Université Sorbonne Paris Nord
Sup Galilée
Spécialité Informatique

Matière : Programmation orientée objet

Automate Cellulaire

Étudiants :
Anis TRABELSI
Zeineb BOUJMIL

Enseignant :
John CHAUSSARD
Pierre FOUILHOUX

15 juin 2024

article listings xcolor

Engagement de non-plagiat

Nous, soussigné(e)s Anis Trabelsi et Zeineb Boujmil, étudiant(e)s en 1ere année spécialité Informatique d'école d'ingénieur à Sup Galilée, déclarons être pleinement conscient(e)s que la copie de tout ou partie d'un document, quel qu'il soit, publié sur tout support existant, y compris sur Internet, constitue une violation du droit d'auteur ainsi qu'une fraude caractérisée, tout comme l'utilisation d'outils d'Intelligence Artificielle pour générer une partie de ce rapport ou du code associé. En conséquence, nous déclarons que ce travail ne comporte aucun plagiat, et assurons avoir cité explicitement, à chaque fois que nous en avons fait usage, toutes les sources utilisées pour le rédiger.

Fait à Paris , le 15/06/2024

Signatures :

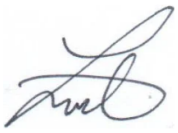
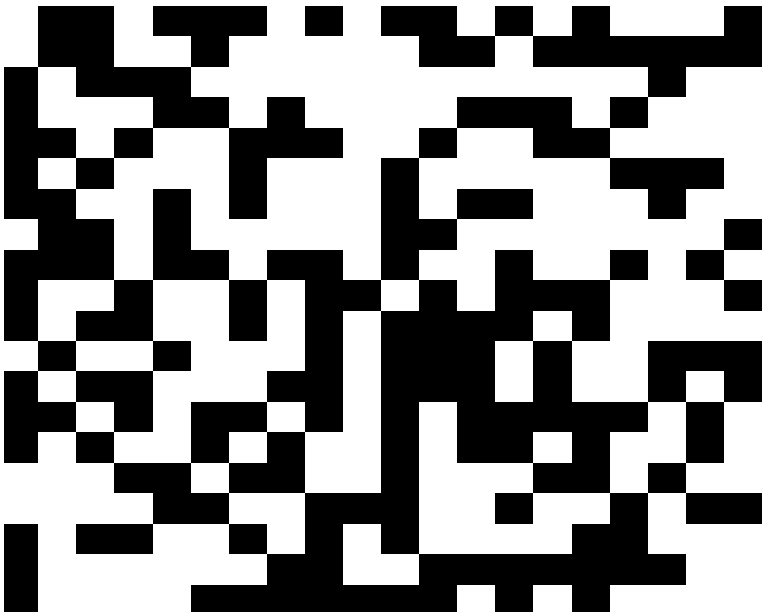


Table des matières

1 Introduction

Ce rapport est le fruit d'un parcours académique dédié à l'étude approfondie de la matière programmation orientée objet , une branche essentielle de notre formation , à l'école d'ingénieur sup galilée . L'objectif de ce document est de synthétiser les connaissances acquises et de les appliquer à un cas pratique, Automates cellulaires, illustrant ainsi la synergie entre théorie et pratique.

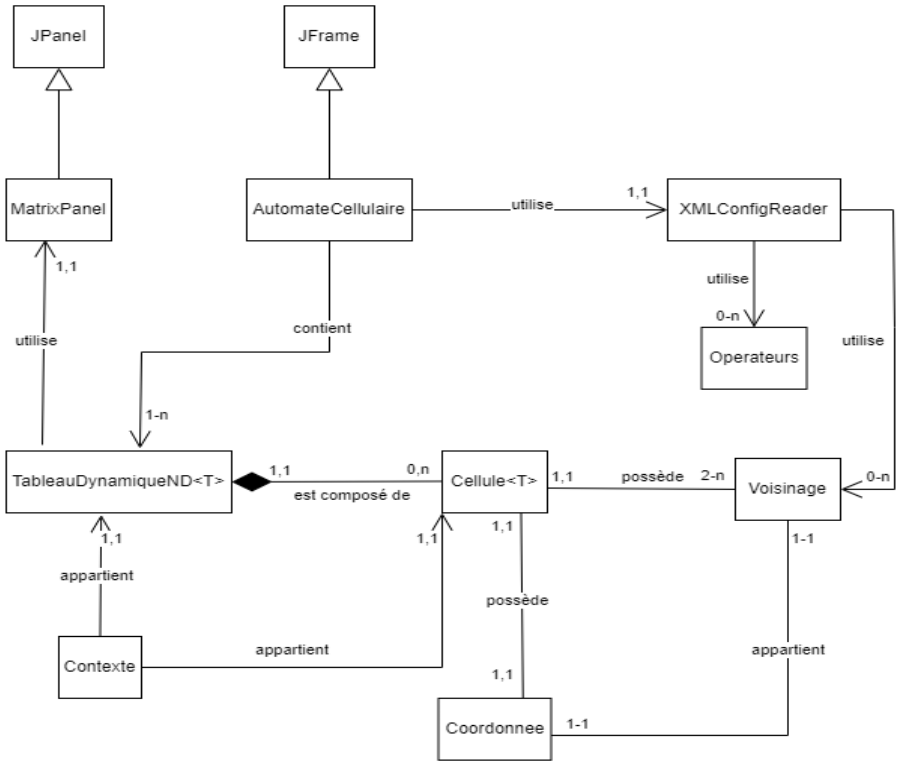


Dans le domaine de la recherche informatique et de l'ingénierie logicielle, les automates cellulaires se présentent comme un champ d'étude fascinant où des règles simples peuvent engendrer des comportements complexes et souvent imprévisibles. Ces systèmes dynamiques, caractérisés par des règles locales appliquées à des structures en grille, peuvent simuler des phénomènes allant de la modélisation de processus biologiques à la résolution de problèmes dans les domaines de l'optimisation et de l'intelligence artificielle. La capacité à implémenter et observer ces systèmes offre non seulement une fenêtre sur des comportements émergents mais ouvre aussi la voie à des applications pratiques dans des domaines aussi variés que la cryptographie, la modélisation climatique, et bien au-delà.

Ce rapport détaille la conception et l'implémentation d'un système modulaire capable de simuler une variété d'automates cellulaires dans un environnement Java. Le projet vise à développer une architecture flexible et extensible capable de gérer différents types d'automates, de la simple simulation en une dimension à des configurations plus complexes en deux ou trois dimensions. L'objectif est de fournir une plateforme robuste pour l'expérimentation avec diverses règles d'évolution et configurations initiales, permettant ainsi d'explorer le potentiel des automates cellulaires pour générer des comportements complexes à partir de règles simples.

2 Diagramme de classe

Nous commencerons par présenter un diagramme de classes global, sans entrer dans le détail des classes. Ensuite, nous examinerons chaque classe du projet individuellement, en décrivant complètement ses fonctions et attributs. Nous mettrons également en lumière les aspects cruciaux pour le fonctionnement du projet, en détaillant les points importants et les segments essentiels du code.



2.1 MatrixPanel et JPanel et AutomateCellulaire et JFrame

Il s'agit d'une relation d'héritage.
AutomateCellulaire hérite de l'interface graphique `JFrame`, ce qui signifie qu'il est une fenêtre Swing avec toutes les fonctionnalités de `JFrame`.
MatrixPanel hérite de l'interface graphique `JPanel`, ce qui signifie qu'il prend toutes les caractéristiques et comportements de `JPanel`.

2.2 AutomateCellulaire et XMLConfigReader

Il s'agit d'une relation de dépendance.
AutomateCellulaire utilise `XMLConfigReader` pour charger et interpréter les fichiers de configuration XML, ce qui est essentiel pour l'initialisation de l'automate et l'interprétation des règles.

2.3 AutomateCellulaire et TableauDynamiqueND<T>

Il s'agit d'une relation de composition.
AutomateCellulaire contient `TableauDynamiqueND<T>`, ce qui signifie qu'il gère et utilise

un tableau dynamique pour maintenir l'état de l'automate cellulaire.

2.4 MatrixPanel et TableauDynamiqueND<T>

Il s'agit d'une relation de dépendance.
MatrixPanel dépend de `TableauDynamiqueND<T>` pour obtenir les données à afficher.

2.5 TableauDynamiqueND<T> et Cellule<T>

Il s'agit d'une relation de composition.
`TableauDynamiqueND<T>` est composé de `Cellule<T>`, chaque élément du tableau est une instance de `Cellule<T>`.

2.6 Contexte et TableauDynamiqueND<Integer>

Il s'agit d'une relation d'association.
Contexte contient `TableauDynamiqueND<Integer>`, ce qui signifie qu'il utilise cette instance pour effectuer des opérations sur l'automate cellulaire.

2.7 Contexte et Cellule<T>

Il s'agit d'une relation de dépendance.
Contexte contient les coordonnées de `Cellule<T>`, ce qui signifie qu'il utilise cette instance pour gérer les positions des cellules dans le tableau. La cardinalité 1,1 explique qu'un contexte traite une seule cellule en question.

2.8 Voisinage et Coordonnee

Il s'agit d'une relation d'association.
Voisinage contient `Coordonnee`, ce qui signifie qu'il maintient une liste de positions relatives pour les cellules voisines.

2.9 Cellule et Voisinage

Il s'agit d'une relation de dépendance.
`Cellule` dépend de `Voisinage` pour obtenir les informations relatives aux cellules voisines.

2.10 Cellule et Coordonnee

Il s'agit d'une relation de dépendance.
`Cellule` dépend de `Coordonnee` pour connaître sa position dans la grille.

2.11 XMLConfigReader et Voisinage

Il s'agit d'une relation de dépendance.
`XMLConfigReader` utilise `Voisinage` pour lire et interpréter les configurations relatives aux voisins des cellules.

2.12 XMLConfigReader et Operateurs

Il s'agit d'une relation de dépendance.
`XMLConfigReader` utilise `Operateurs` pour effectuer des calculs nécessaires lors de la lecture et de l'interprétation des fichiers de configuration XML.

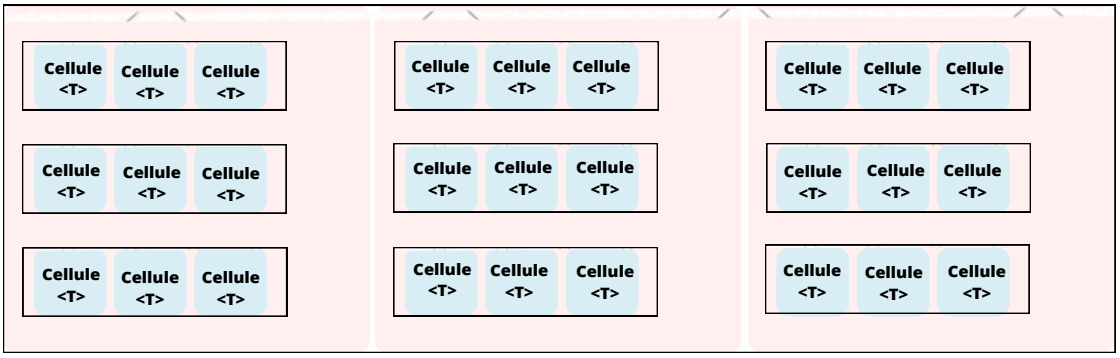
On va définir dans ce qui suit les classes en détail :

3 TableauDynamiqueND<T>

La structure de données TableauDynamiqueND<T> est la base du projet . Cette classe a été conçue pour gérer de manière efficace et flexible des grilles de données multidimensionnelles, où chaque dimension peut varier dynamiquement en taille. Le tableau n'est pas limité à un nombre fixe de dimensions, ce qui permet son utilisation dans des simulations d'automates cellulaires de diverses complexités, allant de simples automates unidimensionnels à des configurations plus complexes en trois dimensions ou plus.

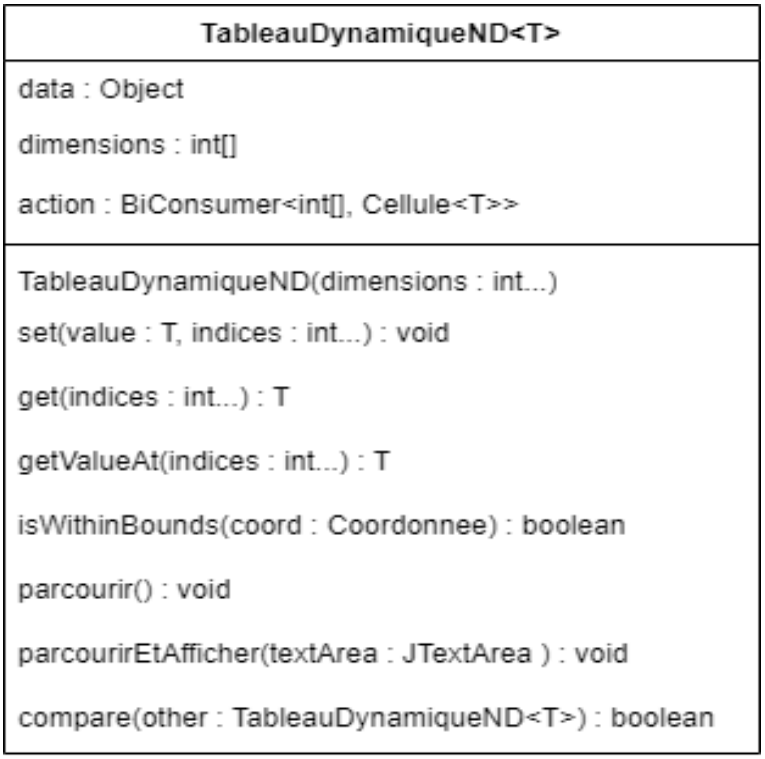
Cette structure est implémentée de manière récursive, où un tableau n-dimensionnel est conceptualisé comme une collection de tableaux (n-1)-dimensionnels. Cette approche récursive facilite non seulement la gestion des données mais permet également une initialisation et un accès aux cellules du tableau de façon intuitive et performante. Chaque cellule du tableau est encapsulée dans une instance de la classe Cellule<T>, qui peut contenir des données de tout type générique T .

Prenant un exemple présenté avec un tableau de 3 dimensions :



La manipulation des cellules se fait via des indices qui correspondent à chaque dimension du tableau, permettant des opérations comme la mise à jour, l'accès et l'évaluation des états des cellules selon des règles définies.

3.1 Diagramme de la classe



Cette classe prend en charge le parcours de toutes les cellules pour l'application de règles ou de modifications . pour cela on trouve ces deux methodes :

3.2 Parcours et affichage :

La fonction parcourirEtAfficher parcours reccursivement le tableau et affiche la valeur de chaque cellule dans une aire de texte (JTextArea), ce qui est utile pour notre interface graphique qu'on va détailler son implementation en ce qui suit .
voici son code :

```
1     public void parcourirEtAfficher(JTextArea textArea) {
2         parcourirEtAfficherRec(data, new int[dimensions.length], 0,
              textArea);
3     }
4
5     private void parcourirEtAfficherRec(Object array, int[] indices
          , int depth, JTextArea textArea) {
6         if (depth == dimensions.length) {
7             textArea.append(((Cellule<T>) array).getValue() + " ");
8         } else {
9             Object[] objArray = (Object[]) array;
10            for (int i = 0; i < dimensions[depth]; i++) {
11                indices[depth] = i;
```

```
                parcourirEtAfficherRec(objArray[i], indices, depth
                    + 1, textArea);
                if (depth < dimensions.length - 1) {
                    textArea.append("\n");
                }
            }
        }
    }
```

3.3 Parcours et action :

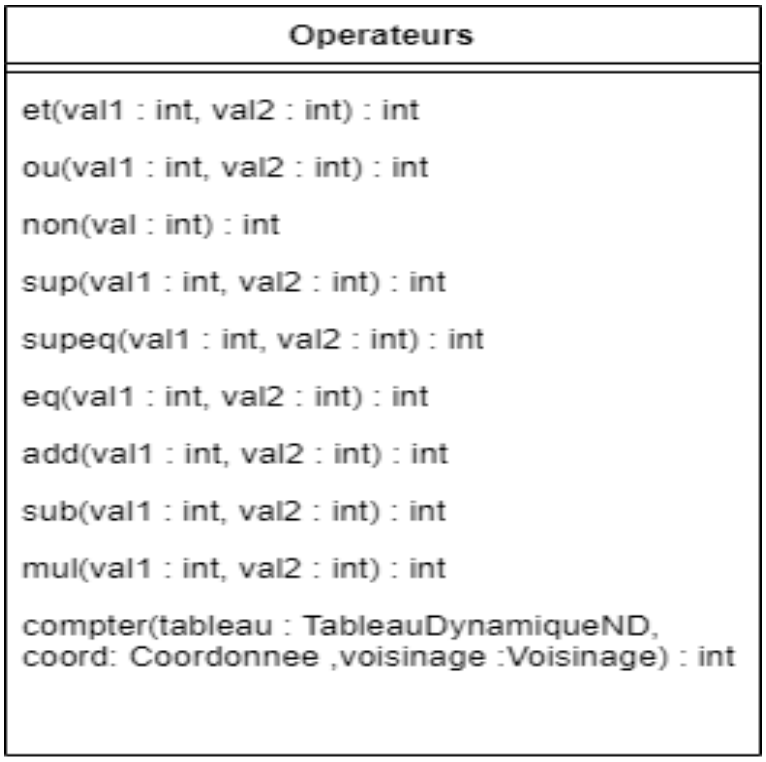
La fonction Parcourir itère récursivement sur chaque élément du tableau et à chaque cellule atteinte (niveau le plus profond de la récursivité), elle exécute l'action action, si celle-ci n'est pas nulle. L'action est l'application de la règle entrée .

```
    public void parcourir() {
        int[] indices = new int[dimensions.length];
        parcourirRec(data, indices, 0);
    }

    private void parcourirRec(Object array, int[] indices, int
        index) {
        if (index == dimensions.length - 1) {
            Cellule<T>[] cellArray = (Cellule<T>[]) array;
            for (int i = 0; i < dimensions[index]; i++) {
                indices[index] = i;
                if (action != null) {
                    action.accept(indices, cellArray[i]);
                }
            }
        } else {
            Object[] objArray = (Object[]) array;
            for (int i = 0; i < dimensions[index]; i++) {
                indices[index] = i;
                parcourirRec(objArray[i], indices, index + 1);
            }
        }
    }
```

Même la fonction chargée de comparer deux tableaux opère de manière récursive ce qui est cohérent avec la nature intrinsèquement récursive de la structure , conçue pour gérer des données multidimensionnelles.
On souligne aussi l'importance du champ action : BiConsumer<int[], Cellule<T» qui peut être configuré pour exécuter une action spécifique sur chaque cellule du tableau. BiConsumer est une interface fonctionnelle qui prend deux arguments et ne retourne pas de résultat. Ici, elle prend un tableau d'indices représentant la position d'une cellule et la cellule elle-même.

4 Operateurs

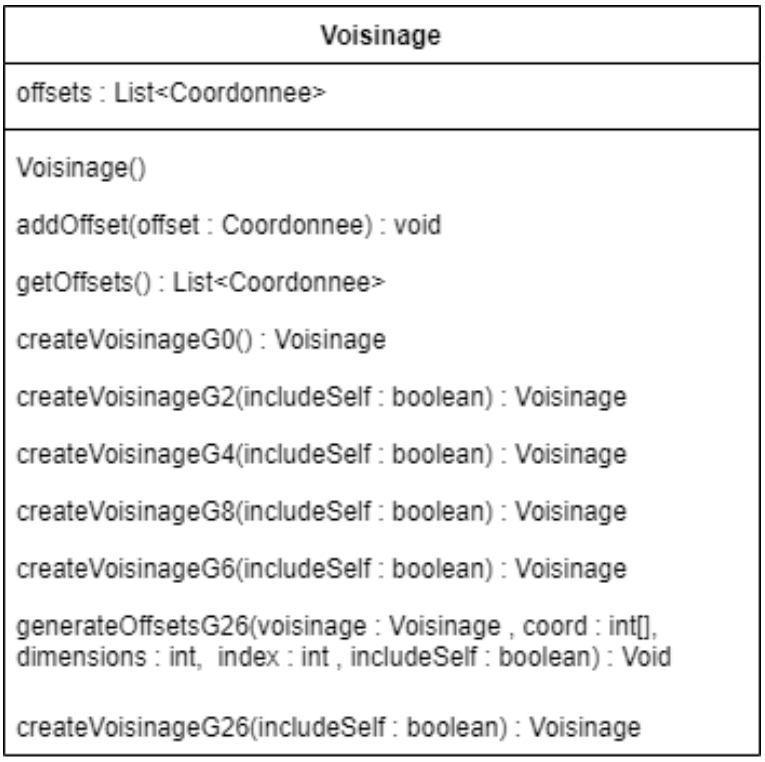
4.1 Diagramme de la classe



Les fonctions telles que et, ou, non, sup, supeq, eq, add, sub, et mul sont des opérations basiques qui correspondent à des opérations logiques et arithmétiques standards donc leur fonctionnement est direct et intuitif.Mais on détaillera le fonctionnement de la fonction compter .
Ses paramètres sont :
tableau : TableauDynamiqueND - Il s'agit de la grille représentant l'automate cellulaire. Ce tableau contient les cellules et leurs états.
coord : Coordonnee - Les coordonnées de la cellule centrale pour laquelle le voisinage doit être évalué.
voisinage : Une instance de Voisinage qui définit les règles de proximité pour la cellule spécifiée. Ce paramètre détermine quelles cellules adjacentes doivent être prises en compte pour le calcul.
La fonction compte le nombre de voisins qui sont "actives" .

5 Voisinage

5.1 Diagramme de la classe



La classe Voisinage joue un rôle crucial en définissant les cellules adjacentes qui influencent le comportement d'une cellule donnée selon les règles de l'automate. Les méthodes createVoisinageG0, createVoisinageG2, createVoisinageG4, createVoisinageG8 et createVoisinageG26 illustrent différentes stratégies pour définir ces environnements locaux comme suit :
createVoisinageG0 : Crée un voisinage où seule la cellule elle-même est incluse .
Remarque : addOffset est une méthode qui ajoute une Coordonnee au voisinage. Chaque Coordonnee représente une position relative par rapport à une cellule centrale. En ajoutant ces coordonnées, on définit les positions des voisins autour de la cellule centrale.

```
1     public static Voisinage createVoisinageG0() {
2           Voisinage voisinage = new Voisinage();
3           voisinage.addOffset(new Coordonnee(0, 0)); // La cellule
               elle-m me
4           return voisinage;
5       }
```

createVoisinageG2 : Pour les automates unidimensionnels, cette fonction définit les voisins immédiats à gauche et à droite de la cellule centrale, avec une option pour inclure la cellule elle-même si G2 est entrée avec un étoile *.

```
1       public static Voisinage createVoisinageG2(boolean includeSelf
           ) {
2           Voisinage voisinage = new Voisinage();
```

```
3           if (includeSelf) {
4               voisinage.addOffset(new Coordonnee(0)); // La cellule
                    elle-m me
5           }
6           voisinage.addOffset(new Coordonnee(-1)); // Voisin de
                gauche
7           voisinage.addOffset(new Coordonnee(1));  // Voisin de
                droite
8           return voisinage;
9       }
```

createVoisinageG4 : Approprié pour les grilles 2D, elle crée un voisinage en forme de croix, capturant les cellules directement adjacentes verticalement et horizontalement avec le meme principe de l'étoile .

```
1       public static Voisinage createVoisinageG4(boolean
            includeSelf) {
2           Voisinage voisinage = new Voisinage();
3           if (includeSelf) {
4               voisinage.addOffset(new Coordonnee(0, 0)); // La
                    cellule elle-m me
5           }
6           voisinage.addOffset(new Coordonnee(-1, 0)); // Voisin de
                gauche
7           voisinage.addOffset(new Coordonnee(1, 0));  // Voisin de
                droite
8           voisinage.addOffset(new Coordonnee(0, -1)); // Voisin du
                dessus
9           voisinage.addOffset(new Coordonnee(0, 1));  // Voisin du
                dessous
10          return voisinage;
11      }
```

createVoisinageG8 : Également pour les grilles 2D, mais cette méthode inclut toutes les cellules adjacentes autour de la cellule centrale, y compris les diagonales, permettant une interaction plus riche toujours avec le meme principe de l'étoile .

```
1           public static Voisinage createVoisinageG8(boolean
                includeSelf) {
2           Voisinage voisinage = new Voisinage();
3           if (includeSelf) {
4               voisinage.addOffset(new Coordonnee(0, 0)); // La
                    cellule elle-m me
5           }
6           voisinage.addOffset(new Coordonnee(-1, 0)); // Voisin de
                gauche
7           voisinage.addOffset(new Coordonnee(1, 0));  // Voisin de
                droite
8           voisinage.addOffset(new Coordonnee(0, -1)); // Voisin du
                dessus
9           voisinage.addOffset(new Coordonnee(0, 1));  // Voisin du
                dessous
```

```
10            voisinage.addOffset(new Coordonnee(-1, -1)); // Voisin
                  haut gauche
11            voisinage.addOffset(new Coordonnee(1, -1));  // Voisin
                  haut droite
12            voisinage.addOffset(new Coordonnee(-1, 1));  // Voisin
                  bas gauche
13            voisinage.addOffset(new Coordonnee(1, 1));  // Voisin bas
                   droite
14            return voisinage;
15        }
```

createVoisinageG6 : Ce voisinage est typiquement utilisé dans des grilles tridimensionnelles où chaque cellule peut interagir avec ses voisins immédiats alignés le long des axes x, y, et z.
Les voisins inclus sont : dans le plan x-y , les quatre voisins immédiats dans le plan horizontal (gauche, droite, avant, arrière). et dans l'axe z , les deux voisins immédiats le long de l'axe vertical (haut et bas).

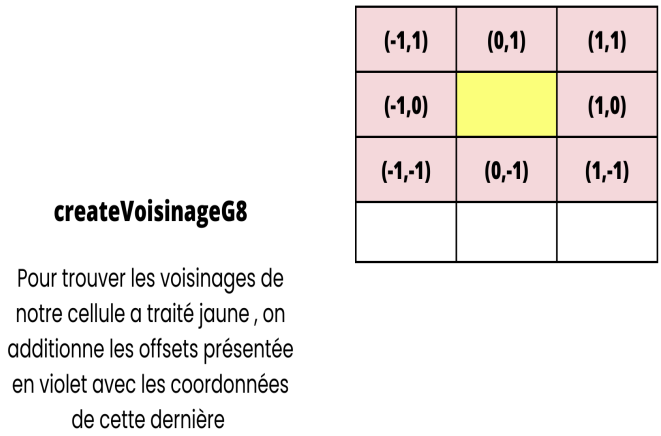
```
1             public static Voisinage createVoisinageG6(boolean
                  includeSelf) {
2             Voisinage voisinage = new Voisinage();
3             if (includeSelf) {
4                 voisinage.addOffset(new Coordonnee(0, 0, 0)); // La
                      cellule elle-m me
5             }
6             voisinage.addOffset(new Coordonnee(-1, 0, 0)); // Voisin
                  de gauche
7             voisinage.addOffset(new Coordonnee(1, 0, 0));  // Voisin
                  de droite
8             voisinage.addOffset(new Coordonnee(0, -1, 0)); // Voisin
                  du dessus
9             voisinage.addOffset(new Coordonnee(0, 1, 0));  // Voisin
                  du dessous
10            voisinage.addOffset(new Coordonnee(0, 0, -1)); // Voisin
                  avant
11            voisinage.addOffset(new Coordonnee(0, 0, 1));  // Voisin
                  arri re
12            return voisinage;
13        }
```

createVoisinageG26 Cette méthode crée un voisinage tridimensionnel incluant toutes les cellules adjacentes à une cellule centrale. Si includeSelf est vrai, la cellule centrale elle-même est également incluse dans le voisinage.

```
1     // Cree un voisinage en 3D avec 26 voisins possibles (voisins
          directs et diagonaux dans un espace 3D)
2     public static Voisinage createVoisinageG26(boolean includeSelf) {
3         Voisinage voisinage = new Voisinage();
4         if (includeSelf) {
5             // Ajoute la cellule elle-m me si includeSelf est true
6             voisinage.addOffset(new Coordonnee(0, 0, 0));
7         }
```

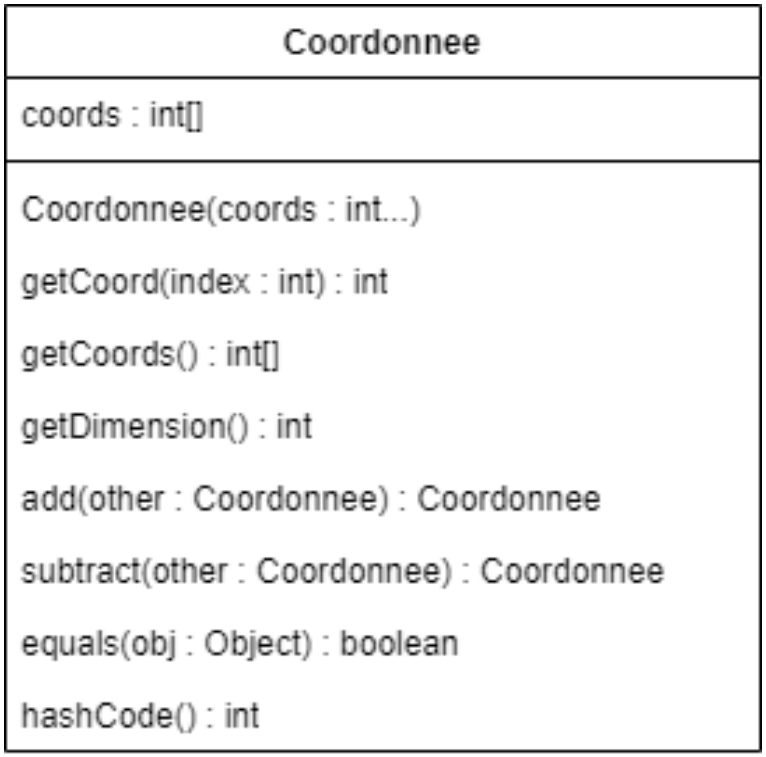
```
8       int[] coord = new int[3];
9       // Genere tous les offsets pour un voisinage de 26 en 3D
10      generateOffsetsG26(voisinage, coord, 3, 0, includeSelf);
11      return voisinage;
12  }
13
14  // Methode recursive pour generer les offsets dans un espace N
        dimensions
15  private static void generateOffsetsG26(Voisinage voisinage, int[]
        coord, int dimensions, int index, boolean includeSelf) {
16      // Si l'index atteint le nombre de dimensions, nous avons une
            combinaison compl te de coordonnees
17      if (index == dimensions) {
18          boolean isSelf = true;
19          // Verifie si la coordonnee generee represente la cellule
                elle-m me (tous les indices sont zero)
20          for (int i = 0; i < dimensions; i++) {
21              if (coord[i] != 0) {
22                  isSelf = false;
23                  break;
24              }
25          }
26          // Ajoute la coordonnee au voisinage si ce n'est pas la
                cellule elle-m me ou si includeSelf est true
27          if (!isSelf || includeSelf) {
28              voisinage.addOffset(new Coordonnee(coord.clone()));
29          }
30          return;
31      }
32
33      // Boucle pour assigner -1, 0, 1    la coordonnee actuelle et
            appeler recursivement pour generer les autres dimensions
34      for (int i = -1; i <= 1; i++) {
35          coord[index] = i;
36          generateOffsetsG26(voisinage, coord, dimensions, index + 1,
                includeSelf);
37      }
38  }
```

Voici un exemple avec G8 qui explique comment les voisinages sont déterminés :



6 Coordonnee

6.1 Diagramme de la classe

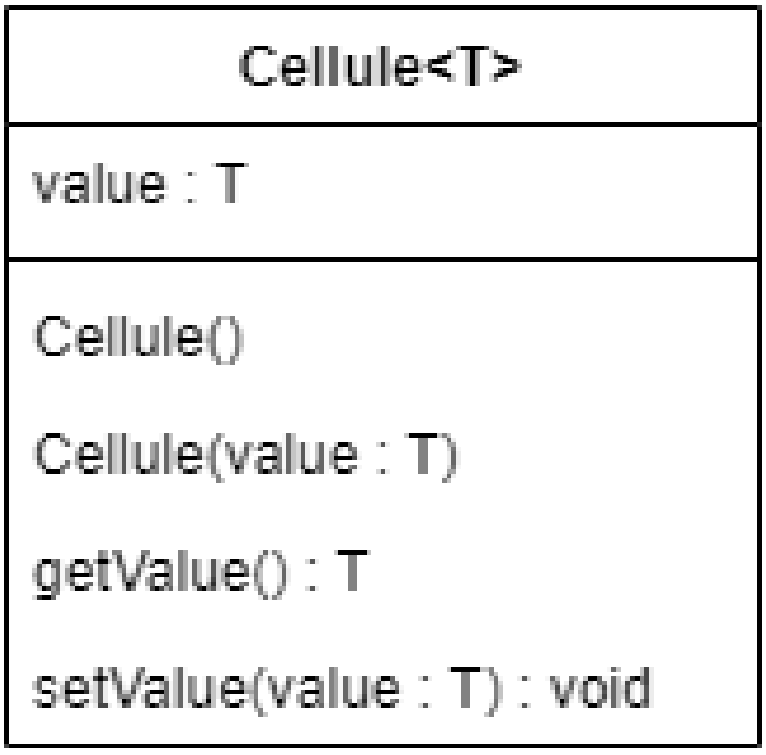


La classe Coordonnee est conçue pour gérer et manipuler les coordonnées dans un espace multidimensionnel. Elle sert principalement à encapsuler les informations de positionnement dans un espace pouvant avoir une ou plusieurs dimensions, possédant comme paramètre un tableau d'entiers qui représente ces coordonnées.

Passant aux méthodes :
getCoord permet d'accéder à une coordonnée spécifique par son indice, tandis que **getCoords** renvoie le tableau entier des coordonnées, offrant une vue complète de l'état spatial d'une instance. Pour la modification des coordonnées, la méthode add fournit des moyens d'ajouter les valeurs d'une autre instance de Coordonnee .

7 Cellule

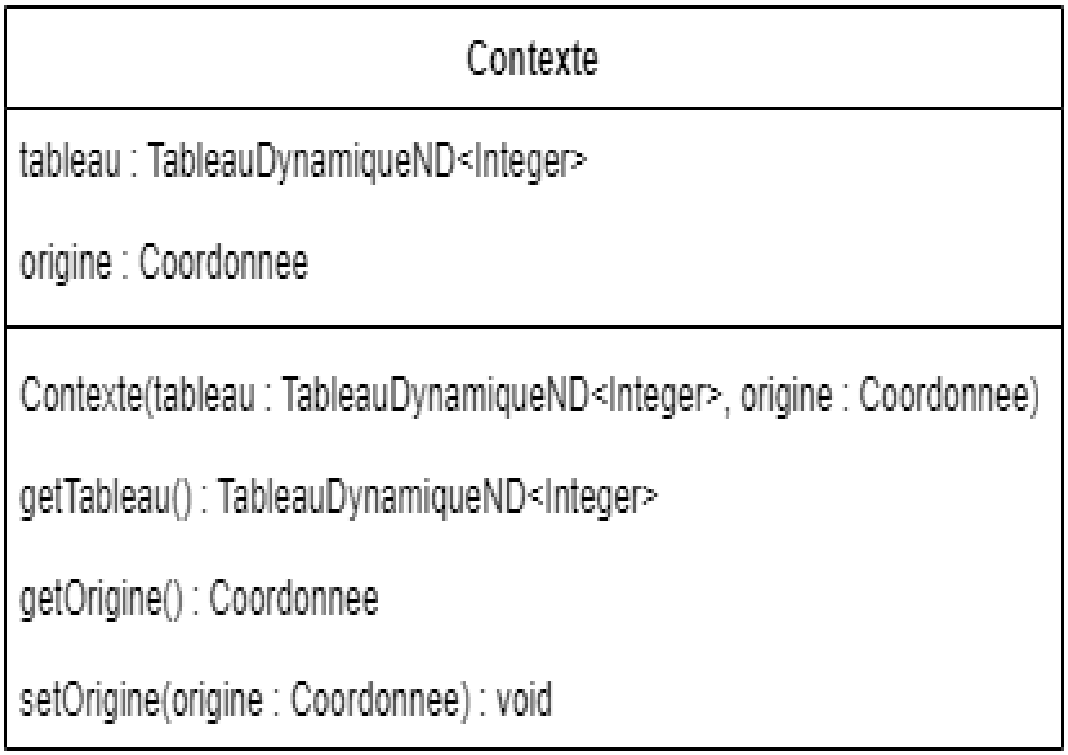
7.1 Diagramme de la classe



Malgré sa simplicité , elle joue un rôle crucial dans la gestion des données . Conçue pour être générique, elle peut contenir n'importe quel type de données (T) . Mais dans notre cas il nous suffit de considerer que le type Integer puisque nos tableaux contiennent soit 0 soit 1 .

8 Contexte

8.1 Diagramme de la classe



8.2 Classe Contexte

La classe `Contexte` est cruciale pour manipuler ou analyser des cellules individuelles dans un tableau dynamique, tout en conservant le contexte global. Elle permet de se concentrer sur une seule cellule, facilitant ainsi les opérations locales.

Intérêt de la Classe Contexte :

- **Gestion du Tableau Dynamique :** L'attribut `TableauDynamiqueND<Integer>` stocke l'état des cellules dans la grille, permettant de gérer des structures de données complexes.
- **Coordonnée d'Origine :** Cet attribut `Coordonnee origine` stocke les coordonnées de la cellule manipulée, servant de référence pour les opérations locales.

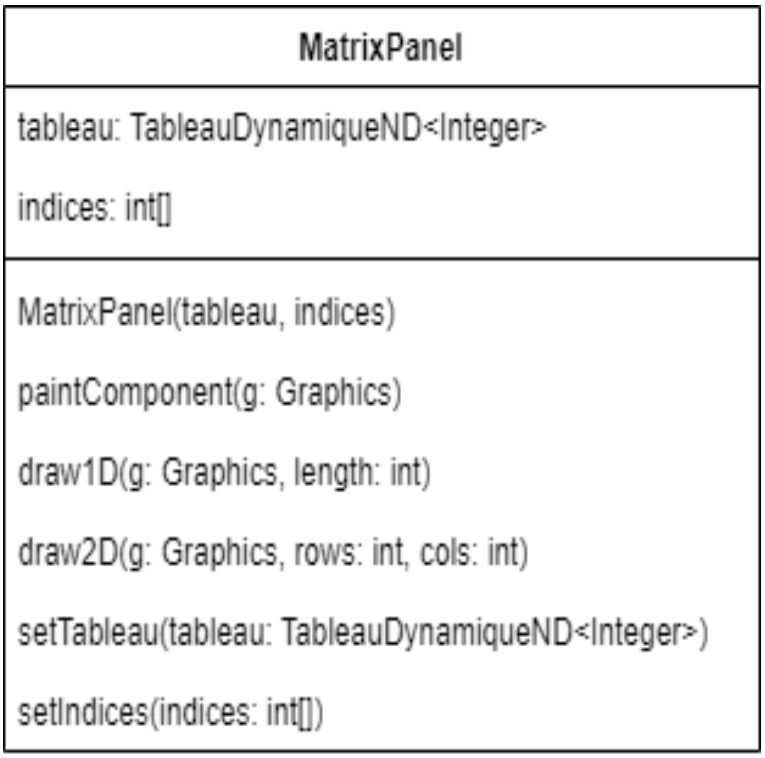
Fonctionnalités Clés :

- **Accès et Modification :** `getTableau()` et `getOrigine()` permettent d'accéder au tableau et à la coordonnée d'origine. `setOrigine(Coordonnee origine)` met à jour la coordonnée d'origine.
- **Isolation du Contexte Local :** En se concentrant sur une cellule spécifique tout en référant au tableau complet, la classe simplifie la mise à jour des états des cellules et l'application des règles.
- **Flexibilité et Extensibilité :** L'utilisation d'un tableau dynamique offre une grande flexibilité pour gérer des automates cellulaires de tailles et configurations variées.

En résumé, la classe `Contexte` permet une manipulation précise des cellules tout en maintenant le lien avec le tableau global, essentielle pour les analyses et opérations complexes.

9 MatrixPanel

9.1 Diagramme de la classe



Cette classe est un composant graphique personnalisé, étendant JPanel de la bibliothèque Swing, conçu pour afficher graphiquement les contenus d'un tableau multidimensionnel géré par un TableauDynamiqueND<Integer>.

Le constructeur **MatrixPanel** initialise le panel avec une référence à un TableauDynamiqueND<Integer> qui contient les données à afficher, et un tableau d'indices qui sert à sélectionner des coupes spécifiques du tableau multidimensionnel pour la visualisation, adapté pour les tableaux de plus de deux dimensions qu'on sélectionne de la drop down list lors de l'exécution du programme .

```
1 public MatrixPanel(TableauDynamiqueND<Integer> tableau, int[]
     initialIndices) {
2     this.tableau = tableau; // Initialise le tableau dynamique
        contenant les donnees.
3     this.indices = initialIndices; // Initialise les indices
        utilis s pour le rendu.
4     setPreferredSize(new Dimension(400, 400)); // Definit la taille
         preferee du panel.
```

```
}
```

La méthode **paintComponent** est override pour personnaliser le dessin du panel. Elle vérifie d'abord si le **tableau** n'est pas **null**. Selon la dimension du tableau (**dimensions.length**), elle choisit entre dessiner une grille **1D** ou **2D**.
draw1D et **draw2D** sont deux méthodes qui gèrent respectivement le dessin des grilles **1D** et **2D**. Elles utilisent les **indices** pour accéder aux valeurs spécifiques dans le tableau et dessinent des rectangles colorés en **noir** ou **blanc** (ou **bleu** pour **2D**) pour représenter l'état de chaque cellule selon ce principe : chaque cellule est représentée par un **rectangle**. La couleur du rectangle (**noir** ou **blanc**) reflète l'état de la cellule (active ou inactive).

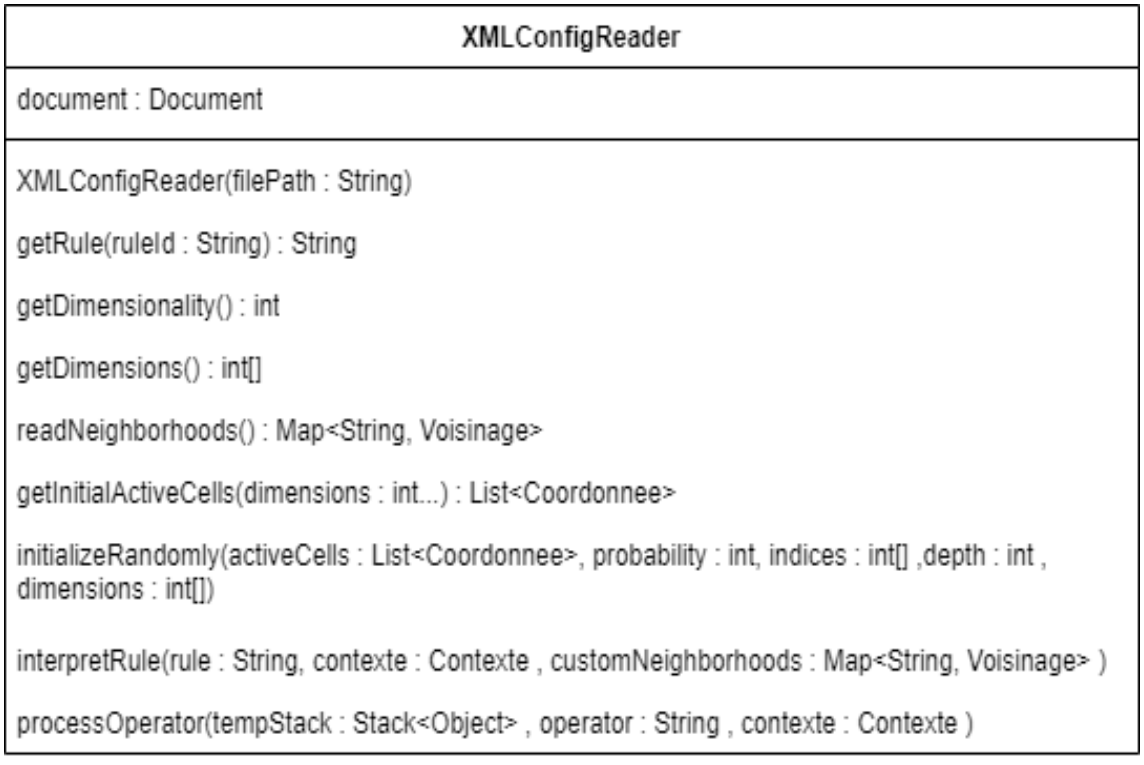
```
@Override
protected void paintComponent(Graphics g) {
        super.paintComponent(g);

        if (tableau == null) {
            return;  // Ne rien dessiner si le tableau est null
        }

        int[] dimensions = tableau.getDimensions();
        if (dimensions.length == 1) {
            draw1D(g, dimensions[0]);
        } else if (dimensions.length >= 2) {
            draw2D(g, dimensions[0], dimensions[1]);
        }
    }
```

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <grid>
        <!-- Dimensionnalit  de la grille (3D dans ce cas) -->
        <dimensionality>3</dimensionality>
        <!-- Dimensions de la grille (30x30x30) -->
        <dimensions>30,30,30</dimensions> <!-- une grille 30x30x30
            -->
        <initialState>
            <!--  tat  initial des cellules dans la grille -->
            <cell>1,1,1</cell> <!-- Cellule active    la position
                (1,1,1) -->
            <cell>1,0,0</cell> <!-- Cellule active    la position
                (1,0,0) -->
            <cell>0,0,0</cell> <!-- Cellule active    la position
                (0,0,0) -->
            <cell>1,2,1</cell> <!-- Cellule active    la position
                (1,2,1) -->
            <cell>2,2,2</cell> <!-- Cellule active    la position
                (2,2,2) -->
            <cell>2,1,1</cell> <!-- Cellule active    la position
                (2,1,1) -->
            <cell>2,0,0</cell> <!-- Cellule active    la position
                (2,0,0) -->
```

```
        </initialState>
    </grid>
    <rules>
        <!-- D finition des r gles de l'automate cellulaire -->
        <rule id="rule">
            <!-- R gle : Si le nombre de cellules vivantes dans le
                voisinage G6 est  gal  1, la cellule devient
               vivante (1), sinon elle devient morte (0) -->
            SI(EQ(COMPTER(G6*),1),1,0)
        </rule>
    </rules>
</configuration>
```

10 XMLConfigReader

10.1 Diagramme de la classe



10.2 Lecture et Parsing du Fichier XML

Au moment de la création d'une instance XMLConfigReader avec un chemin de fichier spécifié, le constructeur initialise le parsing du fichier XML. Ce processus utilise DocumentBuilderFactory et DocumentBuilder pour transformer le fichier XML en un objet Document qui sera utilisé pour interroger les données XML.

10.3 Extraction de Règles Spécifiques :

La méthode **getRule(String ruleId)** parcourt le document XML à la recherche de la balise "rule" et extrait le contenu textuel de la règle correspondant à l'ID fourni.

10.4 Acquisition des Dimensions de la Simulation :

getDimensionality() : Retourne le nombre de dimensions de l'espace de simulation.
getDimensions() : Fournit un tableau des dimensions spécifiques de la simulation.

10.5 Méthodes d'Initialisation :

getInitialActiveCells(int... dimensions) lit les configurations initiales spécifiées dans le XML pour initialiser les états des cellules dans le tableau dynamique. Cela peut inclure la configuration explicite des états de cellules ou une initialisation aléatoire basée sur une probabilité donnée.
Si une balise cell est trouvée alors la fonction initialise à 1 les cellules mentionnées dans le fichier xml sinon si la balise random est rencontrée alors l'initialisatoin se fait avec la fonction initializeRandomly en tenant compte d'un pourcentage de probabilité spécifié en entrée .

10.6 Interprétation et Application des Règles :

Dans la méthode interpretRule de la classe XMLConfigReader, deux structures de données de type Stack (pile), nommées stack et tempStack, sont utilisées pour gérer l'évaluation des expressions complexes contenues dans les règles de configuration XML. Ces piles servent à organiser et exécuter les opérations dans le bon ordre en suivant le principe du dernier entré, premier sorti (LIFO). Voici comment elles fonctionnent ensemble :

Empilement : Au fur et à mesure que les tokens sont lus, ils sont empilés sur stack jusqu'à ce qu'un opérateur nécessitant une évaluation soit atteint.
Transfert et évaluation : Les éléments nécessaires pour l'opération (les opérandes) sont transférés à tempStack pour isoler l'opération en cours.
Exécution de l'opération : L'opérateur est appliqué aux opérandes dans tempStack en prenant que le nombre de paramètres nécessaires (ceci est géré par la focntion processOperator), et le résultat est calculé.
Restauration : Après l'évaluation, le résultat est poussé sur stack, et les éléments de tempStack sont remis sur stack , préparant ainsi la pile pour l'opération suivante ou la conclusion de l'évaluation.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <grid>
        <!-- Dimensionnalite de la grille (3D dans ce cas) -->
        <dimensionality>3</dimensionality>
        <!-- Dimensions de la grille (30x30x30) -->
```

```
        <dimensions>30,30,30</dimensions> <!-- une grille 30x30x30 -->
        <initialState>
            <!-- etat initial des cellules dans la grille -->
            <cell>1,1,1</cell> <!-- Cellule active la position (1,1,1) -->
            <cell>1,0,0</cell> <!-- Cellule active la position (1,0,0) -->
            <cell>0,0,0</cell> <!-- Cellule active la position (0,0,0) -->
            <cell>1,2,1</cell> <!-- Cellule active la position (1,2,1) -->
            <cell>2,2,2</cell> <!-- Cellule active la position (2,2,2) -->
            <cell>2,1,1</cell> <!-- Cellule active la position (2,1,1) -->
            <cell>2,0,0</cell> <!-- Cellule active la position (2,0,0) -->
        </initialState>
    </grid>
    <rules>
        <!-- Definition des r gles de l'automate cellulaire -->
        <rule id="rule">
            <!-- R gle : Si le nombre de cellules vivantes dans le voisinage G6 est egal 1, la cellule devient vivante (1), sinon elle devient morte (0) -->
            SI(EQ(COMPTER(G6*),1),1,0)
        </rule>
    </rules>
</configuration>
```

Initialisation avec random :

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <grid>
        <!-- Dimensionnalite de la grille (2D dans ce cas) -->
        <dimensionality>2</dimensionality>
        <!-- Dimensions de la grille (50x50) -->
        <dimensions>50,50</dimensions>
        <initialState>
            <!-- etat initial des cellules dans la grille : 80 cellules vivantes placees aleatoirement -->
            <random>80</random>
        </initialState>
    </grid>
    <rules>
        <!-- Definition des r gles de l'automate cellulaire -->
        <rule id="rule">
            <!-- R gle : Si le nombre de cellules vivantes dans le voisinage G8 est egal 2, la cellule devient
```

```
                vivante (1), sinon elle devient morte (0) -->
            SI(EQ(COMPTER(G8), 2), 1, 0)
        </rule>
    </rules>
</configuration>
```

10.7 Et si le type de voisinage qu'on veut vérifier n'est pas prédéfinie ?

Si l'utilisateur entre une regle ou compter prend un paramètre autre que les options présentés dans la classe voisinage (autre que G0,2,4,6,8,26) , G40 par exemple , il doit obligatoirement passer dans son fichier xml les offsets des voisinage qu'il veut prendre en compte par cet nouvelle voisinage comme suit , dans une balise 'neighborhood' :

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <grid>
        <!-- Dimensionnalite de la grille (2D dans ce cas) -->
        <dimensionality>2</dimensionality>
        <!-- Dimensions de la grille (50x50) -->
        <dimensions>50,50</dimensions>
        <initialState>
            <!-- etat initial des cellules dans la grille : 80
                cellules vivantes placees aleatoirement -->
            <random>80</random>
        </initialState>
    </grid>
    <neighborhoods>
        <!-- Definition des voisinages -->
        <neighborhood id="G1DLeft">
            <!-- Voisin    gauche (dim0=0, dim1=-1) -->
            <offset dim0="0" dim1="-1"/>
        </neighborhood>
        <neighborhood id="G1DRight">
            <!-- Voisin    droite (dim0=0, dim1=1) -->
            <offset dim0="0" dim1="1"/>
        </neighborhood>
    </neighborhoods>
    <rules>
        <!-- Definition des r gles de l'automate cellulaire -->
        <rule id="rule">
            <!-- R gle : Si le nombre de cellules vivantes dans le
                voisinage G1DRight est egal    2, la cellule
                devient vivante (1), sinon elle devient morte (0)
                -->
            SI(EQ(COMPTER(G1DRight), 2), 1, 0)
        </rule>
    </rules>
</configuration>
```

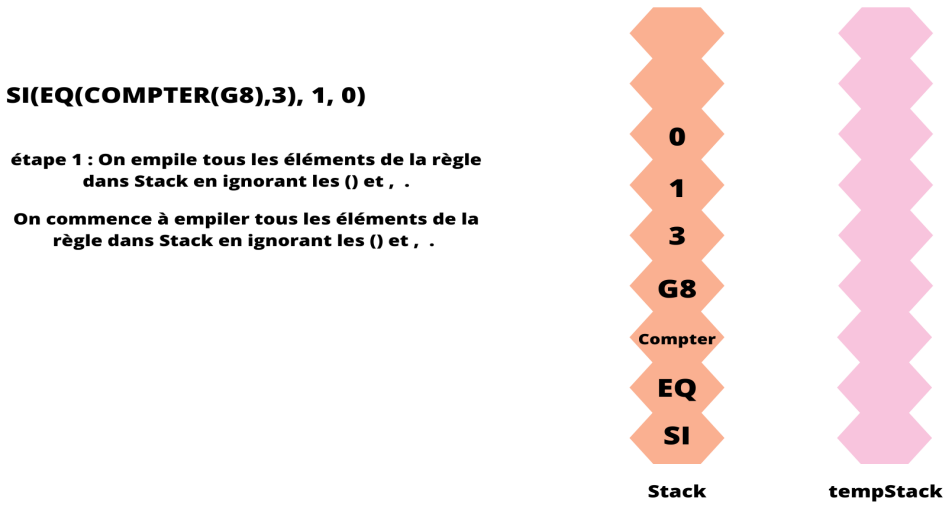
Ceci est géré par la **readNeighborhoods** qui lit les offsets et retourne un Map<String, Voisinage> ou les clés de cette carte sont des objets de type String et Les valeurs sont des objets de type Voisinage et Chaque Voisinage encapsule une collection de coordonnées d'offsets qui définissent les positions relatives des voisins par rapport à une cellule centrale .

```
1
2       public Map<String, Voisinage> readNeighborhoods() {
3           Map<String, Voisinage> neighborhoods = new HashMap<>();
4
5           // Recupere tous les elements <neighborhood> du document
               XML
6           NodeList neighborhoodList = document.getElementsByTagName("
               neighborhood");
7           for (int i = 0; i < neighborhoodList.getLength(); i++) {
8               Node node = neighborhoodList.item(i);
9
10              // Verifie si le n ud est un element
11              if (node.getNodeType() == Node.ELEMENT_NODE) {
12                  Element neighborhoodElement = (Element) node;
13
14                  // Recup re l identifiant du voisinage
15                  String neighborhoodId = neighborhoodElement.
                       getAttribute("id");
16
17                  // Recup re tous les elements <offset> pour ce
                       voisinage
18                  NodeList offsets = neighborhoodElement.
                       getElementsByTagName("offset");
19                  Voisinage voisinage = new Voisinage();
20
21                  for (int j = 0; j < offsets.getLength(); j++) {
22                      Node offsetNode = offsets.item(j);
23
24                      // Verifie si le n ud est un element
25                      if (offsetNode.getNodeType() == Node.
                           ELEMENT_NODE) {
26                          Element offsetElement = (Element)
                               offsetNode;
27                          List<Integer> coordsList = new ArrayList<>
                               ();
28
29                          // Lit les coordonnees dynamiquement en
                               fonction des noms d attributs
30                          int k = 0;
31                          while (offsetElement.hasAttribute("dim" + k
                               )) {
32                              String coord = offsetElement.
                                   getAttribute("dim" + k);
33                              coordsList.add(Integer.parseInt(coord))
                                   ;
```

```
                            k++;
                        }

                        // Convertit List<Integer> en int[]
                        int[] coords = coordsList.stream().mapToInt
                            (Integer::intValue).toArray();
                        voisinage.addOffset(new Coordonnee(coords))
                            ;
                    }
                }
                // Ajoute le voisinage    la map avec son
                    identifiant
                neighborhoods.put(neighborhoodId, voisinage);
            }
        }
        return neighborhoods;
    }
```

10.8 Details du fonctionnement de nos piles

Prenant un exemple de la gestion d'une règle par notre programme
Soit la règle : **SI(EQ(COMPTER(G8),3), 1, 0)**

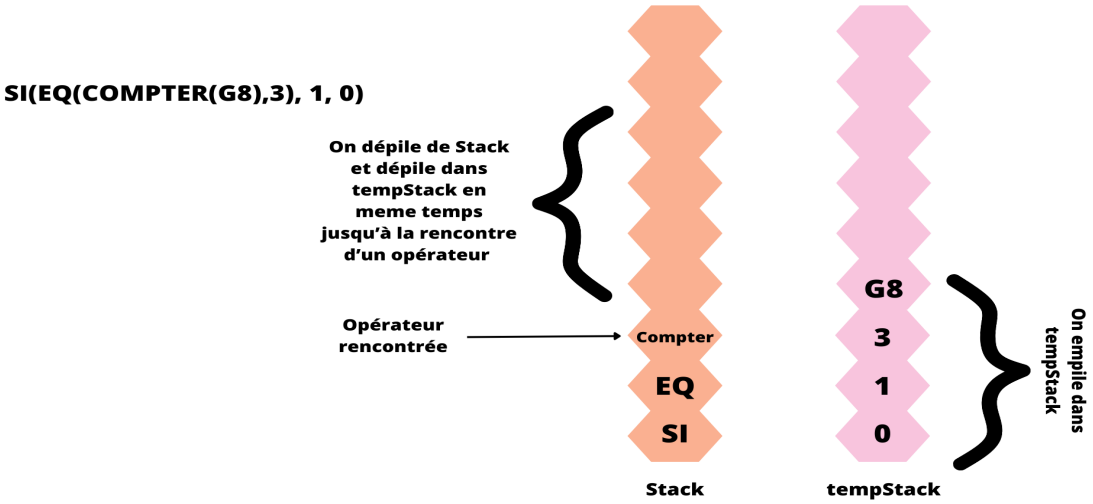


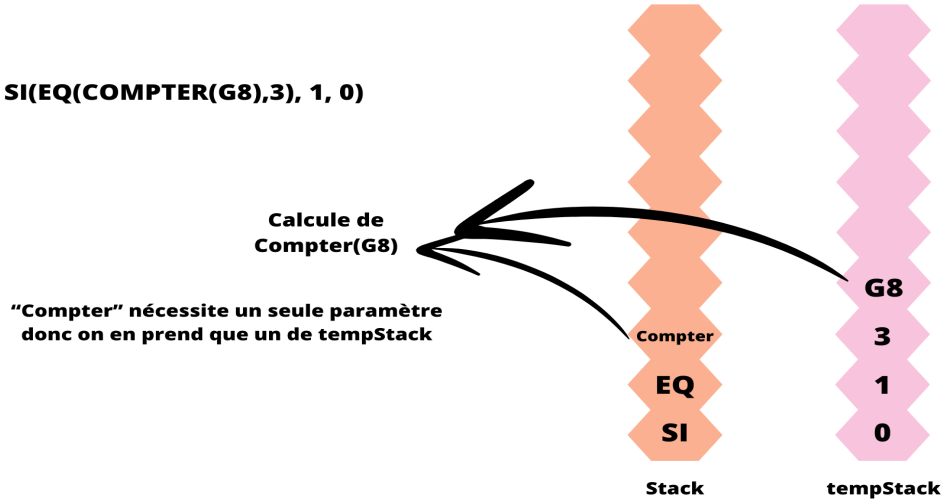

```
        Stack<Object> stack = new Stack<>();
        String[] tokens = rule.split("[(,)]");
        int dimensionality = contexte.getTableau().getDimensions().
            length;

        for (String token : tokens) {
            token = token.trim();
            if (token.isEmpty()) continue;

            try {
```

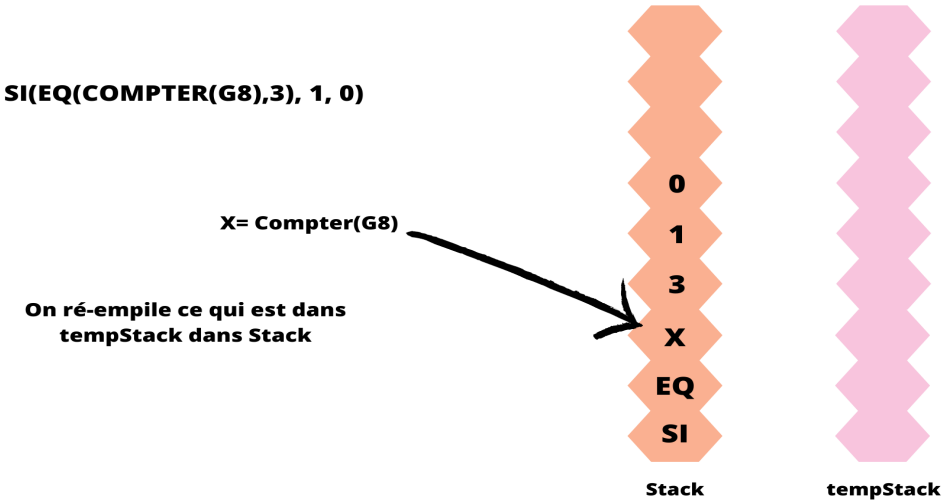
```
10                  if (customNeighborhoods.containsKey(token)) {
11                      Voisinage voisinage = customNeighborhoods.get(
                            token);
12                      // Verification de la compatibilite des
                            dimensions
13                      if (voisinage.getOffsets().size() > 0 &&
                            voisinage.getOffsets().get(0).getCoords().
                            length != dimensionality) {
14                          throw new IllegalArgumentException("Le
                                voisinage personnalise '" + token + "' n
                                'est pas compatible avec les dimensions
                                du tableau.");
15                      }
16                      stack.push(voisinage);
17                      afficherPile(stack); // Afficher la pile apr s
                            avoir pousse le voisinage
18                      continue;
19                  }
20
21                  switch (token) {
22                      case "SI":
23                      case "SUP":
24                      case "SUPEQ":
25                      case "EQ":
26                      case "COMPTER":
27                      case "ET":
28                      case "OU":
29                      case "NON":
30                      case "ADD":
31                      case "SUB":
32                      case "MUL":
33                      case "MIN":
34                      case "MAX":
35                      case "VAL":
36                          stack.push(token);
37                          afficherPile(stack); // Afficher la pile
                                apr s avoir pousse loperateur
38                          break;
39                      case "G0":
40                          stack.push(Voisinage.createVoisinageG0());
41                          afficherPile(stack); // Afficher la pile
                                apr s avoir pousse le voisinage
42                          break;
43                      case "G2":
44                          if (dimensionality != 1) {
45                              throw new IllegalArgumentException("G2
                                    sapplique uniquement aux tableaux 1D
                                    .");
46                          }
47                          stack.push(Voisinage.createVoisinageG2(
```

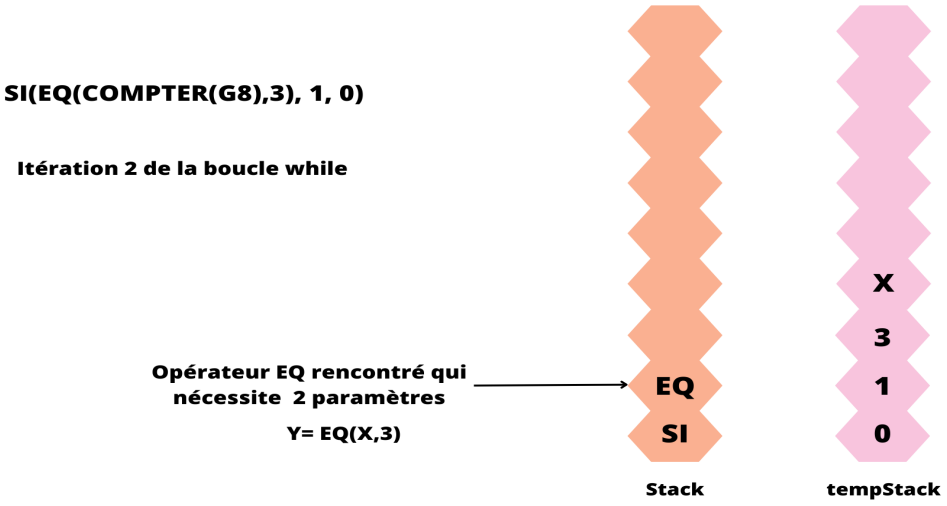
```
                    false));
48                  afficherPile(stack); // Afficher la pile
                        apr s avoir pousse le voisinage
49                  break;
50              case "G4":
51                  if (dimensionality != 2) {
52                      throw new IllegalArgumentException("G4
                            sapplique uniquement aux tableaux 2D
                            .");
53                  }
54                  stack.push(Voisinage.createVoisinageG4(
                        false));
55                  afficherPile(stack); // Afficher la pile
                        apr s avoir pousse le voisinage
56                  break;
57              case "G8":
58                  if (dimensionality != 2) {
59                      throw new IllegalArgumentException("G8
                            sapplique uniquement aux tableaux 2D
                            .");
60                  }
61                  stack.push(Voisinage.createVoisinageG8(
                        false));
62                  afficherPile(stack); // Afficher la pile
                        apr s avoir pousse le voisinage
63                  break;
64              case "G6":
65                  if (dimensionality != 3) {
66                      throw new IllegalArgumentException("G6
                            sapplique uniquement aux tableaux 3D
                            .");
67                  }
68                  stack.push(Voisinage.createVoisinageG6(
                        false));
69                  afficherPile(stack); // Afficher la pile
                        apr s avoir pousse le voisinage
70                  break;
71              case "G26":
72                  if (dimensionality != 3) {
73                      throw new IllegalArgumentException("G26
                             sapplique uniquement aux tableaux 3
                            D.");
74                  }
75                  stack.push(Voisinage.createVoisinageG26(
                        false));
76                  afficherPile(stack); // Afficher la pile
                        apr s avoir pousse le voisinage
77                  break;
78              case "G8*":
79                  stack.push(Voisinage.createVoisinageG8(true
```

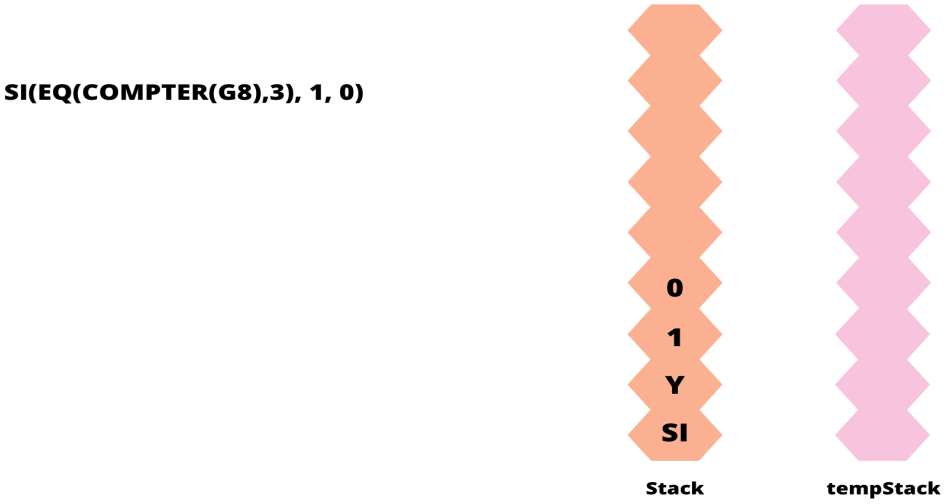
```
                        ));
80                      afficherPile(stack); // Afficher la pile
                            apr s avoir pousse le voisinage
81                      break;
82                  case "G6*":
83                      stack.push(Voisinage.createVoisinageG6(true
                            ));
84                      afficherPile(stack); // Afficher la pile
                            apr s avoir pousse le voisinage
85                      break;
86                  case "G26*":
87                      stack.push(Voisinage.createVoisinageG26(
                            true));
88                      afficherPile(stack); // Afficher la pile
                            apr s avoir pousse le voisinage
89                      break;
90                  case "G4*":
91                      stack.push(Voisinage.createVoisinageG4(true
                            ));
92                      afficherPile(stack); // Afficher la pile
                            apr s avoir pousse le voisinage
93                      break;
94                  default:
95                      if (Character.isDigit(token.charAt(0))) {
96                          stack.push(Integer.parseInt(token));
97                          afficherPile(stack); // Afficher la
                                pile apr s avoir pousse le nombre
98                      }
99                      break;
100             }
101         } catch (Exception e) {
102             System.out.printf("Erreur lors du traitement du
                    token %s: %s%n", token, e.getMessage());
103             e.printStackTrace();
104             throw e;  // Reacheminer lexception apr s la
                    gestion
105         }
106     }
```

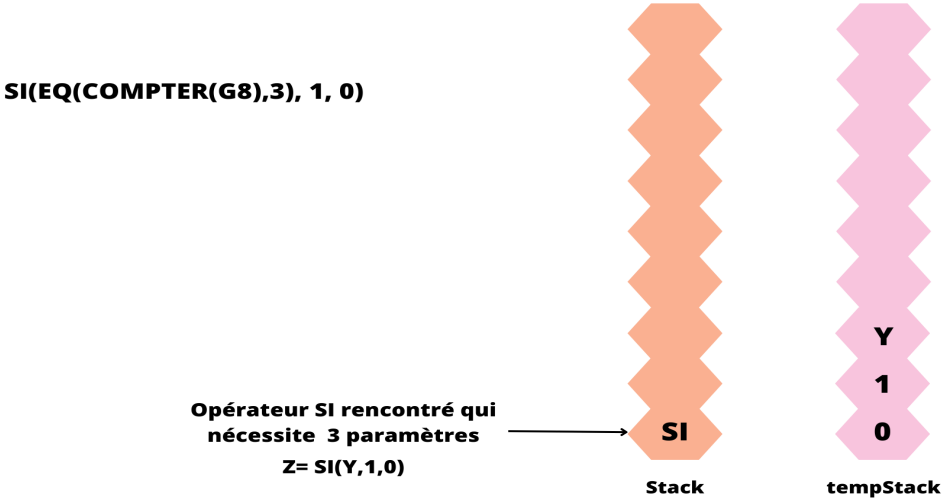

SI(EQ(COMPTER(G8),3), 1, 0)
On dépile de Stack et dépile dans tempStack en meme temps jusqu'à la rencontre d'un opérateur
Opérateur rencontrée
Compter
EQ
SI
Stack
G8
3
1
0
tempStack
On empile dans tempStack

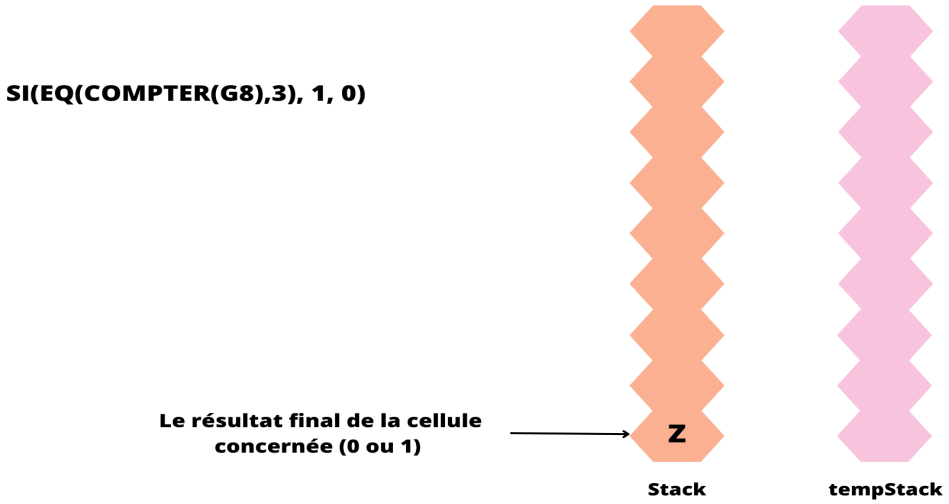

SI(EQ(COMPTER(G8),3), 1, 0)
Calcule de Compter(G8)
"Compter" nécessite un seule paramètre donc on en prend que un de tempStack
Compter
EQ
SI
Stack
G8
3
1
0
tempStack


SI(EQ(COMPTER(G8),3), 1, 0)
X= Compter(G8)
On ré-empile ce qui est dans tempStack dans Stack
0
1
3
X
EQ
SI
Stack
tempStack


SI(EQ(COMPTER(G8),3), 1, 0)
Itération 2 de la boucle while
Opérateur EQ rencontré qui nécessite 2 paramètres
Y= EQ(X,3)
EQ
SI
X
3
1
0
Stack
tempStack


SI(EQ(COMPTER(G8),3), 1, 0)
0
1
Y
SI
Stack
tempStack


SI(EQ(COMPTER(G8),3), 1, 0)
Opérateur SI rencontré qui nécessite 3 paramètres
Z= SI(Y,1,0)
SI
Y
1
0
Stack
tempStack




```
            // Maintenant que la pile est remplie, traiter les
               operateurs
        while (stack.size() > 1) {
            Stack<Object> tempStack = new Stack<>();
            Object element = stack.pop();
            tempStack.push(element);

            // Deplacer les elements jusqu    rencontrer un
               operateur
            while (!stack.isEmpty() && !(stack.peek() instanceof
               String)) {
                tempStack.push(stack.pop());
            }

            if (!stack.isEmpty() && stack.peek() instanceof String)
                {
                String operator = (String) stack.pop();
                System.out.printf("Operateur retire pour
                   traitement: %s%n", operator);

                // Traiter l operateur avec les operandes dans
                   tempStack
                Object result = processOperator(tempStack, operator
                   , contexte);
                stack.push(result);

                // Remettre les elements restants de tempStack dans
                    stack
                while (!tempStack.isEmpty()) {
                    stack.push(tempStack.pop());
                }
                afficherPile(stack); // Afficher la pile apr s
                   avoir traite l operateur
            }
```

```
26          }
27
28          if (stack.size() != 1) throw new IllegalStateException("
               etat final incorrect de la pile apr s traitement.");
29          Object result = stack.pop();
30          System.out.printf("Resultat final extrait de la pile: %s%n"
               , result);
31          return (int) result;
```

La focntion processOperator est privee car elle ne peut etre appelee que par la focntion interpretRule qui est dans la meme classe que elle .

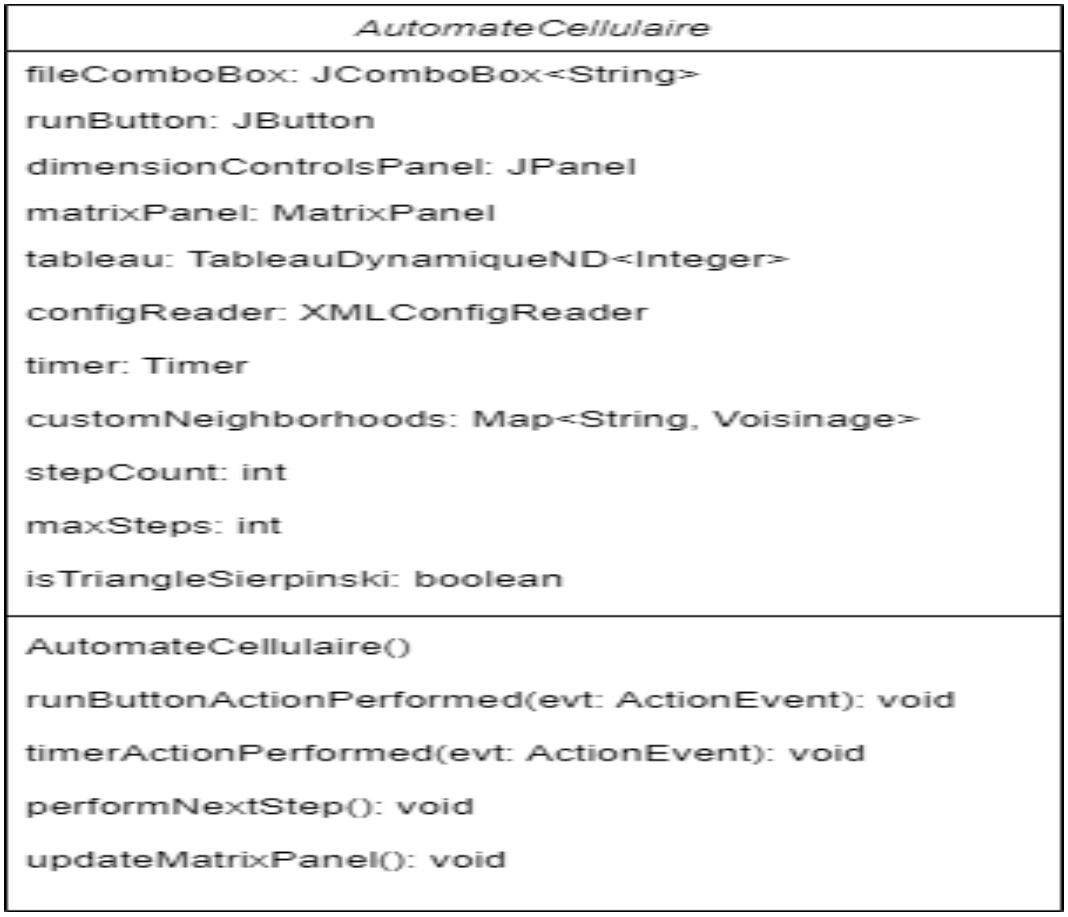
```
1       private Object processOperator(Stack<Object> tempStack, String
            operator, Contexte contexte) {
2           int val1, val2, val3, cond;
3           try {
4               switch (operator) {
5                   case "SI":
6                       cond = (int) tempStack.pop();  // Valeur si
                           faux
7                       val2 = (int) tempStack.pop();  // Valeur si
                           vrai
8                       val3 = (int) tempStack.pop();  // Condition
9                       System.out.printf("Traitement SI(%d, %d, %d)%n"
                           , cond, val2, val3);
10                      return Operateurs.si(cond, val2, val3);
11                  case "SUP":
12                      val2 = (int) tempStack.pop();  // Second
                           argument
13                      val1 = (int) tempStack.pop();  // Premier
                           argument
14                      System.out.printf("Traitement SUP(%d, %d)%n",
                           val1, val2);
15                      return Operateurs.sup(val2, val1);
16                  case "SUPEQ":
17                      val2 = (int) tempStack.pop();
18                      val1 = (int) tempStack.pop();
19                      System.out.printf("Traitement SUPEQ(%d, %d)%n",
                            val1, val2);
20                      return Operateurs.supeq(val2, val1);
21                  case "EQ":
22                      val2 = (int) tempStack.pop();
23                      val1 = (int) tempStack.pop();
24                      System.out.printf("Traitement EQ(%d, %d)%n",
                           val1, val2);
25                      return Operateurs.eq(val1, val2);
26                  case "COMPTER":
27                      Voisinage voisinage = (Voisinage) tempStack.pop
                           (); // L objet Voisinage
28                      System.out.printf("Traitement COMPTER(%s)%n",
                           voisinage);
```

```
29                      return Operateurs.compter(contexte.getTableau()
                            , contexte.getOrigine(), voisinage);
30                  case "ET":
31                      val2 = (int) tempStack.pop();
32                      val1 = (int) tempStack.pop();
33                      System.out.printf("Traitement ET(%d, %d)%n",
                            val1, val2);
34                      return Operateurs.et(val1, val2);
35                  case "OU":
36                      val2 = (int) tempStack.pop();
37                      val1 = (int) tempStack.pop();
38                      System.out.printf("Traitement OU(%d, %d)%n",
                            val1, val2);
39                      return Operateurs.ou(val1, val2);
40                  case "NON":
41                      val1 = (int) tempStack.pop();
42                      System.out.printf("Traitement NON(%d)%n", val1)
                            ;
43                      return Operateurs.non(val1);
44                  case "ADD":
45                      val2 = (int) tempStack.pop();
46                      val1 = (int) tempStack.pop();
47                      System.out.printf("Traitement ADD(%d, %d)%n",
                            val1, val2);
48                      return Operateurs.add(val1, val2);
49                  case "SUB":
50                      val2 = (int) tempStack.pop();
51                      val1 = (int) tempStack.pop();
52                      System.out.printf("Traitement SUB(%d, %d)%n",
                            val1, val2);
53                      return Operateurs.sub(val1, val2);
54                  case "MUL":
55                      val2 = (int) tempStack.pop();
56                      val1 = (int) tempStack.pop();
57                      System.out.printf("Traitement MUL(%d, %d)%n",
                            val1, val2);
58                      return Operateurs.mul(val1, val2);
59                  default:
60                      throw new IllegalArgumentException("Operateur
                            inconnu: " + operator);
61              }
62          } catch (Exception e) {
63              System.out.printf("Erreur lors de l'operation '%s': %s%
                    n", operator, e.getMessage());
64              throw e;
65          }
66      }
```

11 AutomateCellulaire

Cette classe contient notre méthode main .

11.1 Diagramme de la classe



11.2 Configuration de la fenêtre principale

```
1 setTitle("Automate Cellulaire - Triangle de Sierpinski - jeu de la
     vie");
2 setSize(1200, 800);
3 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
4 setLocationRelativeTo(null);
5 setLayout(new BorderLayout());
```

setTitle :Définit le titre de la fenêtre comme "Automate Cellulaire - Triangle de Sierpinski - jeu de la vie".
setSize : Définit la taille de la fenêtre à 1200x800 pixels.
setDefaultCloseOperation : Indique que l'application doit se fermer lorsque la fenêtre principale est fermée.
setLocationRelativeTo(null) : Centre la fenêtre sur l'écran.
setLayout(new BorderLayout()) : Définit un gestionnaire de disposition BorderLayout pour la fenêtre principale.

11.3 Initialisation des composants graphiques

```
JPanel topPanel = new JPanel(new FlowLayout());
JLabel fileLabel = new JLabel("Select XML file:");
topPanel.add(fileLabel);
```

Liste déroulante pour sélectionner le fichier XML :

```
File directory = new File(".");
File[] files = directory.listFiles((dir, name) -> name.endsWith(".
    xml"));
String[] fileNames = Arrays.stream(files).map(File::getName).
    toArray(String[]::new);
fileComboBox = new JComboBox<>(fileNames);
topPanel.add(fileComboBox);
```

Ajout du panneau supérieur à la fenêtre principale :

```
add(topPanel, BorderLayout.NORTH);
```

11.4 Initialisation des panneaux pour les contrôles de dimensions et l'affichage du tableau :

```
dimensionControlsPanel = new JPanel();
add(dimensionControlsPanel, BorderLayout.WEST);
```

Panneau pour empiler les tableaux 1D :

```
tableauPanel = new JPanel();
tableauPanel.setLayout(new BoxLayout(tableauPanel, BoxLayout.Y_AXIS
    ));
```

Panneau pour afficher la matrice du tableau :

```
matrixPanel = new MatrixPanel(null, new int[]{0});
add(new JScrollPane(matrixPanel), BorderLayout.CENTER);
```

11.5 Initialisation du Timer :

A pour role de Creer un Timer Swing qui déclenche un événement toutes les 100 millisecondes et appelle la méthode timerActionPerformed.

```
timer = new Timer(100, this::timerActionPerformed);
```

11.6 La fonction ActionPerformed

Après avoir vérifier si un fichier a été sélectionné ,

```
isTriangleSierpinski = selectedFile.equals("triangle_sierpinski.xml
    ")
```

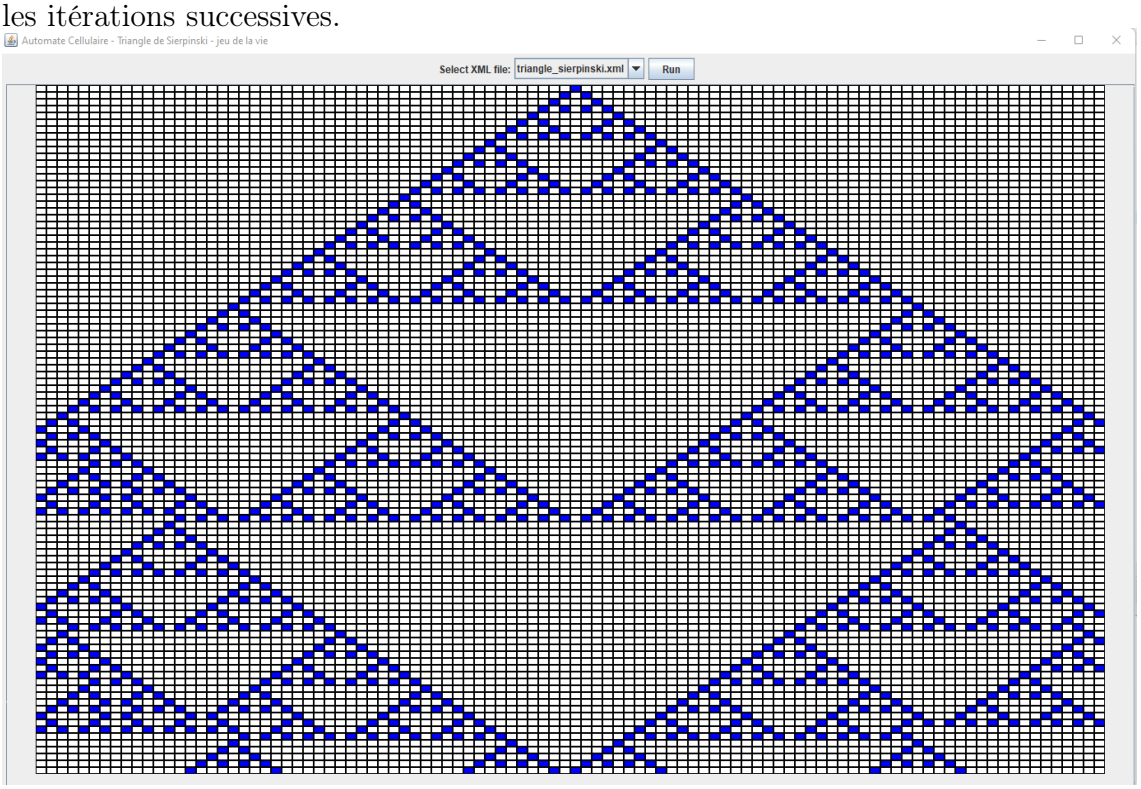
vérifie si le fichier sélectionné est triangle_sierpinski.xml pour activer un traitement spécifique.

```
configReader = new XMLConfigReader(new File(selectedFile).
   getAbsolutePath())
```

A ce niveau on fait appelle à la fonction XMLConfigReader pour utiliser tous son processus .
Un point important qu'on peut pas ignorer c'est que si on est bien dans un fichier nommée triangle_sierpinski alors la méthode appélée est A ce niveau on fait appelle à la fonction XMLConfigReader pour utiliser tous son processus .
Un point important qu'on peut pas ignorer c'est que si on est bien dans un fichier nommée triangle_sierpinski alors la méthode appélée est **display1DArrays** qui est responsable de l'affichage des états calculés en juxtaposant les tableaux 1D au lieu de les écraser.

11.6.1 C'est quoi le triangle de Sierpinski ?

Dans le cadre des automates cellulaires, le triangle de Sierpinski peut être modélisé comme une grille où chaque cellule suit une règle spécifique pour déterminer son état dans les itérations successives.



Règle Spécifique : MUL(SUB(SI(COMPTER(G2*), 1, 0), SI(EQ(COMPTER(G2*), 2), 1, 0)), SUB(1, COMPTER(G0))) .
Cette règle est utilisée pour déterminer si une cellule devient active (1) ou inactive (0) en fonction de son voisinage.
Détail de la règle :
COMPTER(G2*) : Cela compte le nombre de cellules voisines activées selon le voisinage spécifié par G2*.

SI(COMPTER(G2*), 1, 0) : Si le nombre de cellules voisines activées est supérieur à 0, alors la cellule centrale devient active (1). Sinon, elle reste inactive (0).
SI(EQ(COMPTER(G2*), 2), 1, 0) : Cette condition vérifie si le nombre de cellules voisines activées est égal à 2. Si c'est vrai, alors la cellule centrale devient active (1). Sinon, elle reste inactive (0).
SUB(SI(COMPTER(G2*), 1, 0), SI(EQ(COMPTER(G2*), 2), 1, 0)) : Cela soustrait le résultat de la condition SI(COMPTER(G2*), 1, 0) par le résultat de la condition SI(EQ(COMPTER(G2*), 2), 1, 0)).
SUB(1, COMPTER(G0)) : Cela soustrait le nombre de cellules activées dans le voisinage G0 de 1.
MUL(SUB(SI(COMPTER(G2*), 1, 0), SI(EQ(COMPTER(G2*), 2), 1, 0)), SUB(1, COMPTER(G0))) : Cette opération multiplie le résultat de la soustraction précédente par le résultat de SUB(1, COMPTER(G0)).
Alternative : SI(NON(COMPTER(G0)), EQ(COMPTER(G2), 1), 0) : Si la cellule centrale est inactive (NON(COMPTER(G0))), alors elle devient active si exactement une cellule voisine est activée (EQ(COMPTER(G2), 1)). Sinon, elle reste inactive (0).

```
if (isTriangleSierpinski) {
    add(new JScrollPane(tableauPanel), BorderLayout.CENTER);
    display1DArrays(); // Display initial array for
       triangle_sierpinski
} else {
    add(new JScrollPane(matrixPanel), BorderLayout.CENTER);
    matrixPanel.repaint(); // Ensure matrixPanel is displayed for
       other files
}
```

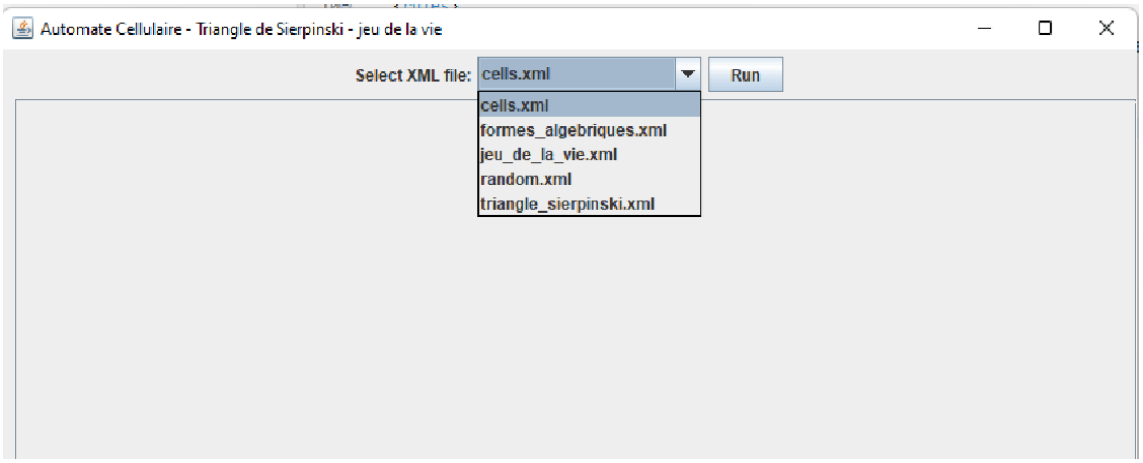
```
private void display1DArrays() {
    int[] dimensions = tableau.getDimensions();
    int length = dimensions[0];

    JPanel rowPanel = new JPanel(new GridLayout(1, length));
    for (int j = 0; j < length; j++) {
        JLabel cellLabel = new JLabel();
        cellLabel.setOpaque(true);
        cellLabel.setHorizontalAlignment(SwingConstants.CENTER);
        cellLabel.setBorder(BorderFactory.createLineBorder(Color.
           BLACK, 1)); // Set thinner border
        int[] indices = new int[]{j};
        int value = tableau.getValueAt(indices);
        cellLabel.setBackground(value == 1 ? Color.BLUE : Color.
           WHITE);
        rowPanel.add(cellLabel);
    }
    tableauPanel.add(rowPanel);

    tableauPanel.revalidate();
    tableauPanel.repaint();
}
```

12 Validation et Test

Cette section est dédiée à l'évaluation approfondie de notre jeu à travers une série de tests rigoureux et de cas d'usage. L'objectif est de démontrer non seulement la fiabilité et la robustesse de notre solution, mais aussi sa pertinence et son efficacité face à des scénarios réels et variés.

12.1 Le Menu

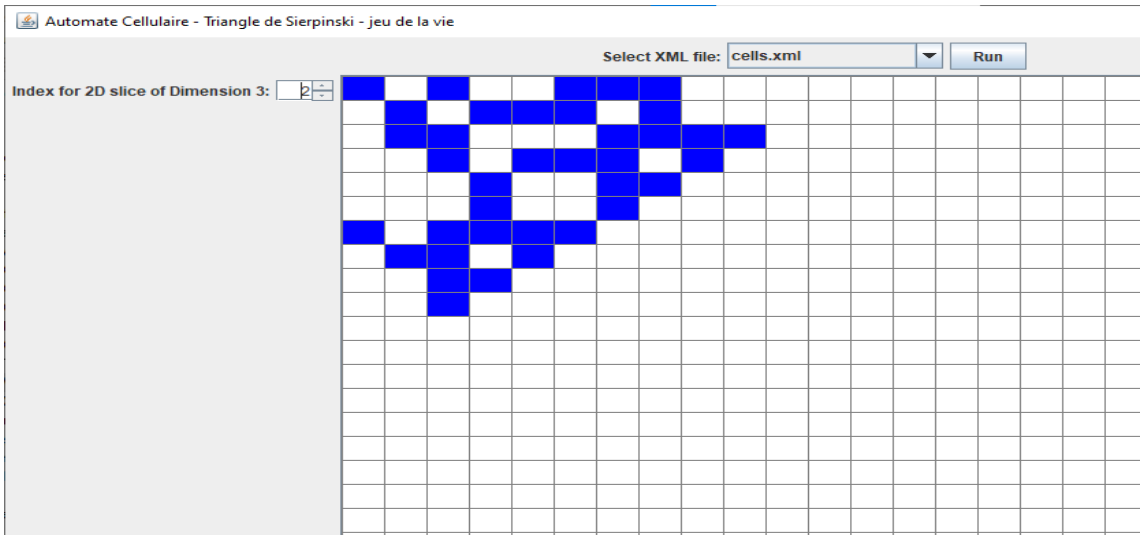
Ceci est le menu poposé lors de l'excécution du programme .



12.2 choix 1 : Cells

si on choisit Cells , le fichier xml qui se cache derière cette option est :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <grid>
        <dimensionality>3</dimensionality>
        <dimensions>30,30,30</dimensions> <!-- a 30x30x30 grid -->
        <initialState>
            <cell>1,1,1</cell>
            <cell>1,0,0</cell>
            <cell>0,0,0</cell>
            <cell>1,2,1</cell>
            <cell>2,2,2</cell>
            <cell>2,1,1</cell>
            <cell>2,0,0</cell>
        </initialState>
    </grid>
    <rules>
        <rule id="rule">
            SI(EQ(COMPTER(G6*),1),1,0)
        </rule>
    </rules>
</configuration>
```

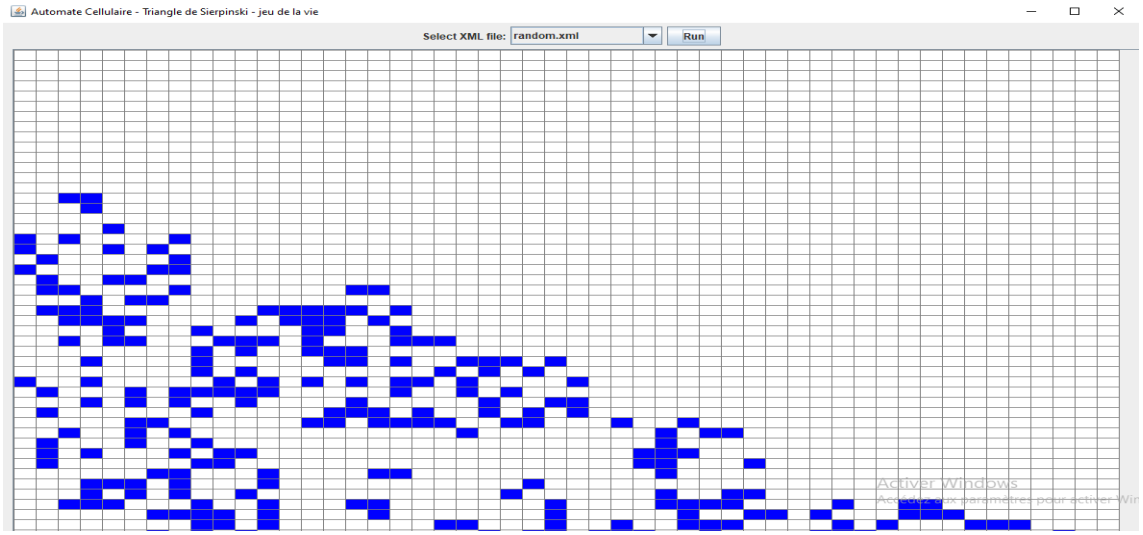


Puisque on est dans le contexte de plus que 2 d , on a le choix de choisir la coupe ou on désire afficher la grille à gauche en haut .
Les états s'affiche automatiquement les uns après l'autres en appliquant la règle .

12.3 Choix 2 : formes algebriques

Le fichier xml de cette option est :

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <grid>
        <dimensionality>2</dimensionality>
        <dimensions>101,101</dimensions> <!-- Corrected comment for
            a 30x30x30 grid -->
        <initialState>
            <!-- Correcting cells to include three coordinates for
               a 3D grid -->
            <cell>0,0</cell>
            <cell>0,100</cell>
            <cell>100,0</cell>
            <cell>100,100</cell>
            <cell>50,50</cell>

        </initialState>
    </grid>
    <rules>
        <rule id="rule">
           SI(EQ(COMPTER(G4), 1), 1, 0)
        </rule>
    </rules>
</configuration>
```



12.4 Choix 3 : Jeu de la vie

12.4.1 C'est quoi le Jeu de la vie ?

Le Jeu de la Vie, inventé par John Horton Conway en 1970, est un automate cellulaire sur une grille bidimensionnelle infinie de cellules carrées. Chaque cellule possède deux états possibles : vivante (1) ou morte (0). L'évolution de l'état des cellules au fil du temps se fait selon des règles simples, mais elles donnent lieu à des comportements complexes et souvent imprévisibles.

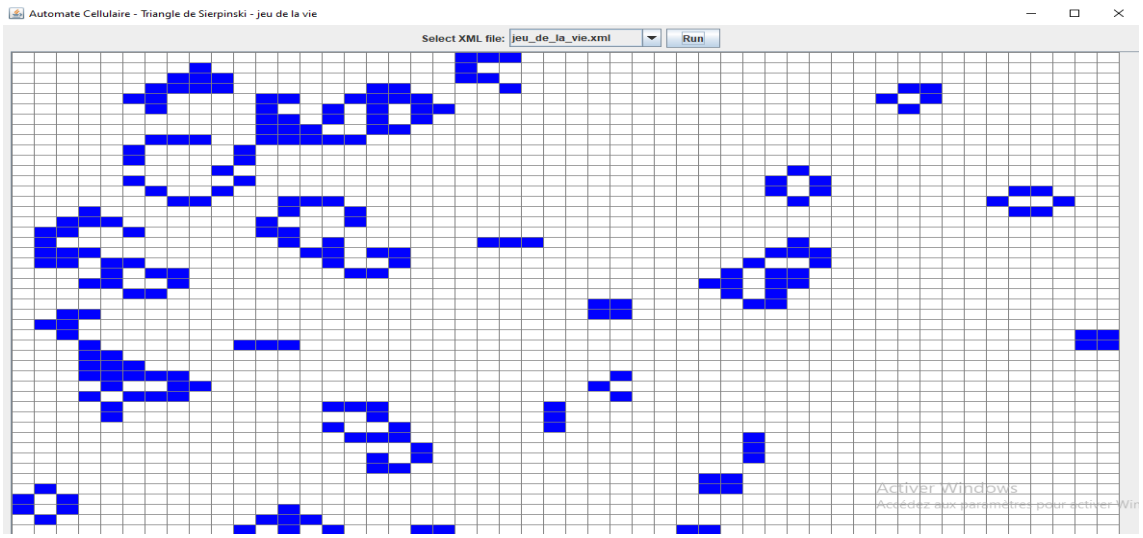
Dans le cadre de ce projet, le Jeu de la Vie est implémenté en utilisant un automate cellulaire bidimensionnel. La règle utilisée pour déterminer l'état des cellules est décrite par l'expression suivante :
SI (EQ (COMPTER (G0) ,1) , SI (OU (EQ (COMPTER (G8) ,2) , EQ (COMPTER (G8) ,3)) , 1 , 0) , SI (EQ (COMPTER (G8) ,3) , 1 , 0))

SI (EQ (COMPTER (G0) ,1) : Cette partie de la règle vérifie si la cellule elle-même est vivante (1).
SI (OU (EQ (COMPTER (G8) ,2) , EQ (COMPTER (G8) ,3)) , 1 , 0) : Si la cellule est vivante, elle reste vivante (1) si elle a exactement 2 ou 3 voisins vivants parmi les 8 voisins. Sinon, elle meurt (0).
SI (EQ (COMPTER (G8) ,3) , 1 , 0) : Si la cellule est morte (0), elle devient vivante (1) si elle a exactement 3 voisins vivants. Sinon, elle reste morte (0).

En résumé, la règle intégrée dans ce projet pour le Jeu de la Vie peut se lire de la manière suivante :
Si une cellule est vivante et a exactement 2 ou 3 voisins vivants, elle reste vivante. Sinon, elle meurt.
Si une cellule est morte et a exactement 3 voisins vivants, elle devient vivante. Sinon, elle reste morte. Le fichier xml derière cette exécution est :

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <grid>
```

```
        <dimensionality>2</dimensionality>
        <dimensions>50,50</dimensions>
        <initialState>
            <random>15</random> <!-- Initial cell at the center -->
        </initialState>
    </grid>
    <rules>
        <rule id="rule">
SI(EQ(COMPTER(G0),1),SI(OU(EQ(COMPTER(G8),2), EQ(COMPTER(G8),3)),
   1, 0),SI(EQ(COMPTER(G8),3), 1, 0))
        </rule>
    </rules>
</configuration>
```

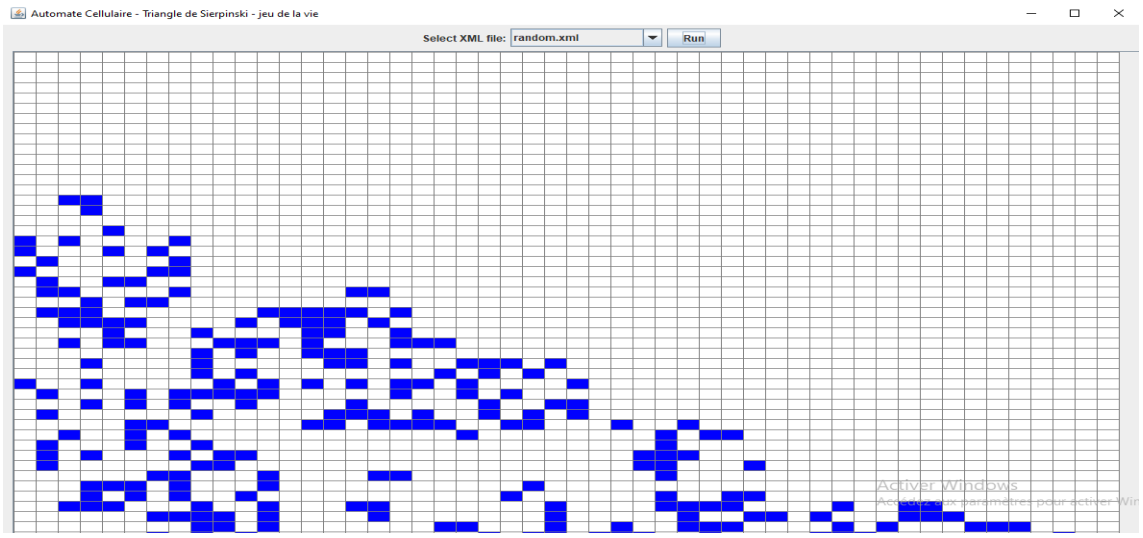


12.5 Choix 4 :

Le fichier xml de cette option est :

```
    <?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <grid>
        <dimensionality>2</dimensionality>
        <dimensions>50,50</dimensions>
        <initialState>
            <random>80</random>
        </initialState>
    </grid>
    <neighborhoods>
        <neighborhood >
            <offset dim0="0" dim1="-1"/>
        </neighborhood>
        <neighborhood >
            <offset dim0="0" dim1="1"/>
        </neighborhood>
```

```
    </neighborhoods>
    <rules>
        <rule id="rule">
SI(EQ(COMPTER(G8), 2), 1, 0)
        </rule>
    </rules>
</configuration>
```

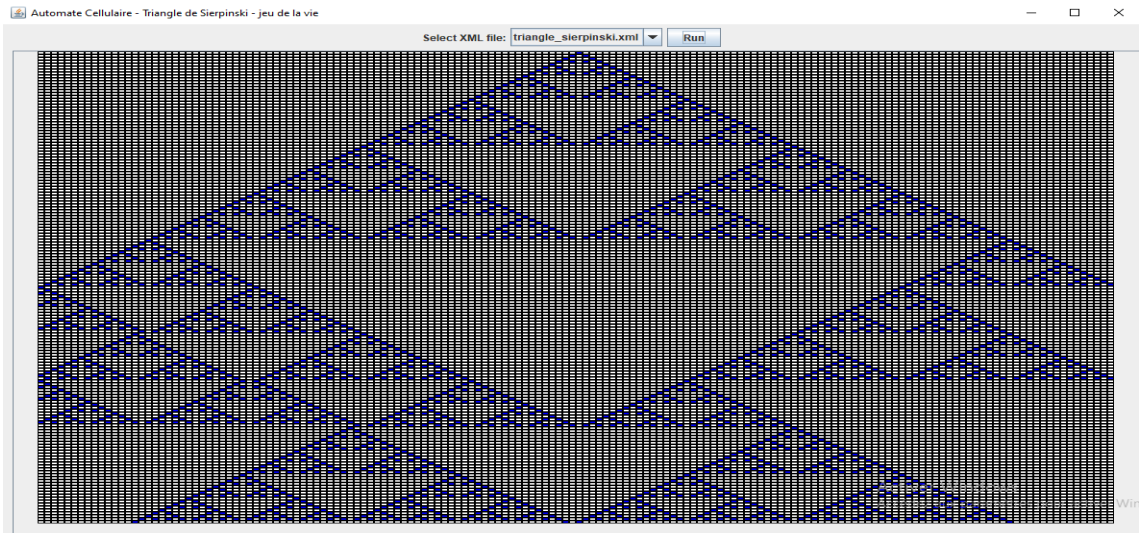


12.6 Choix 5 :

Le fichier xml du triangle de sierpinski est :

```
    <?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <grid>
        <dimensionality>1</dimensionality>
        <dimensions>160</dimensions>
        <initialState>
            <cell>80</cell> <!-- Initial cell at the center for
               better symmetry -->
        </initialState>
    </grid>

    <rules>
        <rule id="rule">
SI(EQ(COMPTER(G2),1),1,0)        </rule>
    </rules>
</configuration>
```

Ce qui diffère des autres exécutions est que ici , les états ne s'écrasent pas et ne passent pas inappercus mais il s'affichent en juxtaposition de tableaux 1d pour qu'on puisse avoir la forme de la triangle .


Automate Cellulaire - Triangle de Sierpinski - jeu de la vie
Select XML file:
triangle_sierpinski.xml
Run

13 Conclusion

En conclusion, notre exploration des automates cellulaires nous a conduit à l'intersection de la programmation avancée et des modèles de comportement émergents. Nous avons conçu des structures de données robustes et des règles précises pour simuler des comportements complexes tels que le Triangle de Sierpinski et le Jeu de la Vie de Conway. Ces simulations ont mis en lumière l'importance des opérateurs mathématiques et des définitions de voisinages pour la manipulation et l'évolution des cellules.
Notre approche méthodique a permis de gérer efficacement les données et les états des cellules, en utilisant des principes de la programmation orientée objet pour garantir extensibilité et flexibilité. Les règles et tableaux ont été soigneusement structurés pour permettre une manipulation fluide et une évolution dynamique des automates cellulaires.
Cette exploration a renforcé notre compréhension que la simulation de comportements cellulaires complexes n'est pas seulement une question de programmation, mais aussi un exercice intellectuel exigeant une profonde compréhension des interactions et des dynamiques cellulaires.
En définitive, ce projet a été une démonstration éloquente de la manière dont la technologie peut être utilisée pour modéliser des systèmes complexes, créant ainsi non seulement des simulations précises mais aussi des outils puissants pour l'étude des comportements émergents. Grâce à une conception réfléchie et à une mise en œuvre rigoureuse, nous avons réussi à créer une simulation robuste et adaptable des automates cellulaires, capable de représenter une multitude de scénarios fascinants.

14 Perspective future

14.1 Intégration de l'intelligence artificielle :

Combiner les automates cellulaires avec des algorithmes d'IA pour créer des systèmes capables d'apprendre et de s'adapter de manière autonome à leur environnement, augmentant ainsi leur utilité et leur efficacité.

14.2 Exploration de nouveaux comportements émergents :

Découvrir de nouveaux modèles de comportement en expérimentant avec différentes règles et configurations, ouvrant la voie à des recherches inédites.

14.3 Création d'outils pédagogiques :

Les automates cellulaires peuvent être utilisés comme outils éducatifs pour enseigner des concepts avancés de mathématiques, de programmation et de dynamique des systèmes, facilitant ainsi l'apprentissage interactif. Par exemple pour la Théorie des Graphes et Algèbre Linéaire , les automates cellulaires peuvent être utilisés pour démontrer des concepts de la théorie des graphes et de l'algèbre linéaire, comme les matrices d'adjacence, les chemins eulériens, ou les transformations linéaires.

Une matrice d'adjacence

Destinations

Origines	A	B	C
A	0	1	0
B	1	0	1
C	1	0	0

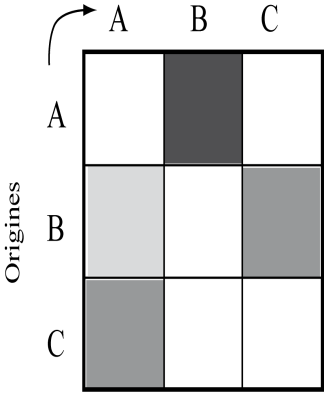
Une matrice de flux

Destinations

Origines	A	B	C
A	0	4	0
B	1	0	2
C	2	0	2

Une matrice-Bertin

Destinations



14.4 Plus de controle :

Ajouter des fonctionnalités de zoom et de pan dans l'interface utilisateur pour permettre une exploration détaillée des automates cellulaires.
Permettre aux utilisateurs de contrôler la vitesse de simulation pour une meilleure analyse des étapes spécifiques.