a pas de problème avec les indices au bord (on regarde la cellule active, la cellule en haut et la cellule à gauche : les cellules inertes des bords haut et gauche ne sont donc pas activées).

Vérification de la cohérence du chemin

Ensuite, on vérifie que le chemin tracé par l'utilisateur :

– Est bien refermé sur lui même

– Et si c'est bien le cas, qu'il n'y a qu'une seule boucle (pas deux circuits séparés par exemple)

C'est l'objet de la méthode **CheminEstFaux** de la classe Grille (page 22), qui renvoie un booléen.

Dans un premier temps, on prend chaque noeud de la grille (donc toutes les cases, même inertes) et on vérifie si :

– Soit aucun côté arrivant au noeud n'est rempli : le chemin ne passe pas par le noeud

– Soit exactement deux des côtés arrivant au noeud sont remplis : le chemin passe par le noeud (et en repart)

Si l'on est pas dans ces situations, c'est que le chemin n'est pas fermé. Il y a donc une anomalie.

L'implémentation dans le code nécessite un traitement particulier pour les bords bas et droit, ainsi que le noeud inférieur droit, car les côtés à vérifier sont indicés différemment.

Dans un second temps, et si tous les chemins sont fermés, on vérifie qu'il n'y a qu'une seule boucle dans toute la grille. Pour ce faire :

1. On parcourt la grille jusqu'à trouver un morceau de chemin
2. On suit ce chemin jusqu'à le refermer. On sait qu'il est fermé grâce à l'étape précédente. On garde en mémoire la donnée de ce chemin.
3. On parcourt à nouveau la grille en vérifiant qu'aucun autre côté n'appartenant pas au chemin détecté n'est plein. Si on en détecte un autre, c'est qu'il y a plus d'une boucle.

Une fois ces étapes terminées, on est assurés qu'il y a une seule boucle dans le chemin et que ce dernier vérifie les contraintes des numéros : la solution convient.

Lorsqu'on appuie sur le bouton "Vérifier le chemin" de la classe Surizarinku, on exécute ces méthodes ; si on reçoit true pour les deux méthodes, on notifie à l'utilisateur qu'il a réussi la grille, ce qui met fin au jeu.

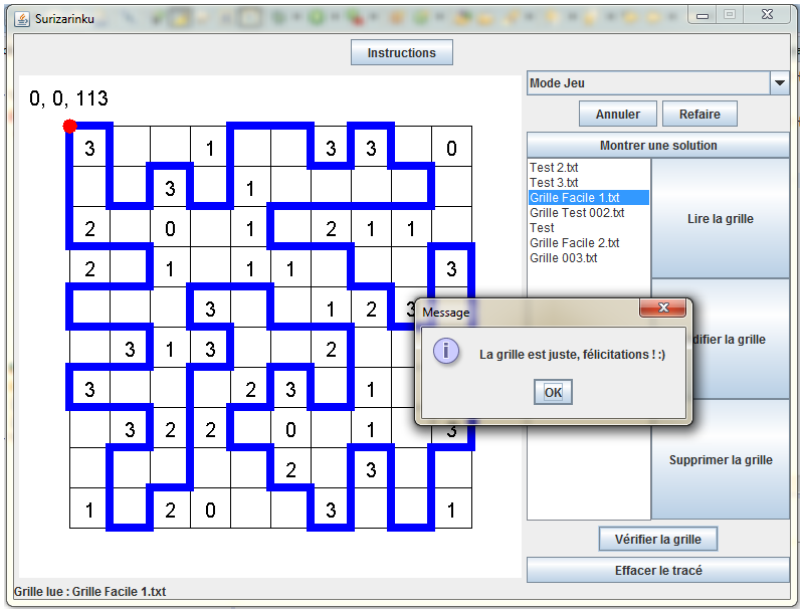
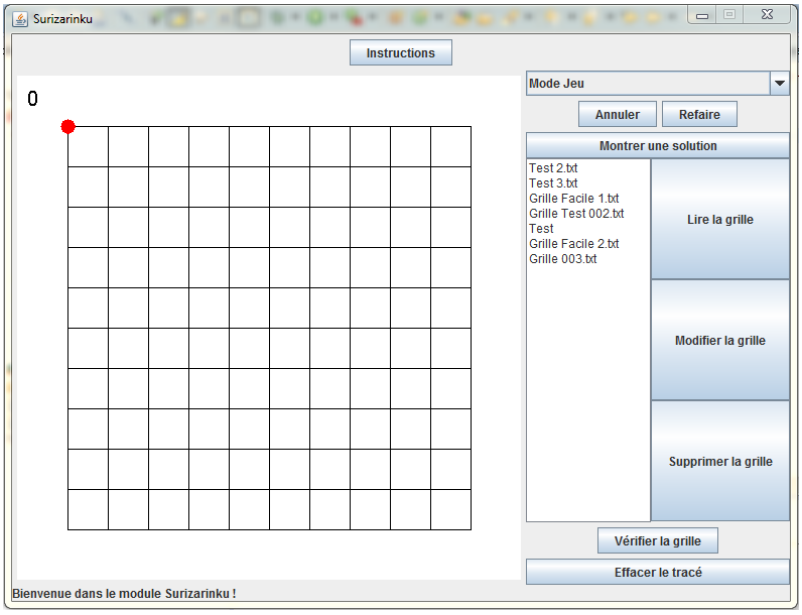


FIGURE 7 – Fin du jeu

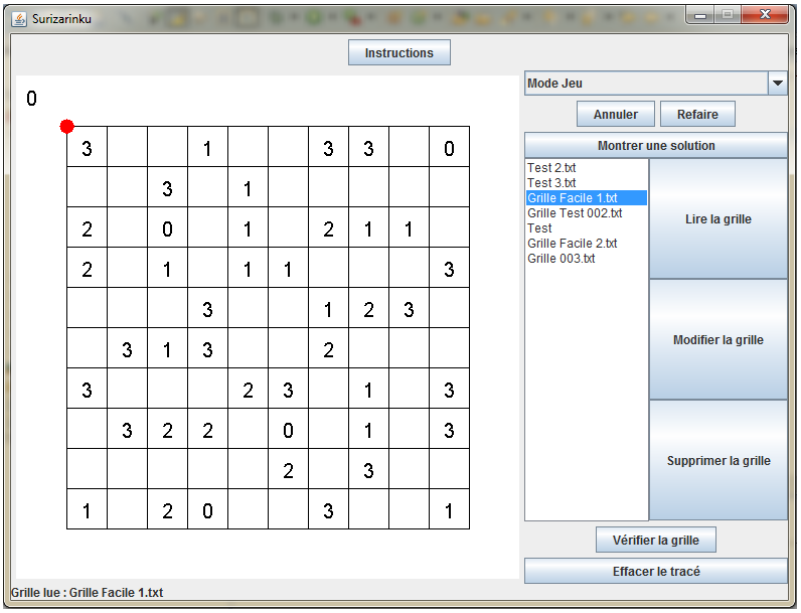
4 Notice d'utilisation

L'interface du jeu lors du lancement prend cette forme :

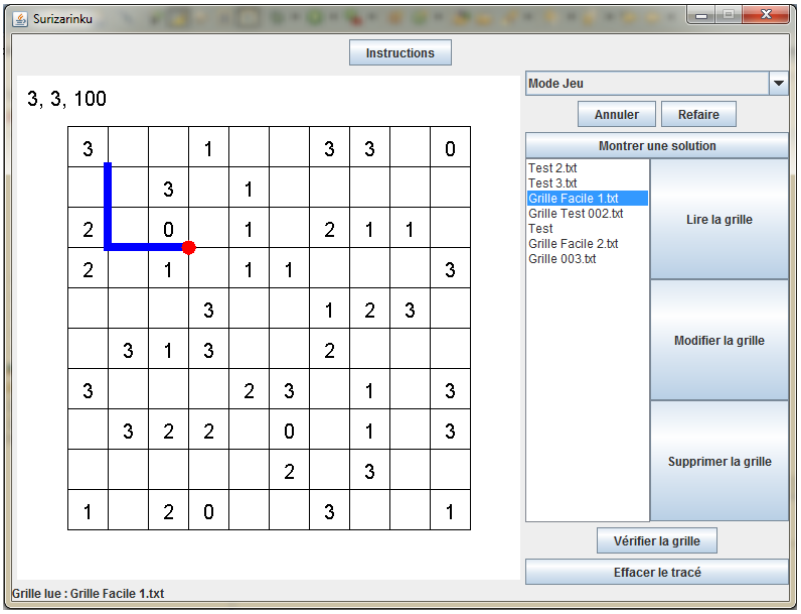


Le bouton "instructions" permet d'afficher une fenêtre qui explique les contrôles du jeu. On peut changer le mode de jeu grâce au champ de choix en haut à droite. Tout en bas, un label qui fournit un retour au joueur par rapport aux opérations effectuées.

4.1 Mode jeu

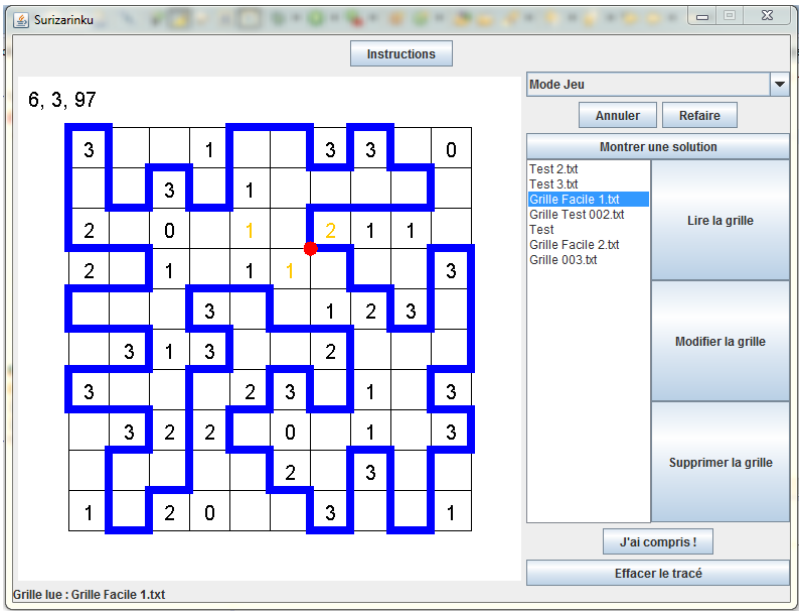


L'utilisateur, pour jouer, sélectionne une grille à charger dans la liste des grilles. Il appuie sur le bouton "Lire une grille". La grille est alors remplie avec les chiffres correspondant à la grille.



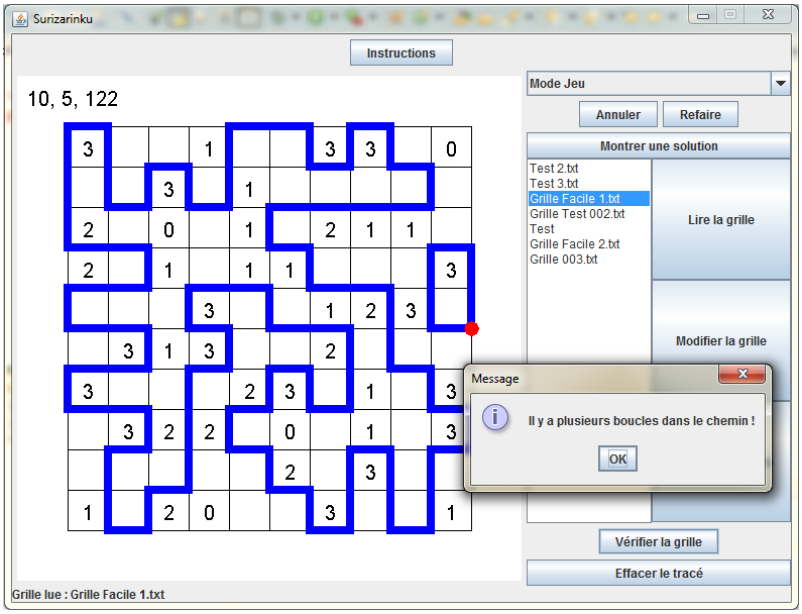
Il peut alors tracer son chemin avec les touches Z, Q, S, D (pour haut, gauche, bas, droite). Le chemin est tracé à partir du "point actif" (en rouge sur la grille), qui peut être replacé par l'utilisateur par un clic gauche sur un noeud de la grille.

Les boutons "Annuler" (resp. "Refaire") annule la dernière action du joueur (resp. rétablit la dernière action annulée), c'est-à-dire que le point actif repart à l'endroit précédent et que le côté modifié revient à son état initial. Le bouton "Effacer le tracé" efface le chemin bleu et remet le point rouge en haut à gauche.

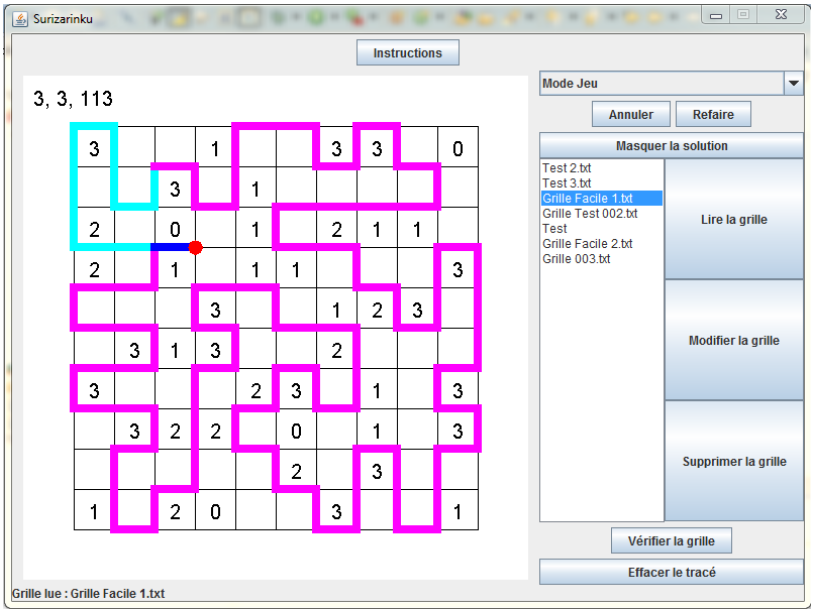


Une fois qu'il a tracé un chemin qui lui paraît juste, il peut cliquer sur le bouton "Vérifier la grille" qui lui indique les éventuelles erreurs : chemin non fermé, cases non respectées.

Si plusieurs boucles sont détectées (il ne doit y avoir qu'un seul chemin qui se referme sur lui-même), alors un message d'erreur apparaît. Pour faire disparaître les traits orange, l'utilisateur peut cliquer à nouveau sur le bouton, qui s'était transformé en "J'ai compris !".

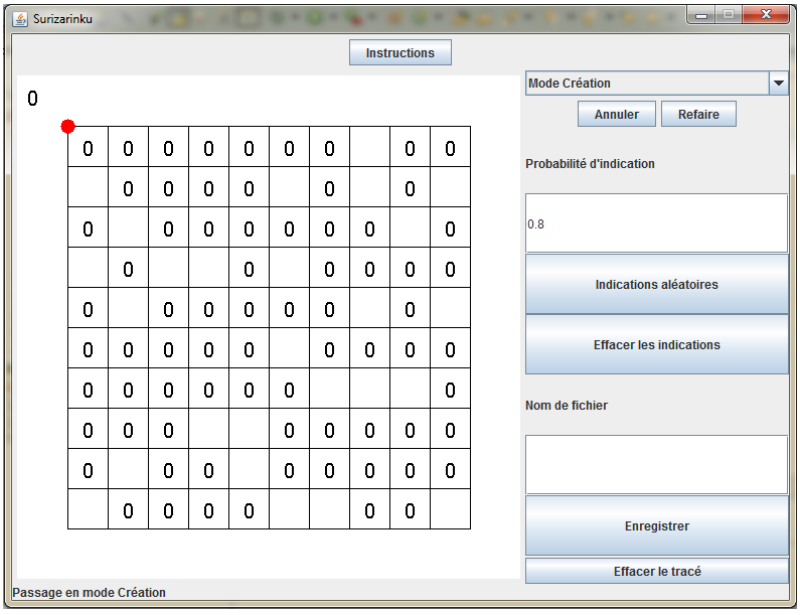


Si le joueur n'arrive pas à finir le jeu, il peut regarder la solution en cliquant sur le bouton "Montrer une solution". La solution apparaît tracée en violet. La couleur cyan signifie que le chemin avait été tracé par l'utilisateur également. Le bleu représente le chemin que l'utilisateur a tracé mais qui n'est pas dans la solution proposée.



A noter qu'il n'y a pas unicité de solution à une grille de Surizarinku. Les fonctions "Vérifier la grille" prennent cela en compte et vérifient la justesse d'une grille par rapport à sa cohérence, sans rapport avec la solution proposée, qui n'est qu'une possibilité.

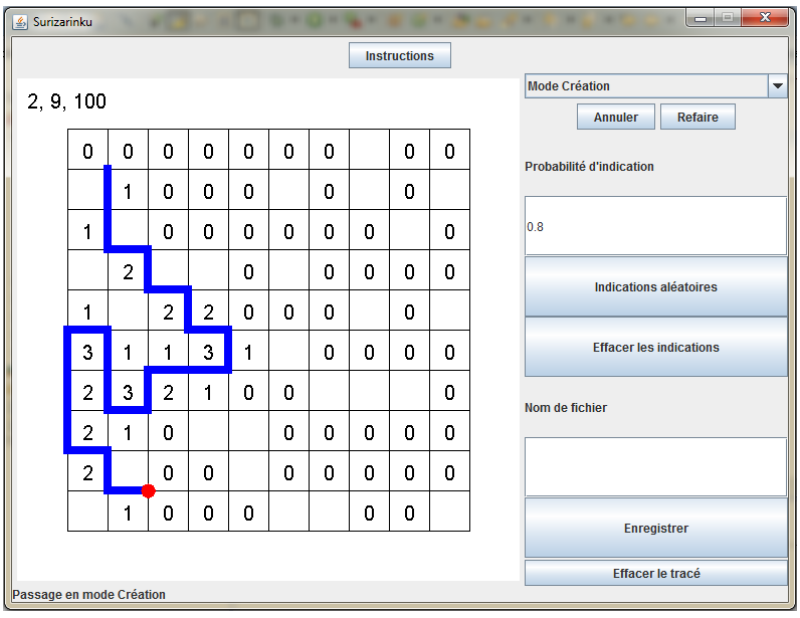
4.2 Mode création



En choisissant "Mode création" dans la liste déroulante en haut à droite, on arrive dans le mode création. Initialement, la grille est vide. En appuyant sur "Indications aléatoires", le programme choisit au hasard les cases qui restreindront l'utilisateur, avec une proportion dans le champ "Probabilité d'indication".

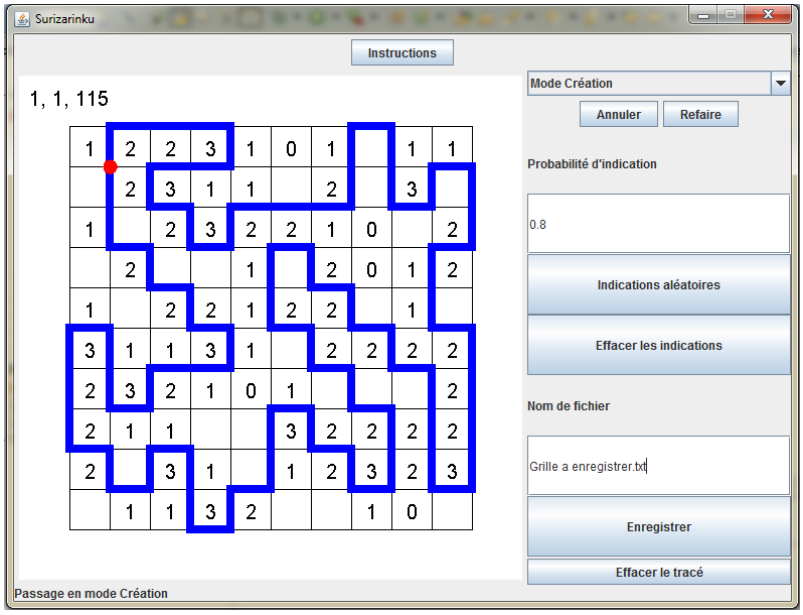
Ensuite, l'utilisateur dessine lui-même son chemin, les indications étant mises à jour automatiquement. Les boutons "Annuler", "Refaire" et "Effacer le tracé" ont le même effet que dans le mode jeu. Les mécanismes de jeu sont identiques (utilisation des touches Z, Q, S, D du clavier et clic de souris sur les noeuds).

Une fonctionnalité est rajoutée : l'utilisateur peut cliquer au centre d'une case pour choisir si le nombre de côtés pleins sera indiqué dans la case en question, ou si cette dernière sera laissée vide.



Lorsque l'utilisateur a fini de dessiner son chemin, il peut sauvegarder la grille. Pour ce faire, il inscrit le nom

sous lequel il souhaite enregistrer la grille (ici Grille à enregistrer.txt, l'extension étant facultative), puis appuie sur "Enregistrer le fichier".



La grille apparaît alors dans la liste dans le menu du mode jeu, et peut être chargée, puis jouée. La solution proposée à l'utilisateur est le chemin tracé dans le mode création.

5 Source du programme

5.1 Classe Action

```
public class Action {
   int x;
   int y;
   Direction d;

   enum Direction {
      H, B, G, D;
   }

   public Action(int coordX, int coordY, Direction directD) {
      x = coordX;
      y = coordY;
      d = directD;
   }

}
```

5.2 Classe Dates

```
public class Dates {
   // Choix de la langue francaise
   static Locale locale = Locale.getDefault();
   static Date actuelle = new Date();

   // Definition du format utilise pour les dates
   static DateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");

   // Donne la date
   public static String date() {
      String dat = dateFormat.format(actuelle);
      return dat;
   }
}
```

5.2.1 Classe Grille

```
public class Grille {

   protected enum Diff {
      F, M, D;
   }

   protected final Diff DIFFICULTE;

   protected final int TAILLEL;
   protected final int TAILLEH;
```

```
protected int[][] nbCotesPleins;
protected boolean[][] coteBEstPlein;
protected boolean[][] coteDEstPlein;
protected boolean[][] cheminEstFaux;
protected boolean[][] celluleEstFausse;

protected final boolean[][] SOLUTION_COTEB_EST_PLEIN;
protected final boolean[][] SOLUTION_COTED_EST_PLEIN;

/*
 * Constructeurs
 */
public Grille(int tL, int tH, int[][] nbCP, boolean[][] cBEP,
      boolean[][] cDEP, boolean[][] sCBEP, boolean[][] sCDEP, Diff d) {
   TAILLEL = tL;
   TAILLEH = tH;
   nbCotesPleins = nbCP;
   coteBEstPlein = cBEP;
   coteDEstPlein = cDEP;
   cheminEstFaux = new boolean[TAILLEL + 1][TAILLEH + 1];
   celluleEstFausse = new boolean[TAILLEL + 1][TAILLEH + 1];
   SOLUTION_COTEB_EST_PLEIN = sCBEP;
   SOLUTION_COTED_EST_PLEIN = sCDEP;
   DIFFICULTE = d;
}

public Grille(int tL, int tH, int[][] nbCP, boolean[][] cBEP,
      boolean[][] cDEP, boolean[][] sCBEP, boolean[][] sCDEP) {
   this(tL, tH, nbCP, cBEP, cDEP, sCBEP, sCDEP, null);
}

public Grille(int tL, int tH, int[][] nbCP) {
   this(tL, tH, nbCP, new boolean[tL + 1][tH + 1],
         new boolean[tL + 1][tH + 1], new boolean[tL + 1][tH + 1],
         new boolean[tL + 1][tH + 1], null);
}

public Grille(int tL, int tH, int[][] nbCP, boolean[][] sCBEP,
      boolean[][] sCDEP) {
   this(tL, tH, nbCP, new boolean[tL + 1][tH + 1],
         new boolean[tL + 1][tH + 1], sCBEP, sCDEP, null);
}

public Grille(int tL, int tH) {
   this(tL, tH, new int[tL + 1][tH + 1], new boolean[tL + 1][tH + 1],
         new boolean[tL + 1][tH + 1], new boolean[tL + 1][tH + 1],
         new boolean[tL + 1][tH + 1], null);
   int[][] grilleEntiers = new int[tL + 1][tH + 1];
```

```
        for (int i = 1; i <= tL; i++) {
            Arrays.fill(grilleEntiers[i], 9);
        }
        nbCotesPleins = grilleEntiers;
    }

    /**
     * Vérifie que le chemin proposé par l'utilisateur est cohérent,
     * c'est-à-dire qu'il n'y a pas de croisement et que c'est bien une boucle.
     * Cela revient en fait à vérifier que pour chaque noeud, il y a exactement
     * zéro ou deux attributs parmi les cotés haut, bas, gauche, droit marqués
     * ``true''. Si une erreur est détectée, on met en évidence le noeud posant
     * problème (via le tableau cheminEstFaux). Si les chemins sont fermés sur
     * eux-mêmes, on vérifie s'il n'y a qu'un seul chemin. Pour cela, on choisit
     * un point de départ puis on suit le chemin qui passe par ce point jusqu'à
     * revenir au point de départ. Par la suite, on vérifie qu'il n'y a aucun
     * autre tracé enregistré.
     */
    protected boolean CheminEstFaux() {
        boolean problemeDetecte = false;
        int compteur = 0;
        // Vérification intérieur de la grille sans côtés droit et bas
        for (int i = 0; i < TAILLEL; i++) {
            for (int j = 0; j < TAILLEH; j++) {
                compteur = 0;
                if (coteBEstPlein[i][j])
                    compteur++;
                if (coteDEstPlein[i][j])
                    compteur++;
                if (coteBEstPlein[i + 1][j])
                    compteur++;
                if (coteDEstPlein[i][j + 1])
                    compteur++;
                if (compteur != 2 && compteur != 0) {
                    cheminEstFaux[i][j] = true;
                    problemeDetecte = true;
                } else {
                    cheminEstFaux[i][j] = false;
                }
            }
        }
        // Vérification bord droit
        for (int j = 0; j < TAILLEH; j++) {
            compteur = 0;
            if (coteBEstPlein[TAILLEL][j])
                compteur++;
            if (coteDEstPlein[TAILLEL][j])
                compteur++;
            if (coteDEstPlein[TAILLEL][j + 1])
```

```
            compteur++;
        if (compteur != 2 && compteur != 0) {
            cheminEstFaux[TAILLEL][j] = true;
            problemeDetecte = true;
        } else {
            cheminEstFaux[TAILLEL][j] = false;
        }
    }
    // Vérification bord bas
    for (int i = 0; i < TAILLEL; i++) {
        compteur = 0;
        if (coteBEstPlein[i][TAILLEH])
            compteur++;
        if (coteDEstPlein[i][TAILLEH])
            compteur++;
        if (coteBEstPlein[i + 1][TAILLEH])
            compteur++;
        if (compteur != 2 && compteur != 0) {
            cheminEstFaux[i][TAILLEH] = true;
            problemeDetecte = true;
        } else {
            cheminEstFaux[i][TAILLEH] = false;
        }
    }
    // Coin inférieur droit
    if (!((coteBEstPlein[TAILLEL][TAILLEH] && coteDEstPlein[TAILLEL][TAILLEH]) || (!
         coteBEstPlein[TAILLEL][TAILLEH] && !coteDEstPlein[TAILLEL][TAILLEH]))) {
        cheminEstFaux[TAILLEL][TAILLEH] = true;
        problemeDetecte = true;
    } else {
        cheminEstFaux[TAILLEL][TAILLEH] = false;
    }

    if (problemeDetecte)
        return true;

    // On va vérifier qu'il n'y a qu'un seul chemin tracé
    boolean[][] premierCheminCoteB = new boolean[TAILLEL + 1][TAILLEH + 1];
    boolean[][] premierCheminCoteD = new boolean[TAILLEL + 1][TAILLEH + 1];
    int pointDeparti = -1;
    int pointDepartj = -1;
    int pointCouranti = -1;
    int pointCourantj = -1;
    Action actionRealisee = null;
    // Etape 1 : repérer un point où commencer
    int i = 0, j = 0;
    for (j = 0; j <= TAILLEH; j++) {
        for (i = 0; i <= TAILLEL; i++) {
            if (coteBEstPlein[i][j] || coteDEstPlein[i][j]) {
```

```
            if (coteBEstPlein[i][j]) {
               pointDeparti = i - 1;
               pointDepartj = j;
               pointCouranti = i;
               pointCourantj = j;
               actionRealisee = new Action(i - 1, j,
                     Action.Direction.D);
               premierCheminCoteB[i][j] = true;
            } else {
               pointDeparti = i;
               pointDepartj = j - 1;
               pointCouranti = i;
               pointCourantj = j;
               actionRealisee = new Action(i, j - 1,
                     Action.Direction.B);
               premierCheminCoteD[i][j] = true;
            }
         }
         if (!(actionRealisee == null))
            break;
      }
      if (!(actionRealisee == null))
         break;
   }

   if (!(pointDeparti == -1)) {
      // Etape 2 : suivre un chemin
      while (pointCouranti != pointDeparti
            || pointCourantj != pointDepartj) {
         switch (actionRealisee.d) {
         case D:
            if (coteDEstPlein[pointCouranti][pointCourantj]) {
               actionRealisee = new Action(pointCouranti,
                     pointCourantj, Action.Direction.H);
               premierCheminCoteD[pointCouranti][pointCourantj] = true;
               pointCourantj = pointCourantj - 1;
            } else if (pointCouranti < TAILLEL
                  && coteBEstPlein[pointCouranti + 1][pointCourantj]) {
               actionRealisee = new Action(pointCouranti,
                     pointCourantj, Action.Direction.D);
               premierCheminCoteB[pointCouranti + 1][pointCourantj] = true;
               pointCouranti = pointCouranti + 1;
            } else if (pointCourantj < TAILLEH
                  && coteDEstPlein[pointCouranti][pointCourantj + 1]) {
               actionRealisee = new Action(pointCouranti,
                     pointCourantj, Action.Direction.B);
               premierCheminCoteD[pointCouranti][pointCourantj + 1] = true;
               pointCourantj = pointCourantj + 1;
            }
```

```
            break;
        case H:
            if (coteDEstPlein[pointCouranti][pointCourantj]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.H);
                premierCheminCoteD[pointCouranti][pointCourantj] = true;
                pointCourantj = pointCourantj - 1;
            } else if (coteBEstPlein[pointCouranti][pointCourantj]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.G);
                premierCheminCoteB[pointCouranti][pointCourantj] = true;
                pointCouranti = pointCouranti - 1;
            } else if (pointCourantj < TAILLEL
                    && coteBEstPlein[pointCouranti + 1][pointCourantj]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.D);
                premierCheminCoteB[pointCouranti + 1][pointCourantj] = true;
                pointCouranti = pointCouranti + 1;
            }
            break;
        case B:
            if (pointCourantj < TAILLEH
                    && coteDEstPlein[pointCouranti][pointCourantj + 1]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.B);
                premierCheminCoteD[pointCouranti][pointCourantj + 1] = true;
                pointCourantj = pointCourantj + 1;
            } else if (coteBEstPlein[pointCouranti][pointCourantj]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.G);
                premierCheminCoteB[pointCouranti][pointCourantj] = true;
                pointCouranti = pointCouranti - 1;
            } else if (pointCouranti < TAILLEL
                    && coteBEstPlein[pointCouranti + 1][pointCourantj]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.D);
                premierCheminCoteB[pointCouranti + 1][pointCourantj] = true;
                pointCouranti = pointCouranti + 1;
            }
            break;
        case G:
            if (coteBEstPlein[pointCouranti][pointCourantj]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.G);
                premierCheminCoteB[pointCouranti][pointCourantj] = true;
                pointCouranti = pointCouranti - 1;
            } else if (coteDEstPlein[pointCouranti][pointCourantj]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.H);
```

```
                premierCheminCoteD[pointCouranti][pointCourantj] = true;
                pointCourantj = pointCourantj - 1;
            } else if (pointCourantj < TAILLEH
                  && coteDEstPlein[pointCouranti][pointCourantj + 1]) {
                actionRealisee = new Action(pointCouranti,
                      pointCourantj, Action.Direction.B);
                premierCheminCoteD[pointCouranti][pointCourantj + 1] = true;
                pointCourantj = pointCourantj + 1;
            }
            break;
         default:
            // Rien
         }
      }

      // Etape 3 : revue des points marqués true
      boolean erreurDetectee = false;
      for (i = 0; i <= TAILLEL; i++) {
         for (j = 0; j <= TAILLEH; j++) {
            if (coteBEstPlein[i][j] != premierCheminCoteB[i][j]) {
               erreurDetectee = true;
            }
            if (coteDEstPlein[i][j] != premierCheminCoteD[i][j]) {
               erreurDetectee = true;
            }
         }
         if (erreurDetectee) {
            JOptionPane.showMessageDialog(null,
                  "Il y a plusieurs boucles dans le chemin !");
            return true;
         }
      }
   }
   return false;
}

/**
 * Vérifie que le chemin proposé par l'utilisateur vérifie bien la
 * contrainte du nombre de côtés adjacents aux cases contenant un chiffre
 * différent de 9. Pour cela, on prend les cellules une par une. Si elle
 * contient un chiffre différent de 9, on compte les côtés pleins (via la
 * méthode RenvoyerCotes. Si une erreur est détectée, on met en évidence la
 * cellule posant problème (via l'attribut celluleEstFausse).
 */
protected boolean CotesSontFaux() {
   boolean erreurDetectee = false;
   for (int i = 1; i <= TAILLEL; i++) {
      for (int j = 1; j <= TAILLEH; j++) {
         if (nbCotesPleins[i][j] != 9) {
```

```
                if (RenvoyerCotes(i, j) != nbCotesPleins[i][j]) {
                    celluleEstFausse[i][j] = true;
                    erreurDetectee = true;
                } else {
                    celluleEstFausse[i][j] = false;
                }
            } else {
                celluleEstFausse[i][j] = false;
            }
        }
    }
    return erreurDetectee;
}

/**
 * Vide les tableaux relatifs aux vérifications des côtés et des chiffres,
 * pour que la grille soit dessinée "normalement".
 */
protected void EffacerVerifier() {
    for (int i = 1; i <= TAILLEL; i++) {
        for (int j = 1; j <= TAILLEH; j++) {
            celluleEstFausse[i][j] = false;
            cheminEstFaux[i][j] = false;
        }
    }
}

/**
 * Change l'état du côté droit de la case concernée i,j
 */
protected void ChangeCoteD(int i, int j) {
    if (coteDEstPlein[i][j]) {
        coteDEstPlein[i][j] = false;
    } else {
        coteDEstPlein[i][j] = true;
    }
}

/**
 * Change l'état du côté bas de la case concernée i,j
 */
protected void ChangeCoteB(int i, int j) {
    if (coteBEstPlein[i][j]) {
        coteBEstPlein[i][j] = false;
    } else {
        coteBEstPlein[i][j] = true;
    }
}
```

```
/**
 * Renvoie le nombre de côtés pleins bordant la case i,j de la grille
 */
protected int RenvoyerCotes(int i, int j) {
   int compteur = 0;
   if (coteBEstPlein[i][j])
      compteur++;
   if (coteDEstPlein[i][j])
      compteur++;
   if (coteBEstPlein[i][j - 1])
      compteur++;
   if (coteDEstPlein[i - 1][j])
      compteur++;
   return compteur;
}

/**
 * Rafraîchit le nombre de côtés pleins de toutes les cases de la grille
 */
protected void RafraichirCotes() {
   int i, j;
   for (i = 1; i <= TAILLEL; i++) {
      for (j = 1; j <= TAILLEH; j++) {
         if (nbCotesPleins[i][j] != 9) {
            nbCotesPleins[i][j] = RenvoyerCotes(i, j);
         }
      }
   }
}

/**
 * Détermine aléatoirement (probabilité d) si la chaque case du tableau
 * donnera une restriction au chemin de l'utilisateur
 */
protected void RemplirNbCotesAlea(double d) {
   int i, j;
   for (i = 1; i <= TAILLEL; i++) {
      for (j = 1; j <= TAILLEH; j++) {
         if (Math.random() < d) {
            nbCotesPleins[i][j] = RenvoyerCotes(i, j);
         } else {
            nbCotesPleins[i][j] = 9;
         }
      }
   }
}

/**
 * Bascule le chemin solution enregistré vers le chemin tracé par
```

```
     * l'utilisateur.
     */
    protected void TracerCheminSolution() {
        int i, j;
        for (i = 0; i <= TAILLEL; i++) {
            for (j = 0; j <= TAILLEH; j++) {
                coteBEstPlein[i][j] = SOLUTION_COTEB_EST_PLEIN[i][j];
                coteDEstPlein[i][j] = SOLUTION_COTED_EST_PLEIN[i][j];
            }
        }
        RafraichirCotes();
    }

    /**
     * Remet à zéro l'état des côtés des cases
     */
    protected void ResetCotesPleins() {
        coteDEstPlein = new boolean[TAILLEL + 1][TAILLEH + 1];
        coteBEstPlein = new boolean[TAILLEL + 1][TAILLEH + 1];
    }

}
```

5.3 Classe Session

```
public class Grille {

    protected enum Diff {
        F, M, D;
    }

    protected final Diff DIFFICULTE;

    protected final int TAILLEL;
    protected final int TAILLEH;

    protected int[][] nbCotesPleins;
    protected boolean[][] coteBEstPlein;
    protected boolean[][] coteDEstPlein;
    protected boolean[][] cheminEstFaux;
    protected boolean[][] celluleEstFausse;

    protected final boolean[][] SOLUTION_COTEB_EST_PLEIN;
    protected final boolean[][] SOLUTION_COTED_EST_PLEIN;

    /*
     * Constructeurs
     */
    public Grille(int tL, int tH, int[][] nbCP, boolean[][] cBEP,
            boolean[][] cDEP, boolean[][] sCBEP, boolean[][] sCDEP, Diff d) {
```

```
        TAILLEL = tL;
        TAILLEH = tH;
        nbCotesPleins = nbCP;
        coteBEstPlein = cBEP;
        coteDEstPlein = cDEP;
        cheminEstFaux = new boolean[TAILLEL + 1][TAILLEH + 1];
        celluleEstFausse = new boolean[TAILLEL + 1][TAILLEH + 1];
        SOLUTION_COTEB_EST_PLEIN = sCBEP;
        SOLUTION_COTED_EST_PLEIN = sCDEP;
        DIFFICULTE = d;
    }

    public Grille(int tL, int tH, int[][] nbCP, boolean[][] cBEP,
            boolean[][] cDEP, boolean[][] sCBEP, boolean[][] sCDEP) {
        this(tL, tH, nbCP, cBEP, cDEP, sCBEP, sCDEP, null);
    }

    public Grille(int tL, int tH, int[][] nbCP) {
        this(tL, tH, nbCP, new boolean[tL + 1][tH + 1],
                new boolean[tL + 1][tH + 1], new boolean[tL + 1][tH + 1],
                new boolean[tL + 1][tH + 1], null);
    }

    public Grille(int tL, int tH, int[][] nbCP, boolean[][] sCBEP,
            boolean[][] sCDEP) {
        this(tL, tH, nbCP, new boolean[tL + 1][tH + 1],
                new boolean[tL + 1][tH + 1], sCBEP, sCDEP, null);
    }

    public Grille(int tL, int tH) {
        this(tL, tH, new int[tL + 1][tH + 1], new boolean[tL + 1][tH + 1],
                new boolean[tL + 1][tH + 1], new boolean[tL + 1][tH + 1],
                new boolean[tL + 1][tH + 1], null);
        int[][] grilleEntiers = new int[tL + 1][tH + 1];
        for (int i = 1; i <= tL; i++) {
            Arrays.fill(grilleEntiers[i], 9);
        }
        nbCotesPleins = grilleEntiers;
    }

    /**
     * Vérifie que le chemin proposé par l'utilisateur est cohérent,
     * c'est-à-dire qu'il n'y a pas de croisement et que c'est bien une boucle.
     * Cela revient en fait à vérifier que pour chaque noeud, il y a exactement
     * zéro ou deux attributs parmi les cotés haut, bas, gauche, droit marqués
     * ``true''. Si une erreur est détectée, on met en évidence le noeud posant
     * problème (via le tableau cheminEstFaux). Si les chemins sont fermés sur
     * eux-mêmes, on vérifie s'il n'y a qu'un seul chemin. Pour cela, on choisit
     * un point de départ puis on suit le chemin qui passe par ce point jusqu'à
```

```
     * revenir au point de départ. Par la suite, on vérifie qu'il n'y a aucun
     * autre tracé enregistré.
     */
    protected boolean CheminEstFaux() {
        boolean problemeDetecte = false;
        int compteur = 0;
        // Vérification intérieur de la grille sans côtés droit et bas
        for (int i = 0; i < TAILLEL; i++) {
            for (int j = 0; j < TAILLEH; j++) {
                compteur = 0;
                if (coteBEstPlein[i][j])
                    compteur++;
                if (coteDEstPlein[i][j])
                    compteur++;
                if (coteBEstPlein[i + 1][j])
                    compteur++;
                if (coteDEstPlein[i][j + 1])
                    compteur++;
                if (compteur != 2 && compteur != 0) {
                    cheminEstFaux[i][j] = true;
                    problemeDetecte = true;
                } else {
                    cheminEstFaux[i][j] = false;
                }
            }
        }
        // Vérification bord droit
        for (int j = 0; j < TAILLEH; j++) {
            compteur = 0;
            if (coteBEstPlein[TAILLEL][j])
                compteur++;
            if (coteDEstPlein[TAILLEL][j])
                compteur++;
            if (coteDEstPlein[TAILLEL][j + 1])
                compteur++;
            if (compteur != 2 && compteur != 0) {
                cheminEstFaux[TAILLEL][j] = true;
                problemeDetecte = true;
            } else {
                cheminEstFaux[TAILLEL][j] = false;
            }
        }
        // Vérification bord bas
        for (int i = 0; i < TAILLEL; i++) {
            compteur = 0;
            if (coteBEstPlein[i][TAILLEH])
                compteur++;
            if (coteDEstPlein[i][TAILLEH])
                compteur++;
```

```
        if (coteBEstPlein[i + 1][TAILLEH])
            compteur++;
        if (compteur != 2 && compteur != 0) {
            cheminEstFaux[i][TAILLEH] = true;
            problemeDetecte = true;
        } else {
            cheminEstFaux[i][TAILLEH] = false;
        }
    }
    // Coin inférieur droit
    if (!((coteBEstPlein[TAILLEL][TAILLEH] && coteDEstPlein[TAILLEL][TAILLEH]) || (!
         coteBEstPlein[TAILLEL][TAILLEH] && !coteDEstPlein[TAILLEL][TAILLEH]))) {
        cheminEstFaux[TAILLEL][TAILLEH] = true;
        problemeDetecte = true;
    } else {
        cheminEstFaux[TAILLEL][TAILLEH] = false;
    }

    if (problemeDetecte)
        return true;

    // On va vérifier qu'il n'y a qu'un seul chemin tracé
    boolean[][] premierCheminCoteB = new boolean[TAILLEL + 1][TAILLEH + 1];
    boolean[][] premierCheminCoteD = new boolean[TAILLEL + 1][TAILLEH + 1];
    int pointDeparti = -1;
    int pointDepartj = -1;
    int pointCouranti = -1;
    int pointCourantj = -1;
    Action actionRealisee = null;
    // Etape 1 : repérer un point où commencer
    int i = 0, j = 0;
    for (j = 0; j <= TAILLEH; j++) {
        for (i = 0; i <= TAILLEL; i++) {
            if (coteBEstPlein[i][j] || coteDEstPlein[i][j]) {
                if (coteBEstPlein[i][j]) {
                    pointDeparti = i - 1;
                    pointDepartj = j;
                    pointCouranti = i;
                    pointCourantj = j;
                    actionRealisee = new Action(i - 1, j,
                            Action.Direction.D);
                    premierCheminCoteB[i][j] = true;
                } else {
                    pointDeparti = i;
                    pointDepartj = j - 1;
                    pointCouranti = i;
                    pointCourantj = j;
                    actionRealisee = new Action(i, j - 1,
                            Action.Direction.B);
```

```
                premierCheminCoteD[i][j] = true;
            }
        }
        if (!(actionRealisee == null))
            break;
    }
    if (!(actionRealisee == null))
        break;
}

if (!(pointDeparti == -1)) {
    // Etape 2 : suivre un chemin
    while (pointCouranti != pointDeparti
            || pointCourantj != pointDepartj) {
        switch (actionRealisee.d) {
        case D:
            if (coteDEstPlein[pointCouranti][pointCourantj]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.H);
                premierCheminCoteD[pointCouranti][pointCourantj] = true;
                pointCourantj = pointCourantj - 1;
            } else if (pointCouranti < TAILLEL
                    && coteBEstPlein[pointCouranti + 1][pointCourantj]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.D);
                premierCheminCoteB[pointCouranti + 1][pointCourantj] = true;
                pointCouranti = pointCouranti + 1;
            } else if (pointCourantj < TAILLEH
                    && coteDEstPlein[pointCouranti][pointCourantj + 1]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.B);
                premierCheminCoteD[pointCouranti][pointCourantj + 1] = true;
                pointCourantj = pointCourantj + 1;
            }
            break;
        case H:
            if (coteDEstPlein[pointCouranti][pointCourantj]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.H);
                premierCheminCoteD[pointCouranti][pointCourantj] = true;
                pointCourantj = pointCourantj - 1;
            } else if (coteBEstPlein[pointCouranti][pointCourantj]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.G);
                premierCheminCoteB[pointCouranti][pointCourantj] = true;
                pointCouranti = pointCouranti - 1;
            } else if (pointCourantj < TAILLEL
                    && coteBEstPlein[pointCouranti + 1][pointCourantj]) {
                actionRealisee = new Action(pointCouranti,
```

```
                    pointCourantj, Action.Direction.D);
                premierCheminCoteB[pointCouranti + 1][pointCourantj] = true;
                pointCouranti = pointCouranti + 1;
            }
            break;
        case B:
            if (pointCourantj < TAILLEH
                    && coteDEstPlein[pointCouranti][pointCourantj + 1]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.B);
                premierCheminCoteD[pointCouranti][pointCourantj + 1] = true;
                pointCourantj = pointCourantj + 1;
            } else if (coteBEstPlein[pointCouranti][pointCourantj]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.G);
                premierCheminCoteB[pointCouranti][pointCourantj] = true;
                pointCouranti = pointCouranti - 1;
            } else if (pointCouranti < TAILLEL
                    && coteBEstPlein[pointCouranti + 1][pointCourantj]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.D);
                premierCheminCoteB[pointCouranti + 1][pointCourantj] = true;
                pointCouranti = pointCouranti + 1;
            }
            break;
        case G:
            if (coteBEstPlein[pointCouranti][pointCourantj]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.G);
                premierCheminCoteB[pointCouranti][pointCourantj] = true;
                pointCouranti = pointCouranti - 1;
            } else if (coteDEstPlein[pointCouranti][pointCourantj]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.H);
                premierCheminCoteD[pointCouranti][pointCourantj] = true;
                pointCourantj = pointCourantj - 1;
            } else if (pointCourantj < TAILLEH
                    && coteDEstPlein[pointCouranti][pointCourantj + 1]) {
                actionRealisee = new Action(pointCouranti,
                        pointCourantj, Action.Direction.B);
                premierCheminCoteD[pointCouranti][pointCourantj + 1] = true;
                pointCourantj = pointCourantj + 1;
            }
            break;
        default:
            // Rien
        }
    }
```

```
        // Etape 3 : revue des points marqués true
        boolean erreurDetectee = false;
        for (i = 0; i <= TAILLEL; i++) {
           for (j = 0; j <= TAILLEH; j++) {
              if (coteBEstPlein[i][j] != premierCheminCoteB[i][j]) {
                 erreurDetectee = true;
              }
              if (coteDEstPlein[i][j] != premierCheminCoteD[i][j]) {
                 erreurDetectee = true;
              }
           }
           if (erreurDetectee) {
              JOptionPane.showMessageDialog(null,
                    "Il y a plusieurs boucles dans le chemin !");
              return true;
           }
        }
     }
     return false;
  }

  /**
   * Vérifie que le chemin proposé par l'utilisateur vérifie bien la
   * contrainte du nombre de côtés adjacents aux cases contenant un chiffre
   * différent de 9. Pour cela, on prend les cellules une par une. Si elle
   * contient un chiffre différent de 9, on compte les côtés pleins (via la
   * méthode RenvoyerCotes. Si une erreur est détectée, on met en évidence la
   * cellule posant problème (via l'attribut celluleEstFausse).
   */
  protected boolean CotesSontFaux() {
     boolean erreurDetectee = false;
     for (int i = 1; i <= TAILLEL; i++) {
        for (int j = 1; j <= TAILLEH; j++) {
           if (nbCotesPleins[i][j] != 9) {
              if (RenvoyerCotes(i, j) != nbCotesPleins[i][j]) {
                 celluleEstFausse[i][j] = true;
                 erreurDetectee = true;
              } else {
                 celluleEstFausse[i][j] = false;
              }
           } else {
              celluleEstFausse[i][j] = false;
           }
        }
     }
     return erreurDetectee;
  }

  /**
```

```
     * Vide les tableaux relatifs aux vérifications des côtés et des chiffres,
     * pour que la grille soit dessinée "normalement".
     */
    protected void EffacerVerifier() {
        for (int i = 1; i <= TAILLEL; i++) {
            for (int j = 1; j <= TAILLEH; j++) {
                celluleEstFausse[i][j] = false;
                cheminEstFaux[i][j] = false;
            }
        }
    }

    /**
     * Change l'état du côté droit de la case concernée i,j
     */
    protected void ChangeCoteD(int i, int j) {
        if (coteDEstPlein[i][j]) {
            coteDEstPlein[i][j] = false;
        } else {
            coteDEstPlein[i][j] = true;
        }
    }

    /**
     * Change l'état du côté bas de la case concernée i,j
     */
    protected void ChangeCoteB(int i, int j) {
        if (coteBEstPlein[i][j]) {
            coteBEstPlein[i][j] = false;
        } else {
            coteBEstPlein[i][j] = true;
        }
    }

    /**
     * Renvoie le nombre de côtés pleins bordant la case i,j de la grille
     */
    protected int RenvoyerCotes(int i, int j) {
        int compteur = 0;
        if (coteBEstPlein[i][j])
            compteur++;
        if (coteDEstPlein[i][j])
            compteur++;
        if (coteBEstPlein[i][j - 1])
            compteur++;
        if (coteDEstPlein[i - 1][j])
            compteur++;
        return compteur;
    }
```

```
/**
 * Rafraîchit le nombre de côtés pleins de toutes les cases de la grille
 */
protected void RafraichirCotes() {
   int i, j;
   for (i = 1; i <= TAILLEL; i++) {
      for (j = 1; j <= TAILLEH; j++) {
         if (nbCotesPleins[i][j] != 9) {
            nbCotesPleins[i][j] = RenvoyerCotes(i, j);
         }
      }
   }
}

/**
 * Détermine aléatoirement (probabilité d) si la chaque case du tableau
 * donnera une restriction au chemin de l'utilisateur
 */
protected void RemplirNbCotesAlea(double d) {
   int i, j;
   for (i = 1; i <= TAILLEL; i++) {
      for (j = 1; j <= TAILLEH; j++) {
         if (Math.random() < d) {
            nbCotesPleins[i][j] = RenvoyerCotes(i, j);
         } else {
            nbCotesPleins[i][j] = 9;
         }
      }
   }
}

/**
 * Bascule le chemin solution enregistré vers le chemin tracé par
 * l'utilisateur.
 */
protected void TracerCheminSolution() {
   int i, j;
   for (i = 0; i <= TAILLEL; i++) {
      for (j = 0; j <= TAILLEH; j++) {
         coteBEstPlein[i][j] = SOLUTION_COTEB_EST_PLEIN[i][j];
         coteDEstPlein[i][j] = SOLUTION_COTED_EST_PLEIN[i][j];
      }
   }
   RafraichirCotes();
}

/**
 * Remet à zéro l'état des côtés des cases
```

```
     */
    protected void ResetCotesPleins() {
        coteDEstPlein = new boolean[TAILLEL + 1][TAILLEH + 1];
        coteBEstPlein = new boolean[TAILLEL + 1][TAILLEH + 1];
    }

}
```

5.4 Classe Session_jeu

```
public class Session_jeu extends Session {

    public Session_jeu(Grille j) {
        super(j);
        addMouseListener(new EcouteurSouris());
        addKeyListener(new EcouteurClavier());
    }

    public Session_jeu(int lar, int hau) {
        super(lar, hau);
        addMouseListener(new EcouteurSouris());
        addKeyListener(new EcouteurClavier());
    }

    private class EcouteurSouris extends MouseAdapter {
        public void mouseClicked(MouseEvent e) {
            int coordSourisX = e.getX() - MARGE;
            int coordSourisY = e.getY() - MARGE;
            int coordX = 999;
            int coordY = 999;
            if ((coordSourisX) % 40 < 10)
                coordX = coordSourisX / 40;
            if ((coordSourisX / 40 + 1) * 40 - coordSourisX < 10)
                coordX = coordSourisX / 40 + 1;
            if ((coordSourisY) % 40 < 10)
                coordY = (coordSourisY - coordSourisY % 40) / 40;
            if ((coordSourisY / 40 + 1) * 40 - coordSourisY < 10)
                coordY = coordSourisY / 40 + 1;
            if (0 <= coordX && coordX <= RenvoyerLargeur() && 0 <= coordY
                    && coordY <= RenvoyerHauteur()) {
                pointActifX = coordX;
                pointActifY = coordY;
            }
            coordonnees = "" + pointActifX + ", " + pointActifY + ", "
                    + keyEventNum;
            repaint();
            grabFocus();
        }
    }
```

```
    private class EcouteurClavier extends KeyAdapter {
        public void keyTyped(KeyEvent e) {
            keyEventNum = e.getKeyChar();

            switch (keyEventNum) {
            case 122:
                if (pointActifY > 0) {
                    pileActEffectuees.push(new Action(pointActifX, pointActifY,
                            Action.Direction.H));
                    grille.ChangeCoteD(pointActifX, pointActifY);
                    pointActifY--;
                }
                break;
            case 115:
                if (pointActifY < RenvoyerHauteur()) {
                    pileActEffectuees.push(new Action(pointActifX, pointActifY,
                            Action.Direction.B));
                    pointActifY++;
                    grille.ChangeCoteD(pointActifX, pointActifY);
                }
                break;
            case 113:
                if (pointActifX > 0) {
                    pileActEffectuees.push(new Action(pointActifX, pointActifY,
                            Action.Direction.G));
                    grille.ChangeCoteB(pointActifX, pointActifY);
                    pointActifX--;
                }
                break;
            case 100:
                if (pointActifX < RenvoyerLargeur()) {
                    pileActEffectuees.push(new Action(pointActifX, pointActifY,
                            Action.Direction.D));
                    pointActifX++;
                    grille.ChangeCoteB(pointActifX, pointActifY);
                }
                break;
            }
            coordonnees = "" + pointActifX + ", " + pointActifY + ", "
                    + keyEventNum;
            repaint();
        }
    }
}
```

5.5 Classe Session_creation

```
public class Session_creation extends Session {

    public Session_creation(Grille j) {
```

```
        super(j);
        addMouseListener(new EcouteurSouris());
        addKeyListener(new EcouteurClavier());
    }

    public Session_creation(int lar, int hau) {
        super(lar, hau);
        addMouseListener(new EcouteurSouris());
        addKeyListener(new EcouteurClavier());
    }

    public void paint(Graphics g) {
        grille.RafraichirCotes();
        super.paint(g);
    }

    public void ResetNbCotesPleins() {
        int[][] grilleEntiers = new int[RenvoyerLargeur() + 1][RenvoyerHauteur() + 1];
        for (int i = 1; i <= RenvoyerLargeur(); i++) {
            Arrays.fill(grilleEntiers[i], 9);
        }
        grille.nbCotesPleins = grilleEntiers;
        repaint();
    }

    private class EcouteurSouris extends MouseAdapter {
        public void mouseClicked(MouseEvent e) {
            int coordSourisX = e.getX() - MARGE;
            int coordSourisY = e.getY() - MARGE;
            int coordX = 999;
            int coordY = 999;
            if ((coordSourisX) % 40 < 10)
                coordX = coordSourisX / 40;
            if ((coordSourisX / 40 + 1) * 40 - coordSourisX < 10)
                coordX = coordSourisX / 40 + 1;
            if ((coordSourisY) % 40 < 10)
                coordY = (coordSourisY - coordSourisY % 40) / 40;
            if ((coordSourisY / 40 + 1) * 40 - coordSourisY < 10)
                coordY = coordSourisY / 40 + 1;
            // Ajouter ici le cas "je clique sur le centre d'une case" et je
            // modifie ou non...
            if ((coordSourisX) % 40 > 10 && 40 - (coordSourisX) % 40 > 10
                    && (coordSourisY) % 40 > 10
                    && 40 - (coordSourisY) % 40 > 10) {
                if (grille.nbCotesPleins[coordSourisX / 40 + 1][coordSourisY / 40 + 1] == 9)
                    {
                    grille.nbCotesPleins[coordSourisX / 40 + 1][coordSourisY / 40 + 1] =
                        grille
                            .RenvoyerCotes(coordSourisX / 40 + 1,
```

```
                        coordSourisY / 40 + 1);
            } else {
                grille.nbCotesPleins[coordSourisX / 40 + 1][coordSourisY / 40 + 1] = 9;
            }
        }

        if (0 <= coordX && coordX <= RenvoyerLargeur() && 0 <= coordY
                && coordY <= RenvoyerHauteur()) {
            pointActifX = coordX;
            pointActifY = coordY;
        }
        coordonnees = "" + pointActifX + ", " + pointActifY + ", "
                + keyEventNum;
        repaint();
        grabFocus();
    }
}

private class EcouteurClavier extends KeyAdapter {
    public void keyTyped(KeyEvent e) {
        keyEventNum = e.getKeyChar();

        switch (keyEventNum) {
        case 122:
            if (pointActifY > 0) {
                pileActEffectuees.push(new Action(pointActifX, pointActifY,
                        Action.Direction.H));
                grille.ChangeCoteD(pointActifX, pointActifY);
                pointActifY--;
            }
            break;
        case 115:
            if (pointActifY < RenvoyerHauteur()) {
                pileActEffectuees.push(new Action(pointActifX, pointActifY,
                        Action.Direction.B));
                pointActifY++;
                grille.ChangeCoteD(pointActifX, pointActifY);
            }
            break;
        case 113:
            if (pointActifX > 0) {
                pileActEffectuees.push(new Action(pointActifX, pointActifY,
                        Action.Direction.G));
                grille.ChangeCoteB(pointActifX, pointActifY);
                pointActifX--;
            }
            break;
        case 100:
            if (pointActifX < RenvoyerLargeur()) {
```

```
                pileActEffectuees.push(new Action(pointActifX, pointActifY,
                        Action.Direction.D));
                pointActifX++;
                grille.ChangeCoteB(pointActifX, pointActifY);
            }
            break;
        }

        coordonnees = "" + pointActifX + ", " + pointActifY + ", "
                + keyEventNum;
        repaint();
    }
  }
}
```

5.6 Classe Surizarinku

```
public class Surizarinku extends JFrame implements ActionListener, ItemListener {

  private Session_jeu sessionJeu;
  private Session_creation sessionCreation;

  private enum TypeActif {
    J, C;
  }

  private TypeActif sessionActive;

  private JPanel panelPrincipal = new JPanel();
  private JPanel panelDroit = new JPanel(new BorderLayout());
  private JPanel panelHaut = new JPanel(new BorderLayout());
  private JPanel panelAR = new JPanel(new FlowLayout());
  private JPanel panelLireSupprimer = new JPanel(new GridLayout(3, 1));
  private JPanel panelInstructions = new JPanel(new FlowLayout());
  private JPanel panelBoutonsVerifierOk = new JPanel(new FlowLayout());
  private JPanel panelCards = new JPanel(new CardLayout());
  private JPanel panelBoutonsJeu = new JPanel(new BorderLayout());
  private JPanel panelBoutonsCreation = new JPanel(new GridLayout(7, 1));

  private String[] comboBoxItems = { "Mode Jeu", "Mode Création" };
  private JComboBoxItemListener comboBoxMode = new JComboBoxItemListener(
      comboBoxItems);

  private JButton boutonReset = new JButton("Effacer le tracé");
  private JButton boutonResetNbCotesPleins = new JButton(
      "Effacer les indications");
  private JButton boutonAnnuler = new JButton("Annuler");
  private JButton boutonRefaire = new JButton("Refaire");
  private JButton boutonLireFichier = new JButton("Lire la grille");
  private JButton boutonModifierFichier = new JButton("Modifier la grille");
```

```
private JButton boutonSupprimerFichier = new JButton("Supprimer la grille");
private JButton boutonRemplirAlea = new JButton("Indications aléatoires");
private JButton boutonEnregistrer = new JButton("Enregistrer");
private JButton boutonVerifierGrille = new JButton("Vérifier la grille");
private JButton boutonToggleSolution = new JButton("Montrer une solution");
private JButton boutonInstructions = new JButton("Instructions");

private List listeGrilles = new List();

private JTextField champTexte = new JTextField();
private JTextField champProba = new JTextField("0.8");

private JLabel labelInformations = new JLabel(
      "Bienvenue dans le module Surizarinku !");
private JLabel labelProba = new JLabel("Probabilité d'indication");
private JLabel labelTexte = new JLabel("Nom de fichier");

public Surizarinku(Session_jeu sj, Session_creation sc) {
   setTitle("Surizarinku");
   setLayout(new BorderLayout());

   panelDroit.setSize(new Dimension(200, 550));

   sessionJeu = sj;
   sessionCreation = sc;
   sessionActive = TypeActif.J;

   comboBoxMode.addActionListener(this);

   LireListeGrilles();
   listeGrilles.addActionListener(this);

   boutonReset.addActionListener(this);
   boutonAnnuler.addActionListener(this);
   boutonRefaire.addActionListener(this);
   boutonLireFichier.addActionListener(this);
   boutonModifierFichier.addActionListener(this);
   boutonSupprimerFichier.addActionListener(this);
   boutonRemplirAlea.addActionListener(this);
   boutonEnregistrer.addActionListener(this);
   boutonVerifierGrille.addActionListener(this);
   boutonToggleSolution.addActionListener(this);
   boutonResetNbCotesPleins.addActionListener(this);
   boutonInstructions.addActionListener(this);

   listeGrilles.addItemListener(this);

   sessionJeu.setLayout(new BoxLayout(sessionJeu, BoxLayout.Y_AXIS));
   sessionCreation.setLayout(new BoxLayout(sessionCreation,
```

```
                BoxLayout.Y_AXIS));
        sessionJeu.setPreferredSize(new Dimension(500, 500));
        sessionCreation.setPreferredSize(new Dimension(500, 500));
        sessionJeu.setFocusable(true);
        sessionCreation.setFocusable(true);

        panelPrincipal.add(sessionJeu);

        panelHaut.add(comboBoxMode, BorderLayout.NORTH);
        panelHaut.add(panelAR, BorderLayout.SOUTH);

        panelAR.add(boutonAnnuler);
        panelAR.add(boutonRefaire);

        panelLireSupprimer.add(boutonLireFichier);
        panelLireSupprimer.add(boutonModifierFichier);
        panelLireSupprimer.add(boutonSupprimerFichier);

        panelBoutonsVerifierOk.add(boutonVerifierGrille);

        panelBoutonsJeu.add(boutonToggleSolution, BorderLayout.NORTH);
        panelBoutonsJeu.add(listeGrilles, BorderLayout.WEST);
        panelBoutonsJeu.add(panelLireSupprimer, BorderLayout.EAST);
        panelBoutonsJeu.add(panelBoutonsVerifierOk, BorderLayout.SOUTH);

        panelBoutonsCreation.add(labelProba);
        panelBoutonsCreation.add(champProba);
        panelBoutonsCreation.add(boutonRemplirAlea);
        panelBoutonsCreation.add(boutonResetNbCotesPleins);
        panelBoutonsCreation.add(labelTexte);
        panelBoutonsCreation.add(champTexte);
        panelBoutonsCreation.add(boutonEnregistrer);

        panelCards.add(panelBoutonsJeu, comboBoxItems[0]);
        panelCards.add(panelBoutonsCreation, comboBoxItems[1]);

        panelDroit.add(panelHaut, BorderLayout.NORTH);
        panelDroit.add(panelCards, BorderLayout.CENTER);
        panelDroit.add(boutonReset, BorderLayout.SOUTH);

        panelInstructions.add(boutonInstructions);
        add(labelInformations, BorderLayout.SOUTH);

        add(panelPrincipal, BorderLayout.CENTER);
        add(panelDroit, BorderLayout.EAST);
        add(panelInstructions, BorderLayout.NORTH);
        addWindowListener(new EcouteurFenetre());
    }
```

```
/**
 * Met à jour la liste des grilles depuis le fichier ad-hoc
 */
private void LireListeGrilles() {
   try {
      listeGrilles.removeAll();
      FileReader in = new FileReader("LISTEGRILLES.txt");
      Scanner sc = new Scanner(in);
      while (sc.hasNextLine()) {
         listeGrilles.add(sc.nextLine());
      }
      sc.close();
   } catch (FileNotFoundException e) {
      System.err.println("Fichier non trouvé");
   }
}

@SuppressWarnings("resource")
/**
 * Renvoie une grille lue depuis le fichier correspondant
 */
private Grille LireGrille(String s) {
   String l;
   Scanner sc2;
   char c;
   int TAILLEL = 0;
   int TAILLEH = 0;
   int[][] nbCotesPleins = new int[1][1];
   boolean[][] solutionCoteBEstPlein = new boolean[1][1];
   boolean[][] solutionCoteDEstPlein = new boolean[1][1];
   int ligneEcoutee;
   int colonneEcoutee;

   try {
      FileReader in = new FileReader(s);
      Scanner sc = new Scanner(in);

      // Lecture de la taille du fichier. On vérifie la présence du
      // caractère 'T' pour le formatage du fichier
      while (sc.hasNextLine() && TAILLEL == 0) {
         l = sc.nextLine();
         if (l != "") {
            sc2 = new Scanner(l);
            if (sc2.hasNext()) {
               c = l.charAt(0);
               switch (c) {
               case '/':
                  continue;
               case 'T':
```

```
                sc2.next();
                if (sc2.hasNextInt()) {
                    TAILLEL = sc2.nextInt();
                    if (sc2.hasNextInt()) {
                        TAILLEH = sc2.nextInt();
                        nbCotesPleins = new int[TAILLEL + 1][TAILLEH + 1];
                        solutionCoteBEstPlein = new boolean[TAILLEL + 1][TAILLEH + 1];
                        solutionCoteDEstPlein = new boolean[TAILLEL + 1][TAILLEH + 1];
                    }
                }
                break;
            default:
                continue;
            }
        }
        sc2.close();
    }
}

// Si le formatage n'était pas bon, on arrête
if (TAILLEL == 0)
    throw (new IOException());

// Lecture de la grille nombre côtés pleins
ligneEcoutee = 1;
while (sc.hasNextLine() && ligneEcoutee <= TAILLEH) {
    l = sc.nextLine();
    sc2 = new Scanner(l);
    if (!sc2.hasNextInt()) {
        continue;
    } else {
        colonneEcoutee = 1;
        while (sc2.hasNextInt() && colonneEcoutee <= TAILLEL) {
            nbCotesPleins[colonneEcoutee][ligneEcoutee] = sc2
                    .nextInt();
            colonneEcoutee++;
        }
        colonneEcoutee = 1;
        ligneEcoutee++;
    }
}

// Lecture de la grille côtés bas
ligneEcoutee = 0;
while (sc.hasNextLine() && ligneEcoutee <= TAILLEH) {
    l = sc.nextLine();
    sc2 = new Scanner(l);
    if (!sc2.hasNextInt()) {
        continue;
```

```
            } else {
                colonneEcoutee = 0;
                while (sc2.hasNextInt() && colonneEcoutee <= TAILLEL) {
                    if (sc2.nextInt() == 1) {
                        solutionCoteBEstPlein[colonneEcoutee][ligneEcoutee] = true;
                    } else {
                        solutionCoteBEstPlein[colonneEcoutee][ligneEcoutee] = false;
                    }
                    colonneEcoutee++;
                }
                ligneEcoutee++;
            }
        }

        // Lecture de la grille côtés droits
        ligneEcoutee = 0;
        while (sc.hasNextLine() && ligneEcoutee <= TAILLEH) {
            l = sc.nextLine();
            sc2 = new Scanner(l);
            if (!sc2.hasNextInt()) {
                continue;
            } else {
                colonneEcoutee = 0;
                while (sc2.hasNextInt() && colonneEcoutee <= TAILLEL) {
                    if (sc2.nextInt() == 1) {
                        solutionCoteDEstPlein[colonneEcoutee][ligneEcoutee] = true;
                    } else {
                        solutionCoteDEstPlein[colonneEcoutee][ligneEcoutee] = false;
                    }
                    colonneEcoutee++;
                }
                ligneEcoutee++;
            }
        }

        sc.close();
        return (new Grille(TAILLEL, TAILLEH, nbCotesPleins,
                solutionCoteBEstPlein, solutionCoteDEstPlein));

    } catch (FileNotFoundException e) {
        System.err.println("Fichier non trouvé");
    } catch (IOException e) {
        System.err.println("Erreur d'IO");
    }
    return null;
}

/**
 * Enregistre la grille de la Session_creation dans un fichier portant le
```

```
     * nom dans le champ texte.
     */
    private void EnregistrerGrille(String s) {
        try {
            // On supprime la ligne correspondante dans la liste si elle existe
            // déjà
            SupprimerGrille(s);

            int i, j;
            String ligne;
            PrintWriter out;

            // En-tête du fichier
            out = new PrintWriter(s);
            out.println("// ------------------------------------------------------");
            out.println("// Fichier d'enregistrement de grille de Surizarinku");
            out.println("// Nom : " + s);
            out.println("// Date : " + Dates.date());
            out.println("// ------------------------------------------------------");
            out.println("// Taille :");
            out.println("T " + sessionCreation.RenvoyerLargeur() + " "
                    + sessionCreation.RenvoyerHauteur());

            // Grille nombre côtés pleins
            out.println("// Grille");
            for (j = 1; j <= sessionCreation.RenvoyerHauteur(); j++) {
                ligne = "";
                for (i = 1; i <= sessionCreation.RenvoyerLargeur(); i++) {
                    ligne = ligne + sessionCreation.grille.nbCotesPleins[i][j]
                            + " ";
                }
                out.println(ligne);
            }

            // Grille solution côté bas
            out.println("// Solution CoteBas");
            for (j = 0; j <= sessionCreation.RenvoyerHauteur(); j++) {
                ligne = "";
                for (i = 0; i <= sessionCreation.RenvoyerLargeur(); i++) {
                    if (sessionCreation.grille.coteBEstPlein[i][j]) {
                        ligne = ligne + 1 + " ";
                    } else {
                        ligne = ligne + 0 + " ";
                    }
                }
                out.println(ligne);
            }

            // Grille solution côté droit
```

```
            out.println("// Solution CoteDroit");
            for (j = 0; j <= sessionCreation.RenvoyerHauteur(); j++) {
               ligne = "";
               for (i = 0; i <= sessionCreation.RenvoyerLargeur(); i++) {
                  if (sessionCreation.grille.coteDEstPlein[i][j]) {
                     ligne = ligne + 1 + " ";
                  } else {
                     ligne = ligne + 0 + " ";
                  }
               }
               out.println(ligne);
            }

            out.close();

            // Mise à jour du fichier liste des grilles
            out = new PrintWriter(
                  new FileOutputStream("LISTEGRILLES.txt", true));
            out.println(s);
            out.close();

         } catch (IOException e) {
            System.err.println("Erreur d'IO");
         }
      }

      /**
       * Supprime la ligne s dans le fichier de la liste des grilles
       */
      private void SupprimerGrille(String s) {
         try {
            Vector<String> monVector = new Vector<String>();
            FileReader in = new FileReader("LISTEGRILLES.txt");
            Scanner sc = new Scanner(in);
            String ligne;
            while (sc.hasNextLine()) {
               ligne = sc.nextLine();
               if (!s.equals(ligne)) {
                  monVector.addElement(ligne);
               }
            }
            sc.close();
            in.close();
            PrintWriter out = new PrintWriter("LISTEGRILLES.txt");
            for (int i = 0; i < monVector.size(); i++) {
               out.println(monVector.get(i));
            }
            out.close();
         } catch (FileNotFoundException e) {
```

```
            System.err.println("Fichier non trouvé");
        } catch (IOException e) {
            System.err.println("Erreur d'IO");
        }
    }

    /**
     * Comportement actionPerformed
     */
    public void actionPerformed(ActionEvent evt) {

        if (evt.getSource() == boutonReset) {
            switch (sessionActive) {
            case J:
                sessionJeu.Reset();
                break;
            case C:
                sessionCreation.Reset();
                break;
            }
            labelInformations.setText("Chemin effacé");
        }

        if (evt.getSource() == boutonAnnuler) {
            switch (sessionActive) {
            case J:
                sessionJeu.AnnulerAction();
                break;
            case C:
                sessionCreation.AnnulerAction();
                break;
            }
        }

        if (evt.getSource() == boutonRefaire) {
            switch (sessionActive) {
            case J:
                sessionJeu.RefaireAction();
                break;
            case C:
                sessionCreation.RefaireAction();
                break;
            }
        }

        if (evt.getSource() == boutonLireFichier) {
            try {
                sessionJeu.grille = LireGrille(listeGrilles.getSelectedItem());
                labelInformations.setText("Grille lue : "
```

```
                + listeGrilles.getSelectedItem());
        } catch (NullPointerException e) {
            System.err.println("Grille non conforme");
        }
    }

    if (evt.getSource() == boutonModifierFichier) {
        champTexte.setText(listeGrilles.getSelectedItem());
        sessionCreation.grille = LireGrille(listeGrilles.getSelectedItem());
        sessionCreation.grille.TracerCheminSolution();
        comboBoxMode.setSelectedItem(comboBoxItems[1]);
        labelInformations.setText("Grille lue : "
                + listeGrilles.getSelectedItem());
    }

    if (evt.getSource() == boutonSupprimerFichier) {
        try {
            SupprimerGrille(listeGrilles.getSelectedItem());
            // Mise à jour de la liste des grilles
            listeGrilles.removeAll();
            LireListeGrilles();
            sessionJeu.grille = new Grille(10, 10);
            labelInformations.setText("La grille a bien été supprimée.");
        } catch (NullPointerException e) {
            System.err.println("Aucun item sélectionné");
        }
    }

    if (evt.getSource() == boutonEnregistrer) {
        EnregistrerGrille(champTexte.getText());
        // Mise à jour de la liste des grilles
        listeGrilles.removeAll();
        LireListeGrilles();
        labelInformations.setText("Grille enregistrée sous le nom : "
                + champTexte.getText() + ".");
    }

    if (evt.getSource() == boutonVerifierGrille) {
        if (boutonVerifierGrille.getText().equals("Vérifier la grille")) {
            sessionJeu.grille.CotesSontFaux();
            sessionJeu.grille.CheminEstFaux();
            if (!sessionJeu.grille.CotesSontFaux() && !sessionJeu.grille.CheminEstFaux())
                {
                JOptionPane.showMessageDialog(null, "La grille est juste, félicitations !
                    :)");
            } else {
                boutonVerifierGrille.setText("J'ai compris !");
            }
        } else if (boutonVerifierGrille.getText().equals("J'ai compris !")) {
```

```
            sessionJeu.grille.EffacerVerifier();
            boutonVerifierGrille.setText("Vérifier la grille");
        }

    }

    if (evt.getSource() == boutonToggleSolution) {
        sessionJeu.ToggleSolutionEstActive();
        if (boutonToggleSolution.getText().equals("Montrer une solution")) {
            boutonToggleSolution.setText("Masquer la solution");
        } else {
            boutonToggleSolution.setText("Montrer une solution");
        }
    }

    if (evt.getSource() == boutonRemplirAlea) {
        try {
            sessionCreation.grille.RemplirNbCotesAlea(Double
                    .parseDouble(champProba.getText()));
        } catch (NumberFormatException e) {
            System.err
                    .println("Entrer un double entre 0 et 1 avec un point");
        }
    }

    if (evt.getSource() == boutonResetNbCotesPleins) {
        sessionCreation.ResetNbCotesPleins();
    }

    if (evt.getSource() == boutonInstructions) {
        JOptionPane.showMessageDialog(null, "", "Instructions",
                JOptionPane.INFORMATION_MESSAGE, new ImageIcon(
                        "Instructions.jpg"));
    }

    sessionJeu.repaint();
    sessionCreation.repaint();

    // Déplacement du focus dans la grille
    switch (sessionActive) {
    case J:
        sessionJeu.grabFocus();
        break;
    case C:
        sessionCreation.grabFocus();
        break;
    }
}
```

```
    public void itemStateChanged(ItemEvent evt) {
        switch (sessionActive) {
        case J:
            sessionJeu.grabFocus();
            break;
        case C:
            sessionCreation.grabFocus();
            break;
        }
    }

    /**
     * Bascule vers la session de jeu
     */
    private void AfficheSessionJeu() {
        // Afficher session jeu
        sessionActive = TypeActif.J;
        panelPrincipal.remove(sessionCreation);
        panelPrincipal.add(sessionJeu);
        CardLayout cl = (CardLayout) (panelCards.getLayout());
        cl.show(panelCards, comboBoxItems[0]);
        pack();
        labelInformations.setText("Passage en mode Jeu");
    }

    /**
     * Bascule vers la session création
     */
    private void AfficheSessionCreation() {
        // Afficher session création
        sessionActive = TypeActif.C;
        panelPrincipal.remove(sessionJeu);
        panelPrincipal.add(sessionCreation);
        CardLayout cl = (CardLayout) (panelCards.getLayout());
        cl.show(panelCards, comboBoxItems[1]);
        pack();
        labelInformations.setText("Passage en mode Création");
    }

    /*
     * Classes auxiliaires
     */

    /**
     * Ecouteur de la fenêtre
     *
     * @author Quentin
     *
     */
```

```
    private class EcouteurFenetre extends WindowAdapter {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    }


    private class JComboBoxItemListener extends JComboBox<String> implements
            ItemListener {

        public JComboBoxItemListener(String[] s) {
            super(s);
            addItemListener(this);
        }

        public void itemStateChanged(ItemEvent evt) {
            if (comboBoxItems[0].compareTo((String) evt.getItem()) == 0) {
                AfficheSessionJeu();
            }
            if (comboBoxItems[1].compareTo((String) evt.getItem()) == 0) {
                AfficheSessionCreation();
            }
            switch (sessionActive) {
            case J:
                sessionJeu.grabFocus();
                break;
            case C:
                sessionCreation.grabFocus();
                break;
            }
        }
    }

    /**
     * Lanceur
     */
    public static void main(String[] args) {
        Surizarinku suri = new Surizarinku(new Session_jeu(10, 10),
                new Session_creation(10, 10));
        suri.setResizable(false);
        suri.setLocation(50, 50);
        suri.pack();
        suri.setVisible(true);
    }
}
```

6 Annexe : Exemple de fichier grille lu par le programme

```
// -------------------------------------------------
// Fichier d'enregistrement de grille de Surizarinku
// Nom : Grille Facile 1.txt
// Date : 20/03/2015 22:32:40
// -------------------------------------------------
// Taille :
T 10 10
// Grille
3 9 9 1 9 9 3 3 9 0
9 9 3 9 1 9 9 9 9 9
2 9 0 9 1 9 2 1 1 9
2 9 1 9 1 1 9 9 9 3
9 9 9 3 9 9 1 2 3 9
9 3 1 3 9 9 2 9 9 9
3 9 9 9 2 3 9 1 9 3
9 3 2 2 9 0 9 1 9 3
9 9 9 9 9 2 9 3 9 9
1 9 2 0 9 9 3 9 9 1
// Solution CoteBas
0 1 0 0 0 1 1 0 1 0 0
0 0 0 1 0 0 0 1 0 1 0
0 0 1 0 1 0 1 1 1 1 0
0 1 1 0 0 0 1 1 0 0 1
0 1 1 0 1 1 0 0 1 0 0
0 1 1 0 1 0 1 1 0 1 0
0 1 1 0 1 0 1 0 0 0 1
0 1 1 0 0 1 0 1 0 0 1
0 0 1 0 0 1 0 0 1 0 1
0 0 0 1 0 0 1 0 0 0 0
0 0 1 0 0 0 0 1 0 1 0
// Solution CoteDroit
0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 1 0 1 1 1 0 0
1 1 1 1 1 0 0 0 0 1 0
1 0 0 0 0 1 0 0 0 0 0
0 0 1 0 0 0 0 1 0 1 1
1 0 0 1 0 1 0 0 1 1 1
0 0 1 0 1 0 0 1 0 0 1
1 0 0 1 0 1 1 1 0 1 0
0 0 1 1 1 0 0 0 0 0 1
0 1 0 1 0 1 0 1 1 1 0
0 1 1 0 0 0 1 1 1 1 0
```