RAPPORT PROJET LO21

AUTOMATES CELLULAIRES - P18

Romain Peres
朱孟申 (Zhu Mengshen)
Victor Ronfaut

Table des matières

I - Description du contexte

a) Qu'est ce qu'un automate cellulaire ?

Représentation

Un automate cellulaire est représenté par une grille qui comporte un nombre fini de cases qu'on appellera des "cellules", qui sont chacunes d'entres-elles dotées d'une valeur ; le cardinal que peut prendre chaque valeur appartient à l'intervalle [2 , n] (avec n un nombre fini supérieur ou égal à 2).
L'ensemble de cette grille sera nommé "état". Ainsi, l'ensemble des cellules de la grille représentent un état.

Principe

Le principe d'un automate cellulaire est tout simplement de faire varier son état, par l'intermédiaire de règles prédéfinies à l'avance ; ces règles sont nommées "règles d'évolution".
En fonction du nombre de fois qu'on appliquera ces même "règles d'évolution" à un état, celui-ci nous génèrera un nouvel état ; On comptera donc autant de "générations" d'états que le nombre de fois où l'on aura appliquer les règles d'évolution à notre état. A la génération 0, l'état sera l'état initial.

La valeur de chaque cellule d'un état est fondamentale car elle va nous permettre de déterminer les valeurs des autres cellules à l'état suivant.

Ainsi pour faire varier un état "E", il suffit de faire varier les valeurs des cellules de la grille à l'état "E+1".

La question qui vient ensuite est : *comment faire varier la valeur de ces cellules ?*

Précédemment, nous avons parlé de "règle d'évolution".
En effet, c'est à partir de la règle d'évolution que nous pouvons faire varier la valeur d'une cellule, en prenant en compte son voisinage.
Le voisinage d'une cellule "X" est représenté par l'ensemble des cellules qui entourent "X".

<u>Par exemple :</u> dans un automate à une dimension, représenté par une grille d'une seule ligne, le voisinage de la cellule numéroté 1 sera l'ensemble des cellules numérotées 0,1 et 2.
Lorsque nous avons le voisinage de la cellule "X" à l'état "E"; les valeurs des cellules du voisinage vont nous permettre d'obtenir la valeur de la cellule "X" à l'état "E+1", en appliquant la règle d'évolution.

En quoi cette "règle d'évolution" va nous permettre de changer les valeurs des cellules à la génération suivante ?

La règle d'évolution est en réalité une règle qui définit en fonction du voisinage d'une cellule "X", la valeur de cette cellule "X" à la génération d'après.

En effet, une sorte de table va enregistrer toutes les configurations de voisinages possibles au sein de l'automate cellulaire.
Cette table va permettre de décider arbitrairement la valeur que va prendre une cellule à la génération d'après, grâce à l'utilisateur qui va choisir une règle précise : la "règle d'évolution".

Nous allons illustrer notre propos par 3 exemples qui sont manipulable dans notre application.

b) Explications des automates cellulaires de notre application

L'automate cellulaire à 1 dimension et 2 valeurs de cellules possibles

Cet automate cellulaire a été manipulé lors des séances de TDs 4 et 5.
Chaque cellule de la grille peut prendre comme valeur 1 ou 0, on compte donc 8 configurations possible de voisinage représentées ci-dessous :
(le voisinage de chaque cellule sera sa cellule de gauche, elle même et sa cellule de droite)

111	110	101	100	011	010	001	000

En fonction de ces 8 voisinages différents, il faut définir quelle sera la valeur, à la génération suivante, d'une cellule pour chaque configurations.
Cela sera donné par la règle d'évolution que l'utilisateur aura choisie.
Dans notre cas précis, il existe 256 règles (de 0 à 255), car il y a 256 façon de faire ce choix.

Ainsi, considérons l'automate cellulaire qui suit, défini par la table suivante,
qui donne la règle "0 0 0 1 1 1 1 0" :

Configuration initiale à l'instant t	111	110	101	100	011	010	001	000
Valeur suivante de la cellule centrale à l'instant $t+1$	0	0	0	1	1	1	1	0

Cela signifie que si une cellule "X" a pour valeur "1" et que sa cellule de droite et sa cellule de gauche ont une valeur de "1" alors cette cellule "X" aura pour valeur "0" à la génération suivante.
En effet on se trouve dans le cas d'une configuration de voisinage de "1 1 1" qui donne à la cellule "X" la valeur "0", dû à la règle choisie par l'utilisateur = "0 0 0 1 1 1 1 0".

Cet automate a été implémenté dans notre application de telle sorte qu'il puisse être possible à l'utilisateur de choisir le nombre de cellule de la grille, ainsi qu'un état de départ.
Pour le faire tourner, il n'aura plus qu'à renseigner la "régle d'évolution" qu'il souhaite lui appliquer

parmi les 256 règles possibles.

L'automate cellulaire à 2 dimensions et 2 valeurs de cellules possibles

Ce type d'automate est représenté par une grille à 2 dimensions, on pourrait l'assimiler à une matrice (N,M) avec N et M choisi par l'utilisateur.
Le fait d'être en 2 dimensions va impacter sur le voisinage de chaque cellule.
Précédemment, nous avons définis ce "voisinage" comme étant l'ensemble des cellules entourant la cellule ciblée.

Ainsi, sur une grille à 2 dimensions, la cellule ciblée ne possède plus 2 cellules voisines mais 8 :

Voisine	Voisine	Voisine
Voisine	X	Voisine
Voisine	Voisine	Voisine

Cela va nous forcer à revoir la "règle d'évolution" qui permet de passer d'une génération d'état à la suivante.

Vous l'aurez compris, dans ce cas de figure nous ne pouvons pas représenter toutes les configurations de voisinages possible dans une table sur 8 bits (table de taille 8).
Il nous sera donc impossible de faire tourner cet automate cellulaire à l'aide d'une seule règle d'évolution passé par l'utilisateur.

Néanmoins, on s'est rendu compte que 2 règles suffisaient pour le faire tourner.
Ainsi, cet automate cellulaire ne possède pas 1 table comme le précédent, mais 2 tables :

- Table 1 : nombre de cellule voisines vivantes (pour une cellule vivante) [entre 0 et 8]
- Table 2 : nombre de cellule voisines vivantes (pour une cellule morte) [entre 0 et 8]

Ainsi, ces 2 tables vont nous permettre de prévoir toutes les configurations possibles de voisinages.

Pour mieux comprendre le principe de ce type d'automate cellulaire nous allons prendre comme exemple celui du "jeu de la vie", que nous devions implémenté dans le cadre de notre projet.

Jeu de la vie

Le “jeu de la vie” créé par John Horton Conway en 1970, est sans doute l’automate cellulaire le plus connu de tous.
Ce “jeu de la vie” est en réalité un cas particulier des automates cellulaires à 2 dimensions et 2 valeurs de cellules possibles, puisqu’il se déroule sur une grille à 2 dimensions.

On se trouve dans ce cas, lorsque la règle d’évolution correspond à la sienne, c’est à dire :

- Une cellule morte possédant exactement 3 cellules voisines vivantes devient vivante, sinon reste morte.
- Une cellule vivante possédant 2 ou 3 voisines vivantes le reste, sinon elle meurt.

(cellule considérée comme “vivante” : sa valeur = 1)
(cellule considérée comme “morte” : sa valeur = 0)

Grâce à cet exemple on se rend compte que pour pouvoir manipuler un automate de ce type il va falloir considérer 2 paramètres important :

1. La valeur de la cellule, à l’instant t
2. Le nombre de cellules vivante, à l’instant t. (Le voisinage de la cellule ciblée)

Le sujet a énoncé qu’on doit au moins :

— Définir le **nombre min** de **voisins vivants** au **temps t** pour qu’**une cellule soit vivante** au **temps t + 1** ;
— Définir le **nombre max** de **voisins vivants** au **temps t** pour qu’**une cellule soit vivante** au **temps t + 1** ;

Pour simplifier et augmenter notre performance, de tel sorte qu’on puisse faire tourner tous les automates à 2 dimensions avec la même implémentation, notre application est capable de :

— Définir le **nombre exact** de **voisins vivants,** au **temps t,** pour qu’**une cellule vivante,** au **temps t,** devienne **vivante ou morte, au temps t + 1** : Table 1 + règle pour cellules vivantes.
— Définir le **nombre exact** de **voisins vivants** au **temps t** pour qu’**une cellule morte, au temps t,** devienne **vivante ou morte, au temps t + 1** : Table 2 + règle pour cellules mortes.

En voici un exemple précis, représentant le “jeu de la vie” :

Pour une cellule vivante au temps t :

Nombre de voisins vivants au temps t	8	7	6	5	4	3	2	1	0
Valeur de cette cellule au temps t+1	0	0	0	0	0	1	1	0	0

Résultat :
Si une cellule vivante (“X” = 1), au temps t, a 2 ou 3 voisines vivantes, elle reste vivante au temps t+1 (“X” -> 1), sinon elle meurt(“X” -> 0).
Ici on vient d’utiliser la règle pour cellules vivantes :
En binaire on peut représenter cette règle comme «0 0 0 0 0 1 1 0 0», et en décimal «24».

Pour une cellule morte au temps t :

Nombre de voisins vivant au temps t	8	7	6	5	4	3	2	1	0
Valeur de cette cellule au temps t+1	0	0	0	0	0	1	0	0	0

Résultat :
Si une cellule morte (“X” = 0), au temps t, a 3 voisines vivantes, elle devient vivante au temps t+1 (“X” -> 1), sinon elle reste morte(“X” -> 0).
Ici, on vient d’utiliser la règle pour cellules mortes :
En binaire on peut représenter cette règle comme «0 0 0 0 0 0 1 0 0», et en décimal «8».

On peut voir qu’ici on utilise une chaîne de 9 chiffres binaires, et elle peut être transférer en décimal de 0-511.
Donc on a 512 possibilités pour une règle d’une cellule vivante, et 512 possibilités pour une règle d’une cellule morte.
On les met dans un même automate : “AutomateDeuxDimension”, donc il y a au total 512*512 possibilités pour un automate à 2 dimensions d’évoluer !

Cet automate a été implémenté dans notre application de telle sorte qu’il puisse être possible à l’utilisateur de choisir **le nombre de cellule de la grille**, en choisissant les dimensions de la grille (hauteur et largeur), ainsi qu’un **état de départ**.
Pour le faire tourner, il n’aura plus qu’à renseigner 2 “règles d’évolution” qu’il souhaite lui appliquer parmi les 512*512 possibilités.

L’automate cellulaire à 1 dimension et 99 valeurs de cellules possibles

Enfin, cet automate est le dernier que nous avons décidé d'implémenter afin de pouvoir manipuler un automate cellulaire avec plus de 2 valeurs possibles pour chaque cellule. Il nous permet, dans un second temps, de montrer que notre application n'est pas remise en cause par l'intégration d'un nouveau type d'automate cellulaire.

Nous avons donc dû imaginer un système permettant à l'utilisateur d'entrer une règle d'évolution, afin de déterminer les valeurs des cellules de la génération suivante, comme dans les 2 automates précédents.

Pour ce faire, nous avons repris une règle d'évolution sur 8 bits, à l'instar de l'automate 1 dimension, car le contexte s'y prêtait : le voisinage d'une cellule de ce type d'automate est représenté par sa cellule de gauche et sa cellule de droite (comme pour les automates à 1 dimension).

Néanmoins, nous avons dû repenser à chaque configuration de voisinage possible, car le cardinal de la valeur de chaque cellule n'est plus de 2 mais d'un nombre entier choisi par l'utilisateurs (On peut donc avoir une cellule valant 98 et une autre valant 13 etc ...).

En ne se focalisant plus sur la valeur de chaque cellule, mais plutôt sur la parité des valeurs de chaque cellule, nous avons pu facilement déduire cette table de voisinage :

I I I	P I I	I P I	I I P	P P I	P I P	I P P	PPP

I : cellule de valeur Impaire
P : cellule de valeur Paire

Ainsi, la règle d'évolution passé par l'utilisateur (sur 8 bits), va nous permettre de déterminer la valeur de chaque cellule à la génération suivante de cette façon :

- Si la règle d'évolution indique 1 pour un voisinage "V" (avec "V" = {"X-1", "X", "X+1"}), alors la valeur (VA) de la cellule "X" à la génération suivante est :

  VA("X") = VA("X") + [VA("X-1") + VA("X+1")]
- Si la règle d'évolution indique 0 pour un voisinage "V" (avec "V" = {"X-1", "X", "X+1"}), alors la valeur (VA) de la cellule "X" à la génération suivante est :

  VA("X") = VA("X") - [VA("X-1") + VA("X+1")]

Ainsi, considérons l'automate cellulaire qui suit, défini par la table suivante,
qui donne la règle "0 0 1 1 0 1 0 1" :

I I I	P I I	I P I	I I P	P P I	P I P	I P P	PPP
0	0	1	1	0	1	0	1

Résultat :
Si une cellule “X” a pour valeur “15” et que sa cellule de droite et sa cellule de gauche ont une valeur de “32” alors cette cellule “X” aura pour valeur “79” à la génération suivante.
En effet on se trouve dans le cas d’une configuration de voisinage de “P I P” qui donne à la cellule “X” la valeur “15 + 32 + 32” = “79”*, dû à la règle choisie par l’utilisateur (“P I P” -> 1).

*l’utilisateur est censé déterminer une valeur minimum et une valeur maximum que les cellules peuvent prendre : si la valeur de l’opération effectué pour déterminer la valeur de la cellule de la génération suivante n’est pas dans l’intervalle [Valeur_min ; Valeur_max], l’algorithme de la fonction setCellule (permettant d’attribuer une valeur à une cellule) fonctionnera comme une fonction modulo, ainsi la valeur qu’il affectera à la cellule de la génération suivante se trouvera forcément dans cet intervalle.

Cet automate a été implémenté dans notre application de telle sorte qu’il puisse être possible à l’utilisateur de choisir le nombre de cellule de la grille, ainsi qu’un état de départ, et une valeur minimum et maximum que peut prendre chaque cellule de la grille. Pour le faire tourner, il n’aura plus qu’à renseigner la “règle d’évolution” qu’il souhaite lui appliquer parmi les 256 règles possibles.

II - Description de l'architecture :

Pour pouvoir implémenter notre code afin que l'application fonctionne, nous avons dû dans un premier temps structurer notre projet en passant par l'étape de conceptualisation.

Lors du TD 4, nous avions déjà pensé à une conceptualisation en UML pour les automates cellulaires à 1 dimension avec 2 valeurs possibles pour chaque cellule. Celui-ci mettait en exergue 4 classes :

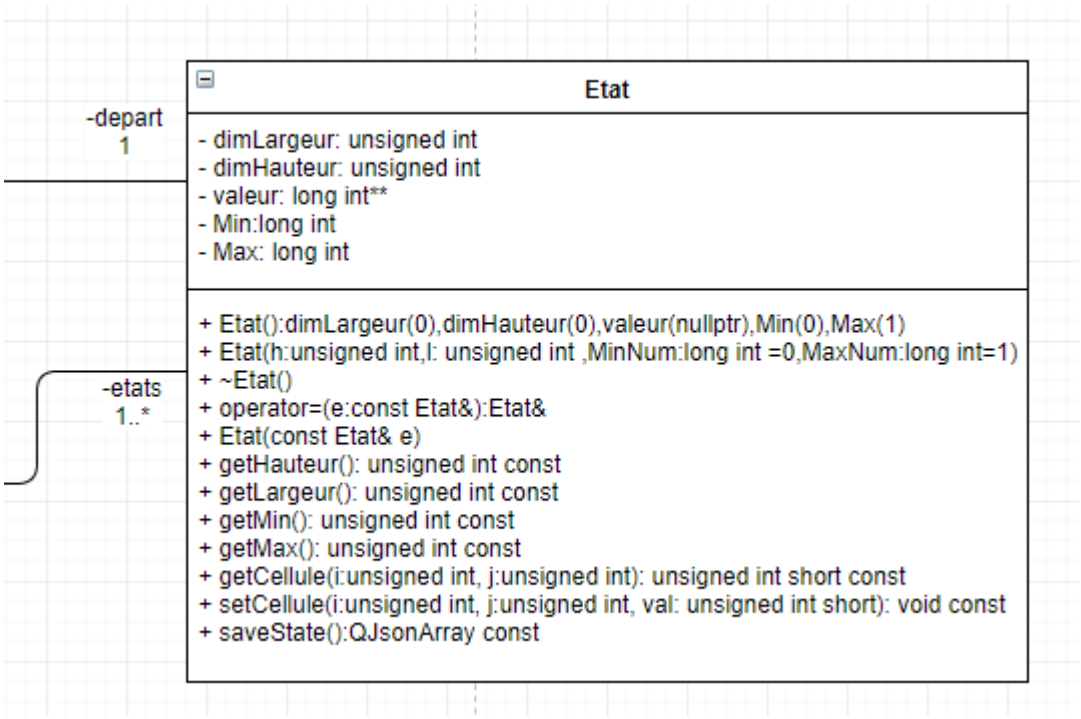
- La classe "Etat"
- La classe "Automate"
- La classe "Simulateur"
- La classe "AutomateManager"

Auquel nous avons rajouté la classe "AutoCell"

La conceptualisation de notre UML pour ce projet, s'en est beaucoup inspiré, en y ajoutant quelques autres classes. Le voici en Annexe (annexe 1).

Comme vous avez pu le remarquer, nous avons gardé la structure proposé dans le td 4.
Néanmoins nous avons dû y apporter quelque modification, à commencer par la classe Etat.

<u>La classe Etat :</u>



Cette classe nous permet de représenter la grille de l'automate cellulaire comme un tableau de 1 dimension (si attribut dimHauteur = 1), ou de 2 dimensions (si dimHauteur > 1).

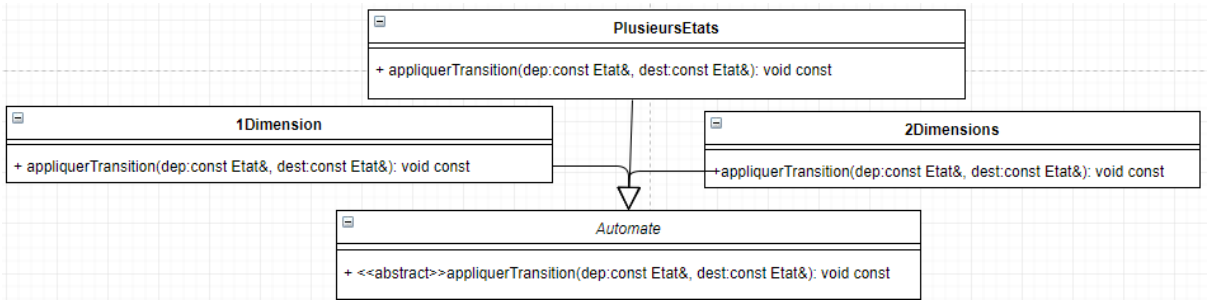
Cette classe est donc conçue pour satisfaire tous les besoins des tableaux 1 dimension et 2 dimensions, qui seront en réalité les grilles des automates 1 dimension et 2 dimensions.

Les attributs **Min et Max** sont créés pour satisfaire le besoin des nouveaux automates cellulaires qui ont plusieurs états.
Ils ont pour rôle de vérifier que les valeurs dans le tableau **valeur** sont bien compris dans l'intervalle [Min ; Max].
Par défaut, Min = 0 et Max = 1, pour satisfaire le besoin de Automate 1 ou 2 dimensions, qui n'ont besoin que de 2 valeurs possibles pour chaque cellules (0 et 1).

Si on entre une valeur qui n'est pas dans l'intervalle, on réaliser une opération de modulo (si le nombre est supérieur à la borne Max de l'intervalle) ou plusieurs soustractions (lorsque le nombre est inférieur à la borne Min de l'intervalle)

P.S. Une architecture idéal aurait été de concevoir 3 types d'états différents (donc 3 classes d'états) pour les 3 types d'automates cellulaires différents. Mais jusque là nous n'avons pas pu améliorer notre code, car les TDs nous permettant d'être plus aidé pour le faire arrivèrent trop tard dans le semestre.

La classe Automate :

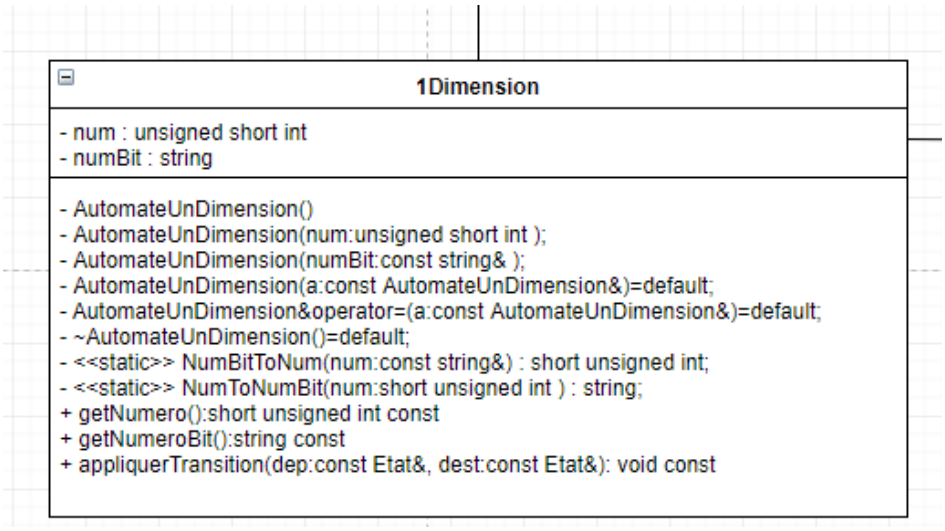


Nous avons pris la décision de transformer cette classe en une classe abstraite et la faire hériter par les 3 autres classes d'automates (que nous verrons juste après) qui représenteront chacunes d'elles un type d'automate.

Automate ne comporte seulement qu'une seule méthode virtuelle pure : appliquerTransition, qui va être définie dans ses 3 classes filles : cette méthode permet de transformer un état en son état suivant (de passer de génération en génération).

Comme le montre le schéma ci-dessus, la classe Automate est la classe mère de 3 sous classes, à commencer par :

La classe AutomateUnDimension :

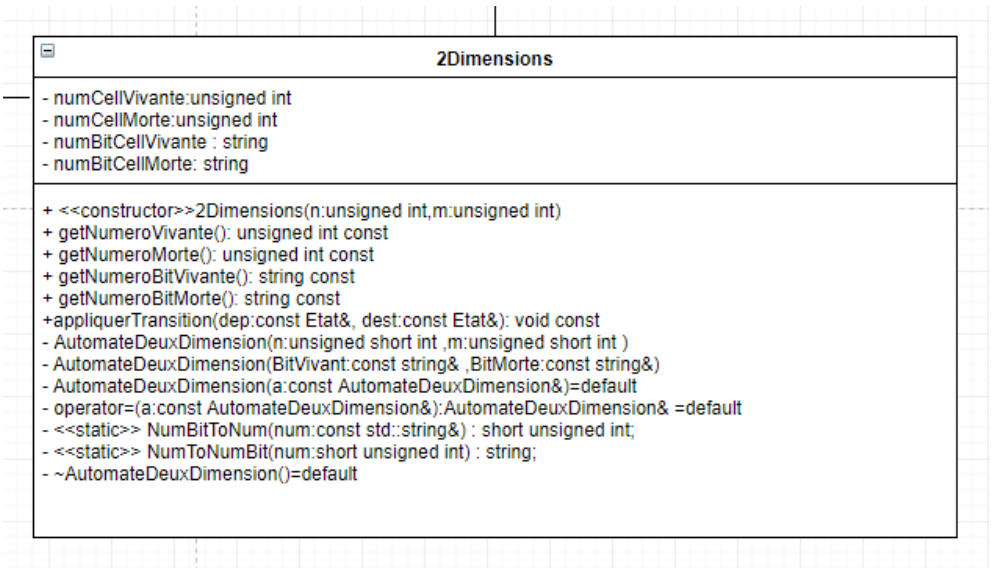


Cette classe nous permet de représenter l'automate cellulaire d'une dimension. Elle comporte un attribut num qui est en réalité la règle d'évolution pour cet automate, et un attribut numBit qui représente cette même règle en bits.

C'est la méthode appliquerTransition qui va faire intervenir l'attribut numBit(donc la règle d'évolution) pour passer d'un état (dep) à son état suivant (dest).

La deuxième classe fille d'Automate est :

La classe AutomateDeuxDimensions:



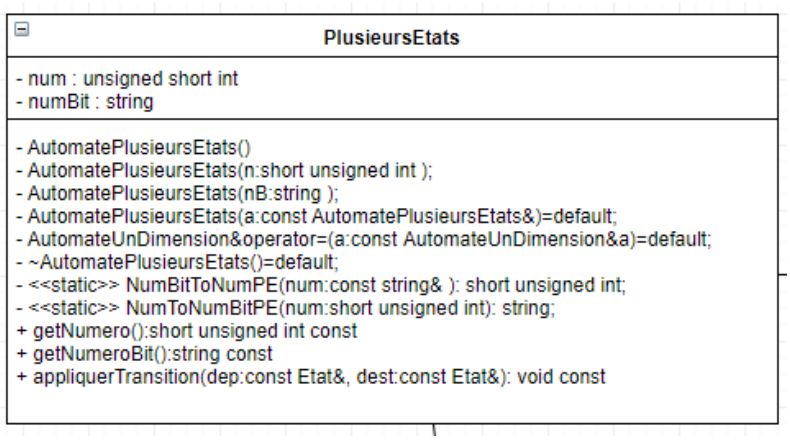
AutomateDeuxDimensions est la classe représentant l'automate cellulaire de 2 dimensions. La classe comporte les 2 règles mentionnées dans la partie I. Cela justifie la présence des 4 attributs présents dans cette classe :

- numCellVivante et numBitCellVivante déterminent la même règle (l'une en bit l'autre en hexdecimal) qui pour chaque cellule vivante va déterminer si la cellule continue de vivre ou pas.
- Pareil pour numCellMorte et numBitCelluleMorte, qui pour chaque cellule morte va déterminetr si la cellule reste morte ou devient vivante.

Ces 2 règles vont s'appliquer dans la dans la méthode appliquerTransition.

Enfin, la dernière classe fille d'Automate est :

<u>La classe AutomatePlusieursEtats :</u>



Enfin, la classe AutomatePlusieursEtats ressemble de près à la classe AutomateUnDimension. La seule réelle différence se trouve dans la méthode appliquerTransition, qui doit permettre à une cellule de changer de valeur en fonction du voisinage (comme expliqué précédemment)

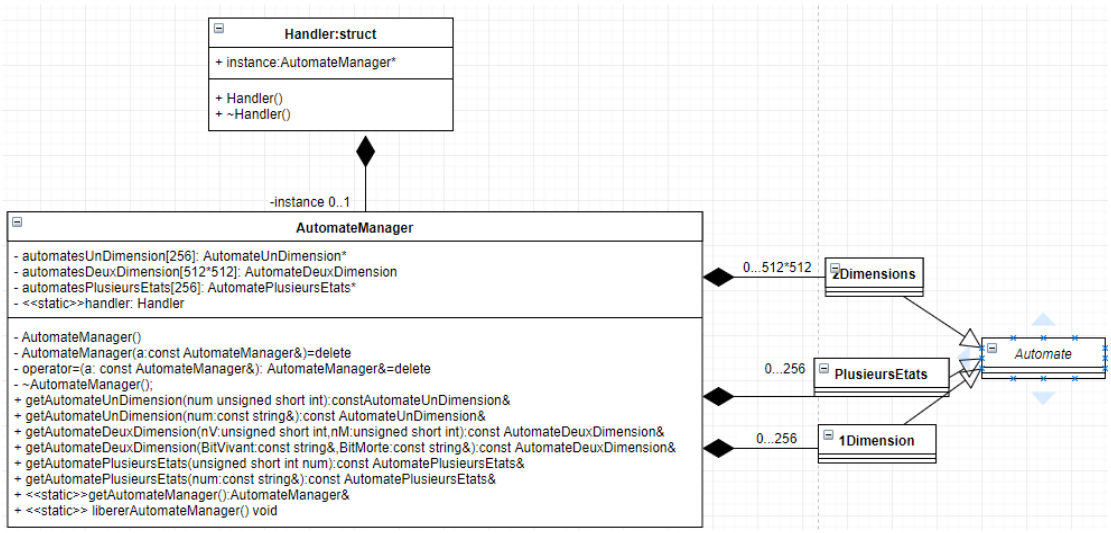
Pour ces 3 classes filles que l'on vient de décrire

On met les fonctions de transitions NumBitToNum() et NumToNumBit() dans les trois sous-classes d'automates séparément, sans passer par la classe mère, car il y a 8 chiffres binaires pour la chaîne de caractères de règle d'AutomateUnDimension et AutomatePlusieursEtats, mais il y en a 9 pour AutomateDeuxDimensions : le format des règles suivant les 3 classes-filles n'est pas le même. De plus, on les déclare comme fonction statique car on doit les utiliser dans le cas où il n'est pas nécessaire d'instancier un automate.

Enfin, on place tous les constructeurs et destructeurs en privé pour enlever la possibilité de construire un automate à l'utilisateur directement, par duplication ou par affectation. Ainsi, on laisse la classe AutomateManager contrôler la fabrication et la suppression en utilisant le design pattern

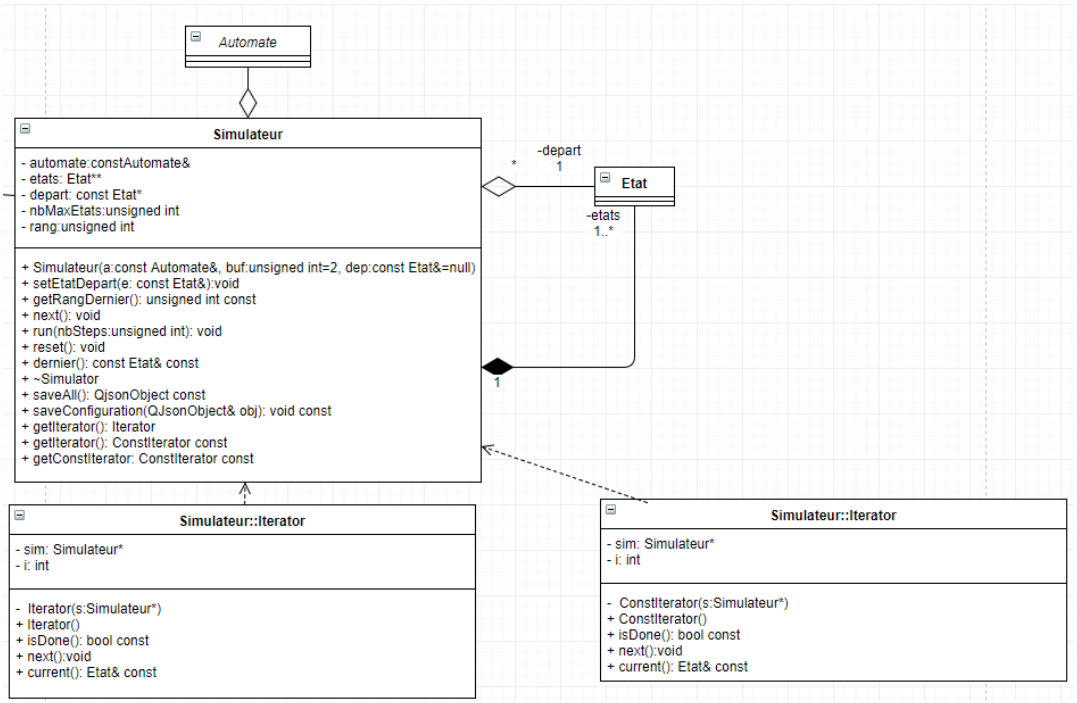
Singleton.

La classe AutomateManager :



C'est la classe qui est le créateur, conteneur et destructeur des instances des classe filles d'Automate. C'est une application de design pattern Singleton.
Il comporte 4 attributs : 3 attributs sont les tableaux qui contiennent toutes les règles de AutomateUnDimension, AutomateDeuxDimensions et AutomatePlusieursEtats.
Il garantit l'unicité de l'instance de chaque règle de ces trois types dans le système.
De plus, la structure Handler garantit qu'il y a seulement une instance de classe AutomateManager dans le système.

La classe Simulateur :



Simulateur est la classe qui permet de simuler l'enchaînement des générations d'automates cellulaires de tous types avec une règle d'évolution et un état qu'on a choisi comme état de départ. Cette classe comporte un buffer qui permet de garder les n derniers états générés (n étant la taille du buffer, précisé par l'utilisateur), et l'état de départ qu'on a choisi préalablement.
L'état de départ est créé par une instance de classe Etat existant, et les états dans le buffer sont créés par Simulateur lui-même avec la fonction `appliquerTransition`.
Elle possède aussi une référence qui pointe sur un automate qui est créé par AutomateManager : ce sera la règle d'évolution.

Enfin, elle comporte aussi un itérateur et un itérateur const qui permet de parcourir les derniers états générés depuis le dernier généré jusqu'au plus ancien encore présent dans le buffer.

La classe AutoCellApp

Pour finir, AutoCellApp est la classe interface de l'application. Contenant un simulateur, c'est elle qui crée la fenêtre de l'application, héritant de la classe QMainWindow (Qt) elle possède nombreux widgets et conteneurs afin de mettre en place l'interface, faisant lien entre l'utilisateur et le programme derrière.

III - L’évolution de l’architecture

La facilité d’évolution des architectures est sûrement l’un des points les plus importants en informatique de nos jours étant donnée que les applications ne sont plus fixes dans le temps et ont pour but d’être constamment revisité et réétudier pour les améliorer, y ajouter de nouvelles fonctionnalités. C’est aussi une chose très difficile que de rendre une application facilement réutilisable, remodelable, simplement et efficacement. Cela demande beaucoup de réflexion et d’anticipation. Malheureusement c’est peut-être là où notre implémentation a pêché.

En effet, Qt est un module du C++ qui nous était peu familier au commencement du projet, et malgré une introduction claire dans les cours, il y avait des choses que l’on ne pouvait pas prévoir.

Néanmoins, notre architecture nous permet une facilité d’ajout de nouveaux automates cellulaires. Grâce à l’héritage sur la classe Automate, nous donnons aux développeurs d'opportunités d’ajouter simplement une classe hérité, de créer propre leur règle et de régler la transition d’état à un autre facilement avec la virtualité de la méthode `appliquerTransition` dans la classe mère. En suivant cette façon de faire, on donne au programmeur l’entière responsabilité de son automate. Le simulateur lui ne s’occupe que d’appliquer la méthode virtuelle pour que la magie s’opère.

D’un autre côté, le fait que l’automate se gère tout seul a posé soucis notamment sur la partie sauvegarde des objets où il a fallu savoir quel automate était de quel type pour savoir s’il fallait sauvegarder une ou deux règles et de quelle taille. Cette partie auquelle nous n’avions pas pensé nous a porté problème et nous avons dû nous abstenir de faire cela de façon évolutive par manque de temps.

Ceci étant dit, comme on peut le voir pour l’automate plusieurs états, nous avons choisis de représenter les états sous forme de nombre plutôt que de créer des couleurs pour plusieurs raisons, notamment celle du fait que si nos états sont trop nombreux, il nous est impossible de voir la différence entre un état et un autre. Ceci permet notamment l’implémentation future et la bonne prise en charge d’autres automates à plusieurs états. La contrepartie est que sur la visibilité, ça devient vite lourd à regarder puisqu’on ne peut pas remarquer instantanément la ressemblance entre plusieurs états.

Du côté du simulateur, l’avantage de notre architecture est que le simulateur applique les fonctions sans se soucier derrière des types d’automate ou d’état. Un avantage majeur lorsque l’on comprend que ça signifie que peu importe si notre état doit être à plusieurs dimensions, si notre automate gère une ou deux dimensions, tout ça est inscrit dès le départ dans le constructeur et le reste se fait simplement.

Toujours sur cette partie là, à la différence du TD nous avons opté pour un tableau d’entier dans notre classe État pour la même raison d’évolution. Cela nous permet de pouvoir avoir des automates à nombre d’états grand sans avoir à revoir notre entière architecture.

Enfin, malgré qu’ici nous n’avons pas implémenter de fonctionnalité sur le choix de l’état initial, il serait peu complexe de réaliser plusieurs type de choix comme la symétrie lors du choix de l’état initial. Étant donnée que ce choix se fait dans une tout autre fenêtre que la fenêtre principal, il suffirait d’implémenter un widget pour choisir l’option de remplissage et le tour est joué.

Conclusion

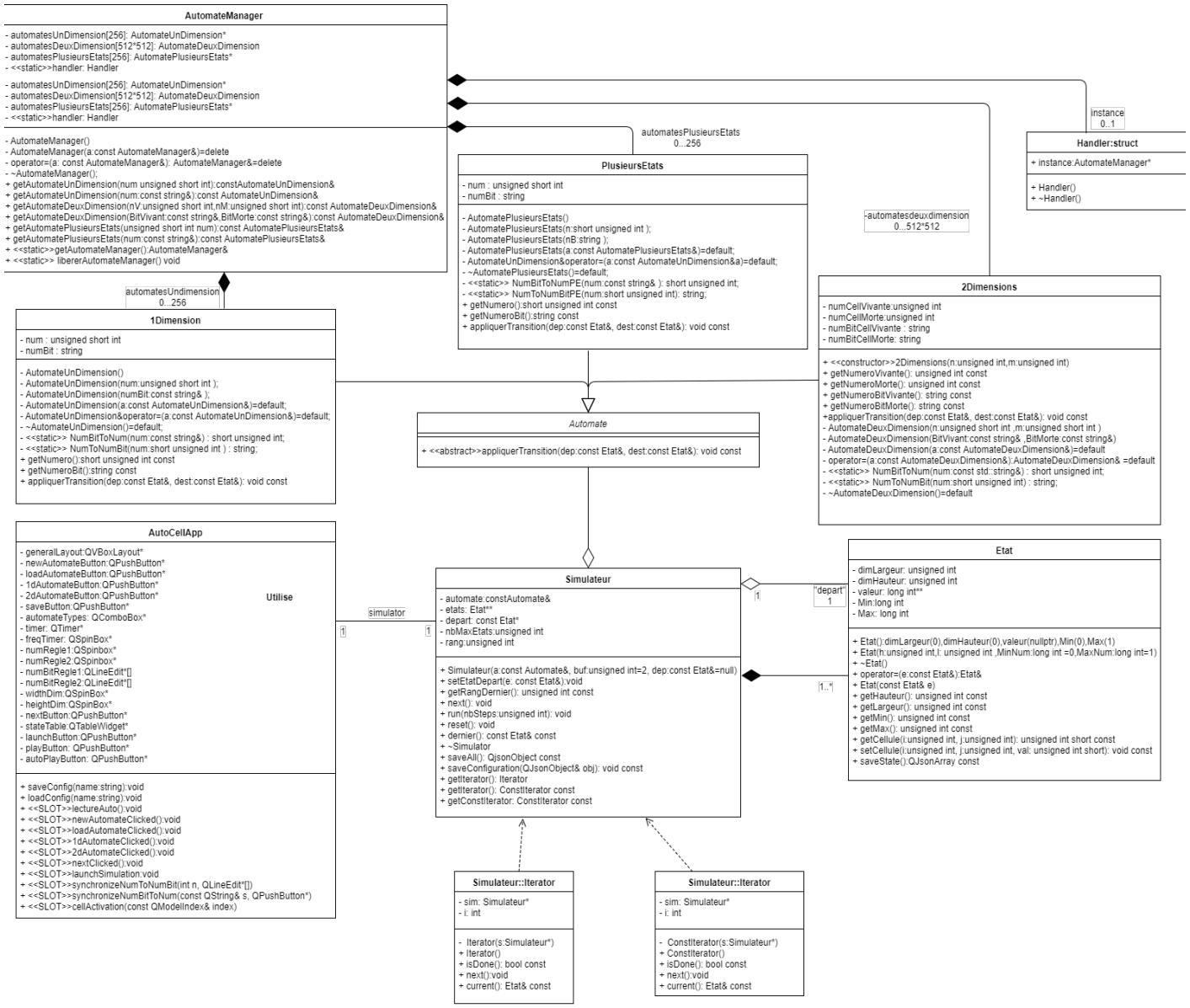
Ce projet a été pour tout le groupe un moment enrichissant qui nous a permis de pouvoir comprendre ce que représente le travail d'équipe, mais aussi de grandes valeurs de programmation orientée objet comme la réflexion de la conception avant implémentation et l'idée de devoir penser notre application non pas dans le présent mais aussi dans le futur en mettant tout en oeuvre pour que celle-ci soit évolutive. La réalisation de ce projet nous a aussi permis de réaliser les défis que représente le travail entre personnes aux cultures différentes.

Il n'y a pas de doute quant à l'imperfection de ce projet, mais nous avons su voir nos erreurs, que nous aurions corrigés si le temps ne nous manquait pas. C'est cette imperfection, et ce manque de temps qui nous ont appris ce que représente un projet, et nous ont fait comprendre qu'il faut dès le début penser à la fin du projet et non pas seulement aux premiers obstacles de la longue route qui nous amène à la version finale.

Mais ce projet a aussi été intéressant de sa conception à son implémentation, mais en dehors, socialement aussi. Travailler en groupe sur un petit projet a été sans aucun doute une drôle mais agréable aventure.

Annexe

Annexe 1 - UML



Annexe 2 - Vidéo de présentation de l'application

Lien vers la vidéo : https://www.youtube.com/watch?v=CmWb5iOJM18&feature=youtu.be

Annexe 3 - Instruction de compilation

Pour la compilation de ce projet, nous avons utiliser le compilateur Qt5.10.1 MinGW 32 bits ainsi que les dernières version de C++. Veillez bien à ce que votre version de Qt ne soit pas trop ancienne. Le compilateur peut aussi impacter l'exécution. Chez certains membres de notre projet, la compilation du code avait du mal à se réaliser à cause de ça.