GANDER Félix - SCHWAIGER Sarah - Groupe TP 1
**Département Informatique Semestre 3**
**Janvier 2020**

IUT Robert Schuman
Institut universitaire de technologie
Université de Strasbourg

# A31 - Combats d'automates cellulaires

# Rapport

## I) Description de l'application

Notre application est un jeu de combat entre des automates cellulaires. Plusieurs joueurs s'affrontent sur une grille représentant la zone d'évolution de leurs automates respectifs.

Chaque automate est constitué de plusieurs cellules ayant chacune un état. Lors de chaque tour, les automates évoluent en fonction de l'état de chacune de leur cellules ainsi que de leur voisines. Si plusieurs automates se retrouvent à occuper une même cellule, alors l'un des joueur est désigné aléatoirement comme vainqueur et remporte cette dite cellule.

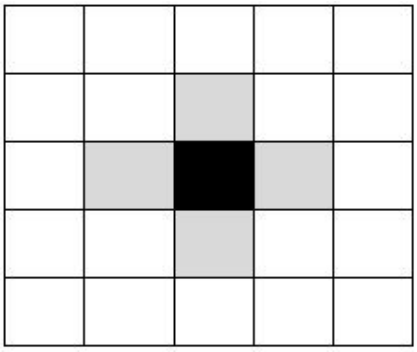
Le joueur ayant éliminé toute les cellules de ses adversaires gagne. Si aucun joueur n'as gagné au bout d'un certain nombre de tour, il y a match nul.

## II) Mode d'emploi

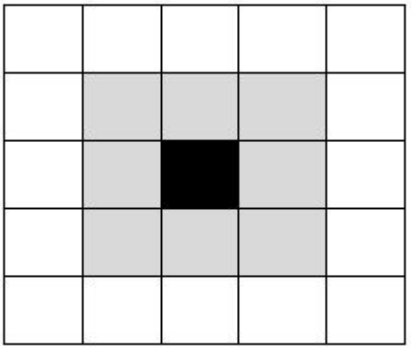
### Première phase :

Les joueurs se mettent d'accord sur le nombre de cellules qui composeront leur automate, la taille de la grille, sa méthode d'extension et le nombre maximal de tour de jeu. Puis, chacun choisit son automate. Il choisit une méthode de recherche du voisinage des cellules ainsi que les règles d'évolution de celles-ci. Les deux automates ne peuvent pas être identiques, ils doivent différés sur au moins l'une de leur deux caractéristiques.

Par exemple le voisinage de Von Neumann se présente ainsi :



Tandis que le voisinage de Moore1 peut être représenté comme tel :



Les règles d'évolutions elles, correspondent aux nombre de voisines d'une cellule ayant un certain état. Elles peuvent aussi dépendre de l'état actuel de la cellule elle même.

#### Deuxième phase:

A tour de rôle, les joueurs sélectionnent une cellule sur la grille afin de former leurs automates. Le joueur qui commence est tiré au sort, ainsi cette information se trouve au dessus de la grille. Cette phase s'arrête lorsque les 2 joueurs ont sélectionné le nombre de cellules défini lors de la première phase. Le bouton suivant au bas de la grille devient alors cliquable et permet de lancer la troisième phase.

#### Troisième phase:

Les automates évoluent sur la grille jusqu'à la victoire d'un joueur ou le nombre maximal d'itération. Les joueurs lancent le jeu en cliquant sur le bouton "start" au bas de la grille, celui-ci va mettre à jour la grille toute les 750 ms. Le nombre de tour restant est quand à lui affiché au dessus de la grille.

## III) Les patrons de conception mis en oeuvre

#### Fabrique paramétrée:

Nous avons décidé de mettre en place des fabriques paramétrées pour la création des automates et de la grille. En effet, la grille peut définir plusieurs méthodes d'extension différentes et chaque automate utilise des types de voisinage ainsi que des règles d'évolution différentes.

Ce patron nous permet donc de créer une grille et des automates dont on ne connaît pas concrètement le fonctionnement à l'avance. Il nous fournit ainsi une liaison simple entre l'interface où se font les choix des joueurs et nos classes métiers, ce qui nous permet de définir facilement quelles stratégies doivent utiliser les automates et la grille.

#### Stratégie:

L'utilisation du patron stratégie nous sert à résoudre le même problème cité dans la section précédente, c'est pourquoi nous le couplons au patron de fabrique paramétrée. En effet, la grille et les automates possèdent différents modes de fonctionnement, leurs opérations doivent donc être facilement modifiables. Les méthodes sont définis par les joueurs en débuts de partie et ces dernières leur sont proposés à l'aide d'énumérations.

De plus, l'application doit permettre facilement l'ajout de nouvelles méthodes d'extension pour la grille ainsi que de nouvelles règles d'évolution et de recherche du voisinage pour les automates. Le patron stratégie nous permet donc de répondre à cette demande en limitant le nombre de classes qui nécessitent des modifications.

#### Façade:

Nous avons utilisé un patron façade dans notre package controller pour la classe ControlFacade afin de centraliser les fonctionnalités de notre application. Cela nous permet aussi de bien différencier la vue du reste de l'application et ainsi d'avoir un faible couplage.

### Singleton:

Nous avons utilisé un patron singleton dans notre package controller pour les classes FabricAutomate et FabricGrid afin de ne posséder que une seule instance de ces classes qui sont accessibles facilement via la méthode getInstance(). Ces classes sont alors responsables de la création de leurs propres instances uniques.

## IV) Comment étendre l'application ?

### Nouveaux automates à 2 états:

Pour ajouter à l'application un nouvel automate à deux états, il faut ajouter à l'énumération RulesMethod qui se trouve dans le package controller le nom du nouvel automate. Puis, dans la fabrique paramétrée FabricAutomate, ajouter un cas au switch qui porte sur les règles d'évolution de la méthode createAutomate() afin de pouvoir créer l'automate en question.

Ensuite, créer une nouvelle classe héritant de l'interface Rule dans le package model. Cette classe doit porter le nouveau nom définit dans l'énumération et devra redéfinir la méthode next(). Dans cette méthode, définir les règles d'évolution souhaitées pour les cellules de l'automate.

Dans la vue Settings, ce choix se répercute sans modification, car une boucle parcourt tous les éléments de l'énumération RulesMethod et les ajoute un à un dans la combobox de choix des règles pour chacun des joueurs.

### Nouvelles méthodes d'extension de la grille:

Pour ajouter à l'application une nouvelle méthode d'extension pour la grille, il faut ajouter à l'énumération GridExtend qui se trouve dans le package controller le nom de la nouvelle méthode d'extension. Puis, dans la fabrique paramétrée FabricGrid, ajouter un cas au switch de la méthode createGrid() afin de pouvoir créer la grille en question.

Ensuite, créer une nouvelle classe héritant de l'interface Extension dans le package model. Cette classe doit porter le nouveau nom définit dans l'énumération et devra redéfinir la méthode addExtension(). Dans cette méthode, définir les règles de définition du voisinage pour les cellules se trouvant aux extrémités de la grille.

Dans la vue Settings, ce choix se répercute sans modification, car une boucle parcourt tous les éléments de l'énumération GridExtend et les ajoute un à un dans la combobox permettant de choisir la méthode d'extension.

### Nouveaux voisinages:

Pour ajouter à l'application un nouveau type de voisinage, il faut ajouter à l'énumération NeighborMethod qui se trouve dans le package controller le nom du nouveau type de voisinage. Puis, dans la fabrique paramétrée FabricAutomate, ajouter un cas au switch qui porte sur le voisinage de la méthode createAutomate() afin de pouvoir créer l'automate en question.

Ensuite, créer une nouvelle classe héritant de l'interface Neighbourhood dans le package model. Cette classe doit porter le nouveau nom définit dans l'énumération et devra

redéfinir la méthode getNeighbor(). Dans cette méthode, définir les règles le champ des recherche des cellules voisines souhaitées pour les cellules de l'automate.

Dans la vue Settings, ce choix se répercute sans modification, car une boucle parcourt tous les éléments de l'énumération NeighborMethod et les ajoute un à un dans la combobox de choix de la recherche des voisins pour chacun des joueurs.

### Automates à plus de deux états :

Pour ajouter à l'application un nouvel état aux automates, il faut ajouter à l'énumération State qui se trouve dans le package model le nom du nouvel état. Puis, dans toutes les classes héritant de l'interface Rule, redéfinir la méthode next() pour faire correspondre les règles d'évolution des cellules en fonction des états possibles. Par la suite il faudra modifier la méthode fight() de la classe Automate afin de définir les nouvelles règles des combats selon les différents états possibles.

Pour ce qui est du package view, dans Game, il faudra modifier la méthode updateGrid() afin de rajouter des couleurs spécifiques au nouvel état pour chaque joueur. De même, on pourra être amené à modifier les couleurs de la méthode ActionListener() des boutons de la grille dans le constructeur de Game(), si l'on souhaite modifier l'état que prenne les cellules lors de l'initialisation, afin de correspondre à celles définies dans updateGrid().

Enfin, on modifiera la méthode endGame() de notre ControlFacade dans le package controller. En effet, les conditions de victoire nécessiteront certainement d'être revues en fonction du nouvel état.