LO21 - Programmation et conception orientées objet
Antoine Jouglet

Université de technologie de Compiègne

Projet LO21 : Autocell

Loïc Adam - Amaury Gelin - Wael Hamdan

17 juin 2018

Table des matières

Introduction

La Programmation Orientée Objet est un paradigme de programmation fondamental en Informatique. Il s'agit de définir des classes d'objets qui représentent des concepts ou des entités. Ces objets possèdent une structure interne composée d'attributs, ainsi qu'un comportement détaillé par des méthodes. Le rassemblement d'attributs et de méthodes au sein d'une même classe est réalisé par un procédé d'abstraction. Par ailleurs, les objets interagissent entre eux à travers des liens qui les lient. Ces interactions permettent de concevoir les fonctionnalités de notre logiciel. La POO, si elle est bien mise en œuvre, nous offre les possibilités de concevoir du code modulaire et générique. Ces qualités sont fondamentales en Génie Logiciel, car elles permettent de rendre nos programmes extensibles, réutilisables et maintenables, et donc de diminuer drastiquement les coûts qui en résultent.
Dans le cadre de notre cours sur la Conception Orientée Objet orchestré par M. Jouglet et ses collègues, il nous a été demandé de réaliser une application gérant des automates cellulaires. L'objectif est de mettre en œuvre les différents concepts de l'orienté objet pour pouvoir produire un programme modulaire, générique et facilement extensible. Ce sujet est intéressant et académique, car il nous fait nous confronter à de nombreuses problématiques de conception. Comme nous le verrons par la suite, les possibilités concernant les automates cellulaires sont quasiment infinies, donc réaliser un programme voulant représenter un bon nombre de possibilités n'est pas une tâche triviale.
Le langage utilisé est le C++ puisque c'est le langage que nous avons utilisé lors de ce cours. Qt est utilisé pour réaliser l'interface graphique.
Ce rapport est composé de trois parties principales. Tout d'abord, nous présenterons le contexte de l'application en définissant notamment la notion d'automate cellulaire. Puis, nous nous attarderons sur le point essentiel du rapport : l'architecture de notre application. Enfin, nous expliquerons en quoi cette architecture permet de nombreuses évolutions futures.

1 Automate cellulaire

Dans cette première partie est introduit le principe d'un automate cellulaire tout en détaillant, assez succinctement, ceux qui sont actuellement présents dans l'application Autocell. Les informations sont basées sur un article d'Encyclopædia Universalis [1] et sur un article de l'encyclopédie collaborative Wikipédia [2].

1. https ://www.universalis.fr/encyclopedie/automates-cellulaires/
2. https ://en.wikipedia.org/wiki/Cellular_automaton

1.1 Principe d'un automate

1.1.1 Historique et utilisations

Les automates cellules ont été découverts durant les années 1940 par le mathématicien américano-polonais Stanislaw Ulam (1909-1984) et par le mathématicien et physicien américano-hongrois John von Neumann (1903-1957). Le but de ces automates cellulaires était de modéliser des phénomènes dynamiques comme la génération dynamique de graphiques à partir de règles simples pour le premier scientifique, ou la modélisation de l'autoreproduction dans la nature pour le deuxième.
Il faut néanmoins attendre les travaux du mathématicien anglais John Horton Conway (1937) dans les années 1970 avec son fameux jeu de la vie, et ensuite la popularisation des automates unidimensionnels une décennie plus tard par le scientifique anglais Stephen Wolfram (1959), pour que les automates deviennent un sujet reconnu et très étudié.
Depuis, les automates cellulaires sont utilisés dans de nombreuses disciplines, dont en voici trois exemples :

- En biologie, les automates cellulaires sont utilisés pour représenter certaines régularités dans la nature. Ainsi, la respiration des plantes à partir de leurs stomates est modélisable par des automates cellulaires.
- En informatique, la règle 30 des automates élémentaires (qui permet aussi de modéliser l'apparence de certains coquillages) a été proposée comme une méthode de chiffrement possible.
- En physique, il est possible à partir d'un automate en deux dimensions de simuler des feux de forêts en prenant en compte l'humidité ou la force du vent.

1.1.2 Définition et fonctionnement

Un automate cellulaire est une grille de cellules dont voici les propriétés générales :

- La grille est d'un nombre de dimensions fini. Dans l'application AutoCell seuls les automates de dimension 1 et 2 sont considérés.
- Chaque cellule est dans un état particulier. Le nombre d'états est lui aussi fini.
- Chaque cellule possède un voisinage, c'est-à-dire un groupe de cellules qui influence sur son évolution future.
- Un automate évolue au cours du temps. Une grille initiale est donnée au début ($t = 0$) et à chaque unité de temps ($t = t + 1$) les états de

chaque cellule sont recalculés. Pour cela, chaque automate possède des règles de transition qui, pour toute cellule à l'instant t, déterminent son état à l'instant $t+1$ à partir de l'état de chaque cellule à l'instant t de son voisinage.

Comme il est impossible d'énumérer tous les automates possibles puisqu'il en existe une infinité, ce rapport ne présentera que les automates existants dans le logiciel. Il existe tout une théorie sur les automates, mais nous n'aborderons pas ce point, car ce n'est pas le but de ce rapport.

1.2 Automate élémentaire

Les automates élémentaires sont des automates à une dimension avec seulement deux états : 0 et 1. Le voisinage de chaque cellule est constitué des deux cellules qui l'entourent.
Il est possible d'énumérer toutes les règles de transition possibles :

- Chaque triplet de cellules (la cellule et son voisinage) ne peut être que dans l'un de ces 8 états : 111, 110, 101, 100, 011, 010, 001, 000.
- Dans chaque cas, l'état suivant de la cellule ne pourra qu'être 1 ou 2.
- Au final, nous avons $2^8 = 256$ règles possibles.

Une règle d'un automate cellulaire se présente alors sous cette forme :

Cellule et son voisinage	111	110	101	100	011	010	001	000
État suivant de la cellule	0	1	1	0	1	0	1	0

TABLE 1 – Règles de transition possibles pour un automate élémentaire

Ici nous avons la règle 106 ($01101010_2 = 106_{10}$). Notre application peut gérer un automate élémentaire en spécifiant la règle et une grille initiale.

1.3 Automate du type jeu de la vie

Toutes les variantes du jeu de la vie, y compris le jeu de la vie lui-même, sont caractérisées de la sorte :

- La grille est en deux dimensions
- Il existe deux types d'états (à ne pas confondre avec deux états) : des états « morts » et des états « vivants ».
- Chaque cellule à un voisinage dit de Moore (cf le tableau 2) .

- L'état d'une cellule à l'instant $t+1$ est en fonction de son état à l'instant t et du nombre de cellules dans son voisinage dans un état « vivant » à l'instant t.

Voisin Nord-Ouest	Voisin Nord	Voisin Nord-Est
Voisin Ouest	Cellule considérée	Voisin Est
Voisin Sud-Ouest	Voisin Sud	Voisin Sud-Est

TABLE 2 – Voisinage de Moore

Le logiciel AUTOCELL est conçu pour faciliter l'intégration d'automates qui remplissent ces conditions. En plus du jeu de la vie, l'automate QuadLife, qui est une variante du jeu de la vie, est présent.

1.3.1 Jeu de la vie

Le jeu de la vie, contrairement à son nom, n'est pas exactement un jeu au sens large du terme, car il ne requiert aucun joueur et son évolution dépend seulement de son état initial (éventuellement inséré par un humain).
L'automate possède deux états : « mort (0) » et « vivant » (1) et quatre règles :

- Une cellule vivante avec moins de deux cellules vivantes dans son voisinage meurt à l'instant suivant.
- Une cellule vivante avec entre deux et trois cellules vivantes dans son voisinage survit à l'instant suivant.
- Une cellule vivante avec plus de trois cellules vivantes dans son voisinage meurt à l'instant suivant.
- Une cellule morte avec exactement trois cellules vivantes dans son voisinage devient vivante à l'instant suivant.

Le logiciel AUTOCELL permet de simuler un automate de la sorte à partir d'une grille initiale. Il est aussi possible de modifier le nombre de cellules minimum et maximum pour qu'une cellule vive à l'instant suivant et le nombre nécessaire pour qu'elle naisse à l'instant suivant.

1.3.2 QuadLife

QuadLife est une variante du jeu de la vie qui est implantée dans le logiciel AUTOCELL afin de montrer la modularité de ce dernier.
L'automate possède 5 états : un « mort » (0) et quatre « vivants » (1-2-3-4). Les règles de transition sont les quatre règles suivantes :

- Une cellule vivante avec moins de deux cellules vivantes dans son voisinage meurt à l'instant suivant.
- Une cellule vivante avec entre deux et trois cellules vivantes dans son voisinage survit à l'instant suivant.
- Une cellule vivante avec plus de trois cellules vivantes dans son voisinage meurt à l'instant suivant.
- Une cellule morte avec exactement trois cellules vivantes dans son voisinage devient vivante à l'instant suivant. L'état de la cellule est déterminé de la sorte : si les trois cellules du voisinage sont dans des états différents (par exemple 1-2-3), alors la cellule prend l'état vivant restant (par exemple 4). Sinon, elle prend l'état qui est le plus présent parmi ses trois voisins (par exemple elle prend l'état 2 si deux cellules ont l'état 2 et une l'état 1).

Cette fois-ci, l'application n'autorise pas de changer le nombre de cellules nécessaires pour vivre ou mourir.

2 Architecture

Dans cette deuxième partie, nous allons présenter l'architecture de notre application. Pour ce faire, nous présenterons notre diagramme de classes en UML, nous expliciterons nos choix de conception et nous justifierons l'emploi de certains patrons de conception.

2.1 Diagramme de classes

La réalisation d'une application utilisant les concepts de l'orienté objet n'est pas une tâche facile. En effet, il est nécessaire de bien réfléchir à l'architecture de l'application avant même d'écrire une quelconque ligne de code. Cette étape se nomme la modélisation. C'est une étape fondamentale, car un bon modèle facilitera grandement l'implémentation des différentes fonctionnalités attendues, et nous permettra de produire du code maintenable, réutilisable et extensible.
La figure 1 présente le diagramme de classes que nous avons élaboré (en utilisant le langage de modélisation standard UML). Notez qu'il ne s'agit que d'un diagramme simplifié puisque nous ne représentons pas les attributs et les méthodes des différentes classes. En effet, cela alourdirait considérablement le diagramme et ne permettrait donc pas une bonne lecture globale. Cependant, nous expliciterons les détails nécessaires dans la suite du rapport. Un fichier contenant le diagramme complet est également consultable.

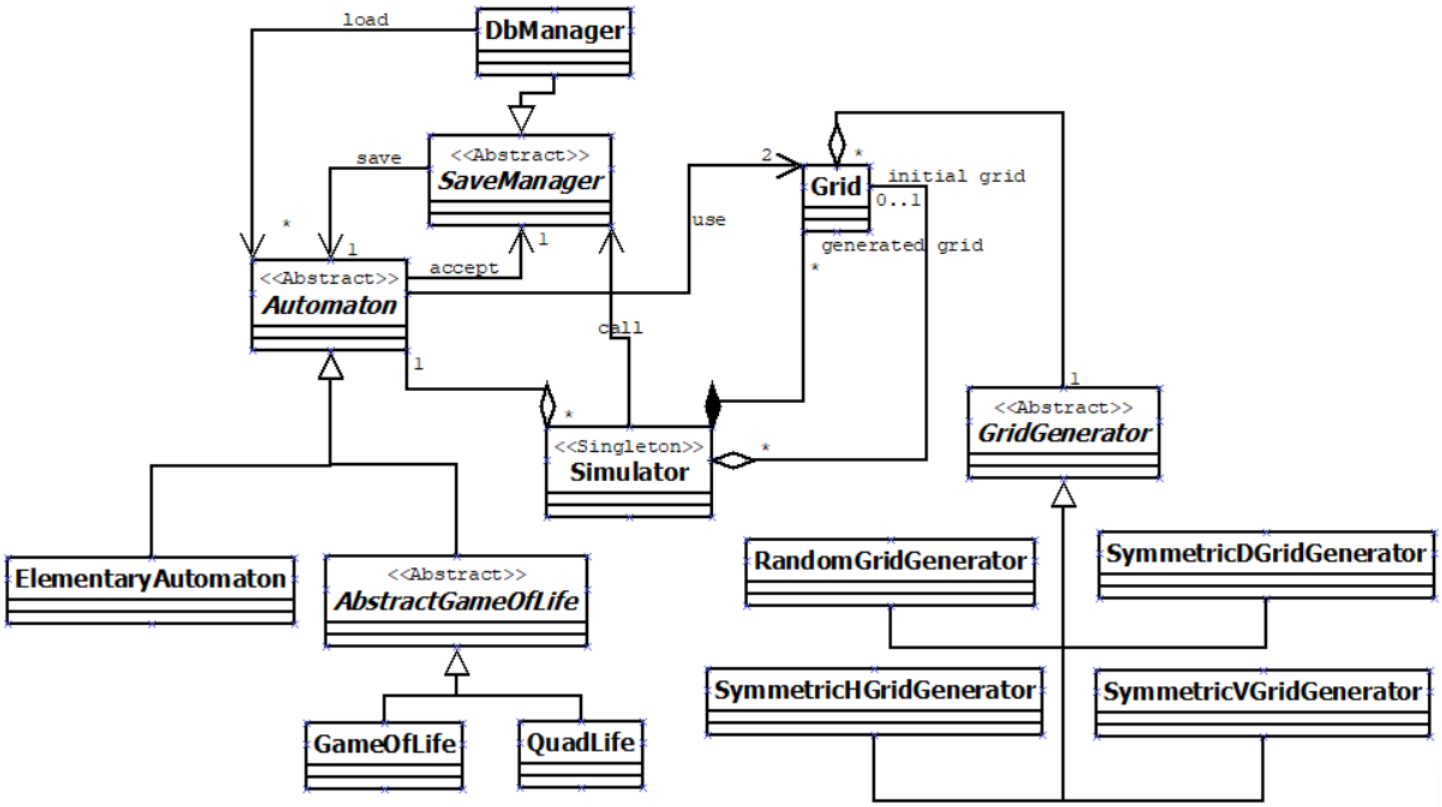

FIGURE 1 – Diagramme de classes simplifié en UML

2.2 Automates

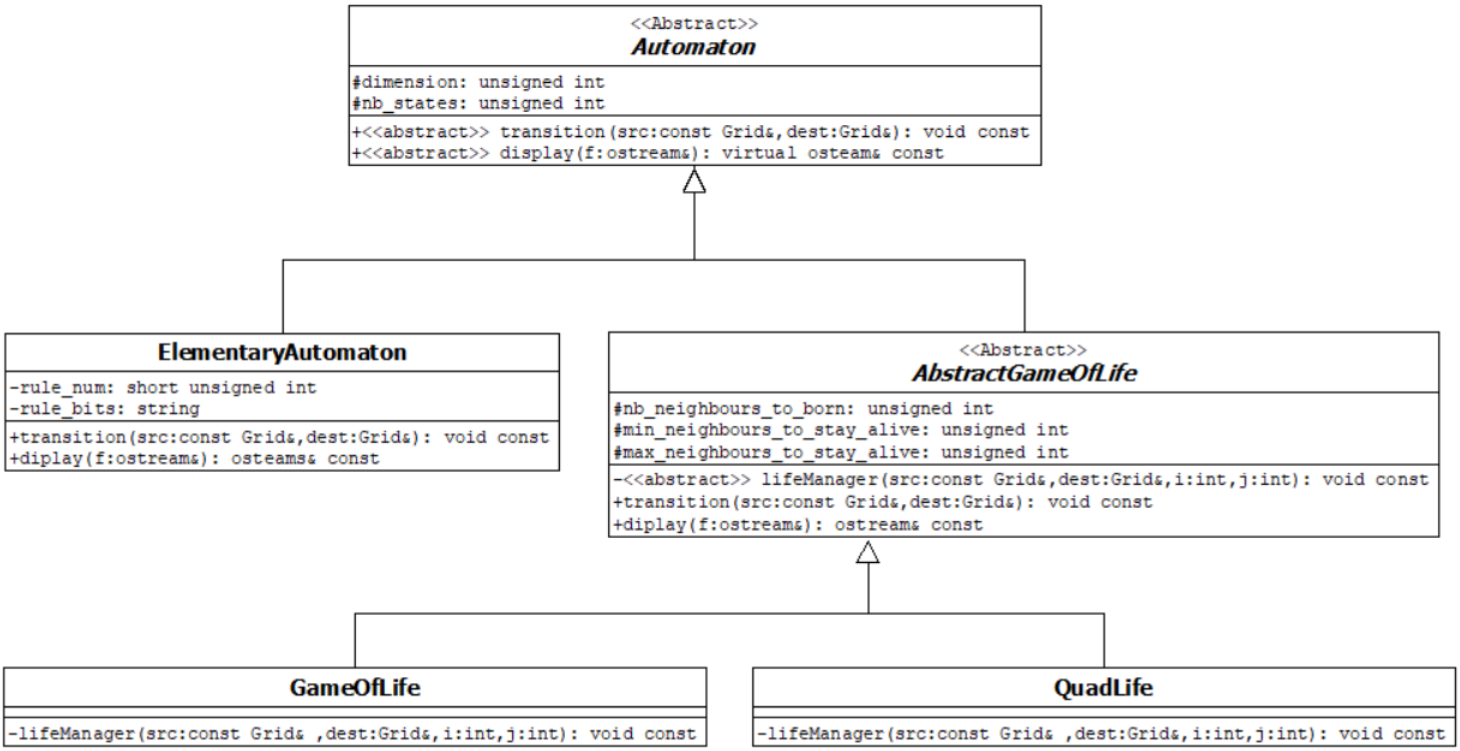

FIGURE 2 – Coupe du diagramme de classes axée sur les automates.

Remarque : seuls les attributs et certaines méthodes importantes pour nos explications ont été représentés.

Tous les types d'automates possèdent certaines caractéristiques communes. Ils possèdent notamment une dimension (limitée à 1 ou 2 dans cette application) et un nombre d'états. Ils ont également des comportements communs, comme la possibilité de réaliser des transitions sur une grille d'états, ou la possibilité de s'afficher (affichage des attributs). Cependant, des automates particuliers possèdent des caractéristiques spécifiques, en plus de celles communes. Cela nous a naturellement conduit à réaliser une spécialisation de la classe **Automaton** à travers de l'héritage.
De plus, on constate que chaque type d'automate a sa propre règle de transition, représentée par une méthode *transition()*. Celle-ci n'est à priori pas définissable depuis la classe Automaton, ce qui nous amène à en faire une méthode virtuelle pure et donc la classe Automaton devient abstraite. Par cette méthode virtuelle pure, nous instaurons du polymorphisme, ce qui nous

permet de respecter le principe de substitution : il doit être possible de substituer une instance d'une super-classe par une instance de n'importe laquelle de ses classes dérivées tout en gardant un comportement adéquat. La méthode *transition()* doit être définie dans les classes dérivées pour qu'elles soient concrètes et instanciables. En 3^{e} type d'automate, nous avons souhaité implémenter l'automate **QuadLife**. Puisqu'il s'agit d'une variante du jeu de la vie classique, nous avons constaté qu'il partageait un bon nombre de caractéristiques avec ce dernier. Ainsi, nous avons inséré un autre niveau d'héritage avec une nouvelle classe abstraite dont héritent **GameOfLife** et QuadLife : **AbstractGameOfLife**. La règle de transition entre GameOfLife et QuadLife est très similaire, mis à part pour la naissance des cellules. De ce fait, nous avons défini *transition()* dans AbstractGameOfLife, et nous avons instauré une nouvelle méthode virtuelle pure *lifeManager()* qui est appelée dans *transition()* en utilisant le patron de conception **patron de méthode**. Celle-ci est redéfinie dans GameOfLife et QuadLife.
En ce qui concerne l'affichage, l'affichage des caractéristiques générales d'un automate est délégué à Automaton avec une méthode virtuelle (et même virtuelle pure puisque nous souhaitons forcer un utilisateur-programmeur à redéfinir cette méthode dans de futures classes dérivées). Cette méthode est donc redéfinie dans **ElementaryAutomaton** et AbstractGameOfLife, où elle appelle en premier lieu la méthode générale avant de faire l'affichage spécifique à chaque automate. Pour un affichage simplifié et dans les normes, il est préférable de redéfinir l'opérateur de flux de sortie «. Ainsi, nous l'avons redéfini pour le plus haut niveau de hiérarchie (Automaton) en appelant la méthode *display()* de manière polymorphique avec un **patron de méthode** (puisque *display()* est abstraite à ce niveau de hiérarchie, et ce même si elle possède une définition). Chaque spécialisation d'automate « est un » automate (héritage public) donc l'utilisation de l'opérateur « qui prend en paramètre une référence d'automate est toujours possible. Le polymorphisme sur *display()* fait ensuite son œuvre pour nous assurer un comportement cohérent et ainsi respecter le principe de substitution.

2.3 Représentation des cellules

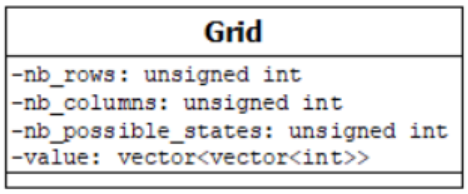

FIGURE 3 – Coupe du diagramme de classes axée sur la grille.

Nous avons fait le choix de ne pas créer de classe pour représenter une cellule. Selon notre modélisation, une cellule n'est représentée que par son état, qui est un nombre entier compris entre 0 et le nombre d'états possibles pour un automate particulier. Les automates de notre application doivent pouvoir manipuler des grilles de cellules à une et deux dimensions. Nous stockons donc nos cellules dans une classe **Grid** qui contient une matrice d'entiers (si le nombre de lignes vaut 1 il s'agit d'une grille à une dimension). Pour gérer cette matrice, nous avons fait le choix d'utiliser un **conteneur** présent dans la **bibliothèque standard** (la STL). Ces conteneurs sont très intéressants à utiliser, car ils permettent entre autres de ne pas créer de dépendance forte entre notre code et la structure de données utilisée. Ainsi, il sera possible de changer de structure de données (par exemple pour gagner en performances) sans remettre en cause le reste de notre code. On respecte ici le principe ouvert/fermé qui dit que notre application doit être facilement extensible et modulable sans que cela ne remette en cause ce qui a déjà été développé auparavant. De plus, les conteneurs de la STL nous offrent une interface d'utilisation riche et simple à utiliser, ce qui nous permet de ne pas réinventer la roue constamment. Nous avons choisi d'utiliser la classe **vector** afin d'obtenir un vecteur de vecteurs d'entiers pour représenter notre matrice de cellules. Pour parcourir la grille, nous utilisons bien évidemment le patron de conception **itérateur** qui est directement implémenté dans la classe vector et de manière générale dans tous les conteneurs de la STL. Cela nous permet de rendre indépendant le parcours des données et la structure de données utilisée.

2.4 Gestion des simulations

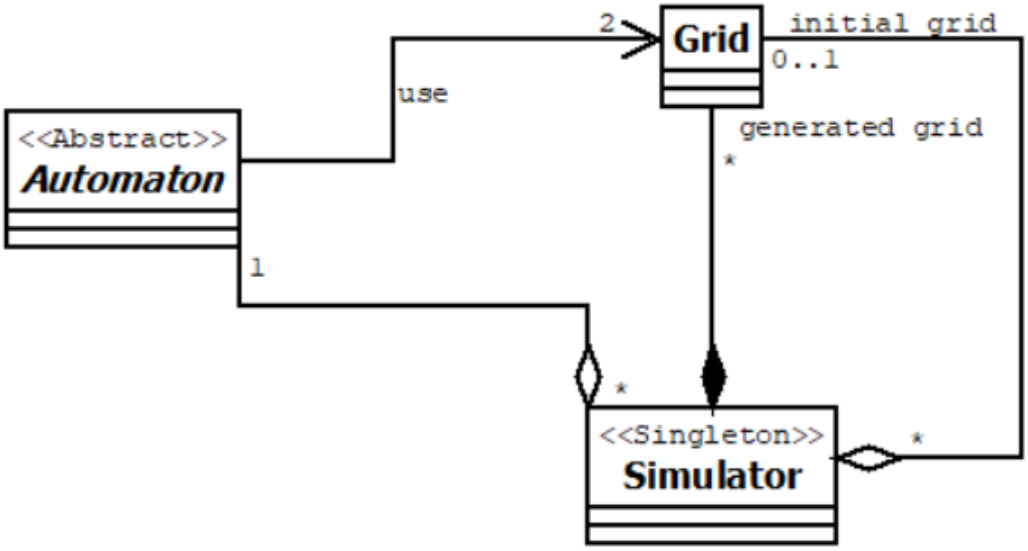

FIGURE 4 – Coupe du diagramme de classes axée sur le simulateur.

La gestion des simulations d'un automate, c'est-à-dire les passages successifs d'une grille de cellules à une autre en suivant une certaine règle de transition, est effectuée par un simulateur. Nous avons créé une classe **Simulator** qui regroupe toutes les caractéristiques et le comportement d'un simulateur : il s'agit de la clé de voute du programme. Un objet de la classe Simulator possède une référence sur un automate, mais n'est pas responsable du cycle de vie de cet automate. C'est donc bien une agrégation qui lie les deux classes. Il possède également une grille initiale, qui n'est d'autre qu'un pointeur sur un objet de la classe Grid pouvant être fourni ou non lors d'une construction. Pour les mêmes raisons que précédemment, une agrégation lie la classe Simulator et la classe Grid. En plus de cela, un simulateur génère différentes grilles lors des transitions successives. Il possède un buffer circulaire qui permet de garder en mémoire les dernières grilles de la simulation. Cela crée une composition entre Simulator et **Grid** puisque Simulator est totalement en charge du cycle de vie des grilles générées. Enfin, étant donné que les automates manipulent des grilles à travers la méthode *transition()*, on a une association unidirectionnelle d'**Automaton** vers Grid. Notons également que le buffer de grilles dans Simulator est géré à l'aide d'un **vector** de pointeurs de grilles. L'utilisation d'un conteneur de la STL se justifie de la même manière que dans la partie 2.3
Nous avons choisi d'implémenter le design pattern **Singleton** pour la classe

Simulator. En effet, puisque nous ne souhaitons pas faire plusieurs simulations simultanément dans notre application, il est inutile d'avoir plusieurs instances de Simulator en même temps. De plus, un simulateur contient des grilles et un automate donc est relativement lourd en mémoire. Il n'est donc pas forcément souhaitable de laisser la possibilité à l'utilisateur de créer plusieurs simulateurs ou d'en faire des copies. Le design pattern Singleton nous assure que l'on aura toujours au plus une instance active de Simulator dans notre programme et règle donc un problème logique et un problème de place mémoire.

2.5 Génération de grilles d'états

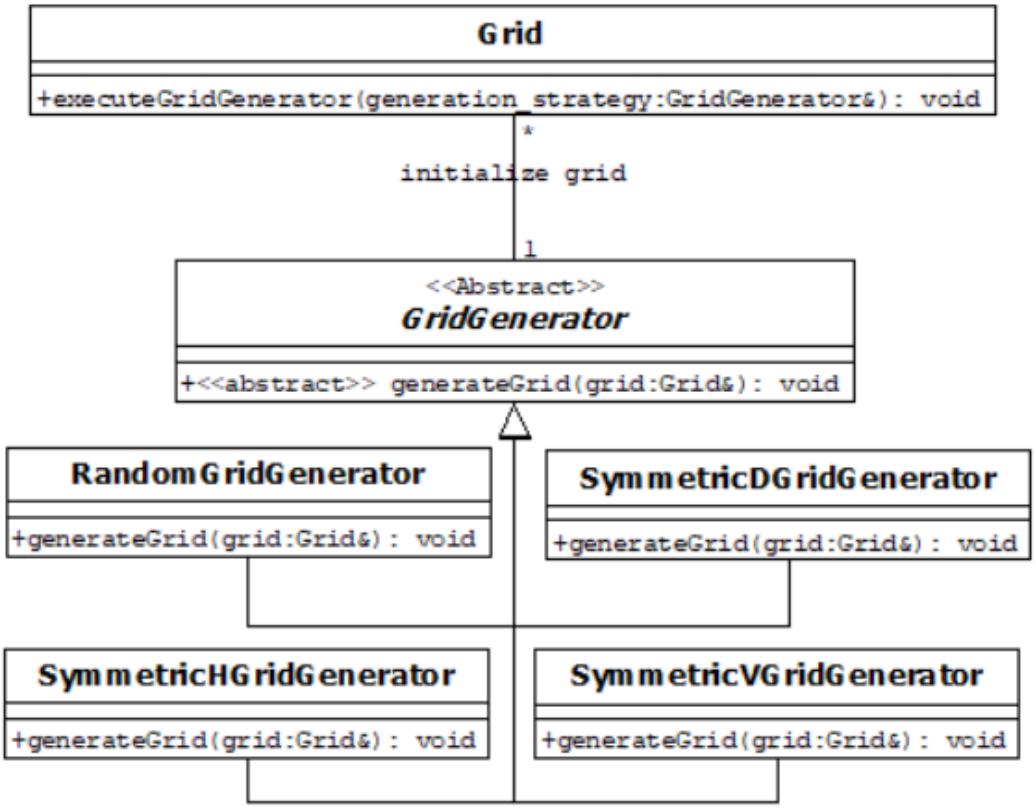

FIGURE 5 – Coupe du diagramme de classes axée sur les générateurs de grilles d'états.

Le logiciel AUTOCELL génère de base une grille où toutes les cellules sont à l'état « mort » (0). Dès lors, l'utilisateur a deux possibilités pour remplir la grille : la remplir manuellement (avec des setters) ou utiliser un générateur de

grilles. Dans l'état actuel de l'application, il existe 4 générateurs différents : un générateur aléatoire (qui remplit les cellules aléatoirement avec tous les états possibles d'une grille) et trois générateurs symétriques. Ces deniers réalisent un remplissage aléatoire symétrique par rapport à la diagonale (sous réserve d'une grille carrée), par rapport à l'axe verticale ou par rapport à l'axe horizontal. Le générateur de grilles sert donc à générer une grille de départ qui sera utilisée pour démarrer une simulation. Concrètement, l'utilisateur a le choix entre plusieurs algorithmes qui réalisent la même tâche, sans dépendance les uns des autres et qui sont donc interchangeables. Dès lors, nous avons choisi d'implémenter le patron de conception **stratégie** afin de pouvoir aisément permuter les différents algorithmes.
Pour ce faire, une classe abstraite **GridGenerator** contient une méthode virtuelle pure *generategrid()* qui prend en paramètre une référence sur une grille. De cette classe mère héritent les classes filles qui correspondent chacune à un générateur particulier. Dans chaque classe fille la méthode de génération de la classe mère est redéfinie pour correspondre à l'algorithme qu'elle implémente. Une classe de contexte, ici **Grid**, exécute la méthode de génération grâce à une méthode qui prend en paramètre une référence d'un objet de type GridGenerator. L'héritage public nous permet de dire que chaque générateur de grilles particulier « est un » GridGenerator. Ainsi, il est possible d'appeler la méthode *executeGridGenerator()* dans Grid avec n'importe quel générateur de grilles. Le polymorphisme mis en place grâce à la méthode virtuelle pure redéfinie dans les classes filles nous assure un comportement adéquat.

2.6 Sauvegarde et chargement d'automates

Pour sauvegarder et charger des automates, notre choix s'est porté sur l'utilisation d'une base de données **SQLite**. En effet, c'est un bon compris entre l'utilisation de fichiers (simple à mettre en place mais très pénible à maintenir) et l'utilisation d'un SGBD classique (en général assez lourd). Cette base de données est décrite en annexe A.1. Chaque type d'automate doit pouvoir être enregistré dans une table différente. Le simulateur gère un pointeur générique d'automate et les méthodes de sauvegarde prennent en paramètre un automate particulier (par exemple jeu de la vie). Néanmoins, le sous-classement (downcasting) n'est pas réalisé implicitement dans le cadre d'un héritage public puisqu'il s'agit d'une relation « est un » unidirectionnelle. Alors qu'il est tout à fait juste de dire qu'un **ElementaryAutomaton** « est un » **Automaton**, l'inverse n'est pas forcément vrai. De ce fait, un patron de conception **visiteur** a été implanté afin d'éviter l'utilisation fastidieuse et redondante des opérateurs de cast dynamique. Pour d'implanter ce patron de conception, une classe abstraite de gestion de sauvegarde a été implémentée.

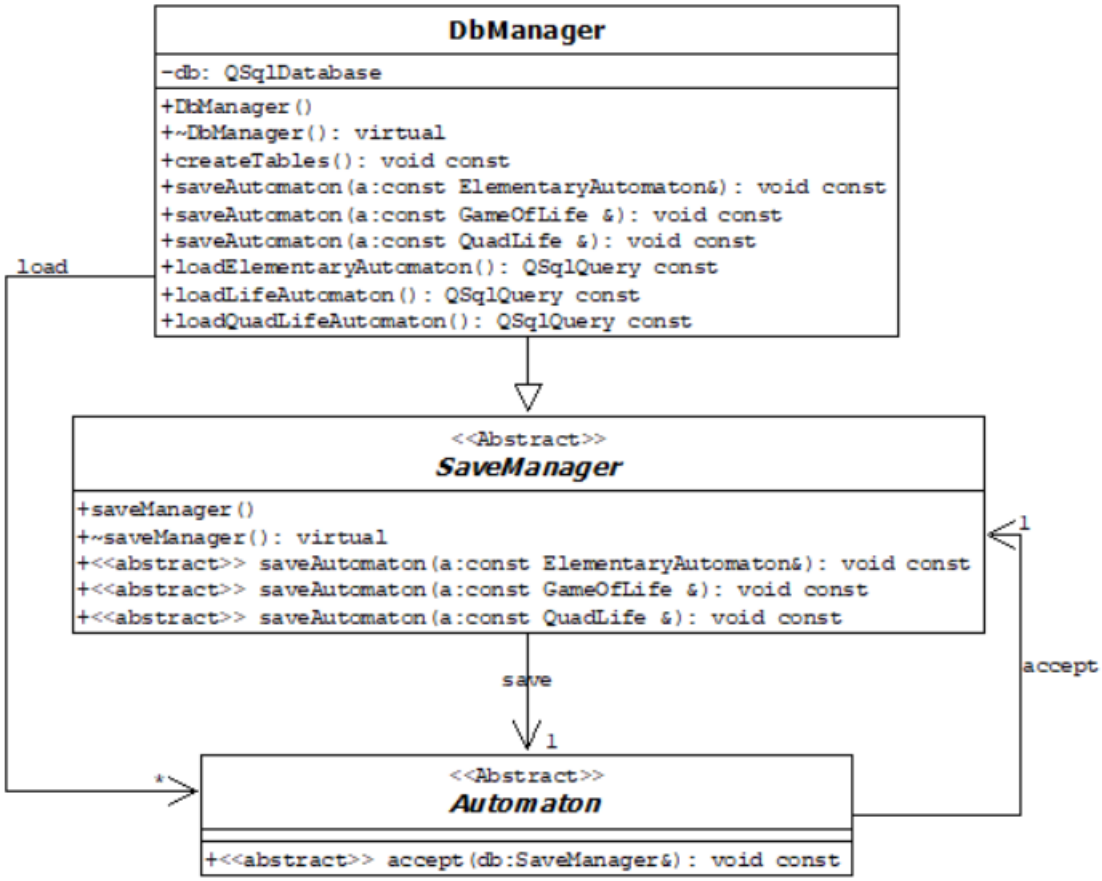

FIGURE 6 – Coupe du diagramme de classes axée sur le gestionnaire de sauvegarde d'automates.

Elle possède une méthode de sauvegarde virtuelle pure qui est surchargée trois fois pour correspondre aux trois types d'automates. Chaque méthode surchargée prend en paramètre une référence de l'automate correspondant. De cette classe dérive une classe gérant l'interface entre la base de données et l'application : **DbManager**. Lors de sa construction, l'objet se connecte automatiquement à la base et se déconnecte lors de sa destruction. Les trois méthodes surchargées sont redéfinies dans DbManager afin de pouvoir sauvegarder dans la base de données les attributs indispensables de nos automates. Dès lors, l'objet qui gère la sauvegarde des automates a besoin de pouvoir visiter les classes automates afin de pouvoir reconnaître l'objet qui lui a été transmis en paramètre. Pour cela, chaque automate doit avoir une méthode qui accepte la visite de notre gestionnaire d'automates. Cette dernière prend en paramètre une référence d'un gestionnaire d'automates et l'automate fait

appel à la méthode de sauvegarde du gestionnaire avec comme paramètre une référence à lui-même. Le simulateur se charge d'appeler la sauvegarde de son automate en lui demandant d'accepter le gestionnaire d'automates. Pour le chargement, la stratégie est différente : cette fois-ci la classe qui gère la sauvegarde possède trois méthodes de chargement différentes (chacune avec un nom différent) pour gérer le chargement de tous les automates d'un même type. En effet, l'objet qui gère le chargement sait déjà à quel type d'automate il a affaire, sa méthode étant appelée dans la partie graphique après avoir choisi le type d'automate souhaité.

2.7 Interface graphique

L'intérêt de ce rapport est de vous présenter l'architecture de l'application et non son interface graphique. Ainsi, les détails de l'interface humain-machine sont présents dans la vidéo qui accompagne ce rapport.
La première fenêtre est définie dans la classe **MainWindow**. Celle-ci permet de choisir le type d'automate que l'on souhaite simuler et instancie ensuite la fenêtre qui correspond à ce choix. Pour générer la fenêtre correspondante au bon automate, nous avons choisi de reprendre la même architecture que présentée dans la sous-partie 2.2. Ainsi, une classe abstraite **AutoCell** contient les attributs et méthodes communs entre toutes les fenêtres. Une autre classe abstraite, **AutoCellAbstractGol**, sert de modèle pour les fenêtres d'automate du type jeu de la vie. Enfin, les classes **AutoCellElem**, **AutoCellGol** et **AutoCellQuad** représentent un automate élémentaire, jeu de la vie et QuadLife respectivement.

3 Modularité

Lors de la réalisation d'une application en utilisant les concepts de la POO, il est nécessaire de respecter le principe ouvert/fermé : l'application doit être facilement extensible (ouverture) sans pour autant remettre en cause le code existant (fermeture). Ce principe est fondamental en Génie Logiciel puisqu'il permet d'être efficace dans l'ajout de nouvelles fonctionnalités tout en gardant une bonne maintenabilité. Nous allons montrer que notre application respecte ce principe à travers une bonne modularité.

3.1 Ajout d'un nouvel automate

Pour rappel, chaque automate particulier fait l'objet d'une classe dans notre application. Ainsi, si un utilisateur-programmeur souhaite ajouter un

automate, il doit créer une nouvelle classe. Cette nouvelle classe devra hériter de la classe abstraite **Automaton** pour obtenir les caractéristiques et comportements généraux d'un automate. Il est possible d'hériter directement d'Automaton comme c'est le cas par exemple pour **ElementaryAutomaton**, ou bien d'hériter d'une classe abstraite intermédiaire comme **AbstractGameOfLife** ou une autre classe qui aurait été créée pour généraliser des comportements. Par transitivité, le nouvel automate héritera bien de la classe Automaton. Si l'on souhaite hériter d'AbstractGameOfLife, il faudrait bien évidemment que le nouvel automate soit une variante du jeu de la vie. Nous avons interdit l'héritage depuis ElementaryAutomaton, **GameOfLife** et **QuadLife** à l'aide du mot clef final, puisque ces classes représentent des automates spécifiques qui n'ont pas pour but d'être de nouveau spécialisés.
En héritant de la classe abstraite Automaton, il est bien sûr nécessaire de redéfinir les méthodes virtuelles pures dont notre nouvelle classe hérite si on souhaite qu'elle soit concrète et donc instanciable. Ainsi, il faut redéfinir la méthode *transition()* en mettant en place la règle de transition de notre choix entre différentes grilles. De plus, nous avons choisi de forcer l'utilisateur à redéfinir la méthode *display()* qui possède une définition dans Automaton mais qui est quand même abstraite (ou virtuelle pure). Cette définition pourra être utilisée en premier lieu à l'aide de l'opérateur de résolution de portée avant de définir les instructions d'affichage spécifiques à notre nouvel automate. Si on hérite d'AbstractGameOfLife, il ne faudra pas redéfinir *transition()* (qui est concrète à ce niveau d'héritage), mais uniquement la méthode *lifeManager()* qui gère la naissance de nouvelles cellules.
En ce qui concerne la sauvegarde et le chargement d'automates, les ajouts nécessaires seront expliqués dans la partie 3.3.

3.2 Ajout d'un nouveau générateur d'états

L'ajout d'un nouveau générateur d'état est extrêmement facile dans l'application AutoCell. Ainsi, pour rappel, chaque générateur d'état est représenté par une classe qui hérite d'une classe abstraite de générateur d'état : **GridGenerator**. Celle-ci ne possède qu'une seule méthode virtuelle pure : *void generateGrid(Grid& grille)* qui prend en paramètre une référence d'une grille. Dès lors, il suffit de créer une nouvelle classe qui hérite publiquement de cette classe abstraite et dans celle-ci il suffit de redéfinir la méthode tout en gardant bien sûr la même nomenclature. En général, il faut récupérer le nombre de ligne et de colonnes avec *unsigned int getNbRow()* et *unsigned int getNbCol()* respectivement, parcourir toutes les cellules et à chaque fois, si nécessaire, changer la valeur de la cellule avec *void setCell(int i, int j, int val)* qui prend en paramètre la ligne i et la colonne j d'une cellule, mais aussi

l'état val qu'il faut lui donner.

A titre d'exemple, il est possible d'imaginer un générateur d'états qui, après avoir récupéré le nombre de lignes et de colonnes de la grille (notons l et c), parcourt tous les éléments à l'aide de deux boucles imbriquées (i et j sont les positions dans chaque boucle) commençant à partir de zéro et terminant à l−1 et c−1 respectivement. A chaque fois, la cellule est modifiée avec la valeur 1 (soit val = 1 et i et j comme premiers paramètres).

3.3 Sauvegarde d'un nouvel automate et changement du système

La gestion de la sauvegarde d'un automate est très intéressante, car elle est très modulable. Effectivement, il est possible d'ajouter un nouvel automate à sauvegarder et à charger très aisément. Il est aussi possible de changer de système de sauvegarde avec quelques modifications dans le programme.

Tout d'abord, grâce au patron de conception visiteur, une classe abstraite de sauvegarde **SaveManager** a été crée. Cette classe possède une méthode de sauvegarde virtuelle pure qui a été surchargée trois fois (pour les trois automates différents) : *void saveAutomate(const *Nom automate*&) const.* *Nom automate* est en fait le nom de la classe qui correspond à l'automate qu'il faut sauvegarder. Ainsi, la méthode prend en paramètre une référence sur un automate. La méthode est redéfinie dans le classe fille. Pour sauvegarder un nouvel automate, il faut donc surcharger une nouvelle fois la méthode à la fois dans la classe mère, mais aussi dans la classe fille où elle est utilisée. Dans le cas où on souhaite redéfinir la méthode dans la classe fille **DbManager**, il faut utiliser un objet de type **QSqlQuery** qui est fourni dans Qt pour accéder au contenu de la base de données. Pour le chargement, les méthodes dans la classe gérant base de données sont sous la forme *QSqlQuery load*Nom automate*() const* où *Nom automate* est encore une fois le nom de l'automate en question. Ici, la méthode n'est pas présente dans la classe mère abstraite. Pour charger un nouvel automate, il faut donc créer une nouvelle méthode. Pour plus de détails sur la base de données et sur les requêtes SQL, il faut se référer à l'annexe A.1.

Dans le cas du changement de système de sauvegarde, si jamais une base de donnée SQLite demande trop d'efforts (bien qu'elle soit très petite), il est possible de changer le système en créant une nouvelle classe fille qui hérite publiquement de classe mère abstraite. Les méthodes de sauvegarde peuvent être redéfinies sans soucis et sans aucun changement dans le reste du code. Pour le chargement malheureusement le type de retour change et ainsi il est nécessaire de faire d'autres modifications dans le reste du programme...

3.4 Modification partie graphique

L'intérêt de notre application est de pouvoir modifier la partie graphique sans pour autant impacter le reste du programme. Ainsi le cœur du programme, l'architecture, a été développé avant la partie graphique de manière à avoir une architecture autonome. Dès lors, la partie graphique a été créé sans ajouter de nouveaux attributs ou de nouvelles méthodes dans le cœur de l'application. L'interface graphique doit pouvoir se débrouiller qu'à partir de ce l'architecture lui donne. Au final, il est possible de modifier toute la partie graphique sans remettre en cause le fonctionnement du programme qui lui reste autonome.

A Annexes

A.1 Base de données

A.1.1 Tables

Chaque type d'automate a une table qui lui correspond. Dans cette table on ne sauvegarde que les arguments qui sont utilisés dans le constructeur. La clé primaire de chaque table est l'ensemble des arguments, ce qui empêche de sauvegarder deux automates identiques.

Automate élémentaire :

Numéro règle

Jeu de la vie :

Nb Naissance	Nb voisins min	Nb voisins max

QuadLife :

Nb voisins min	Nb voisins max

Table 3 – Tables de la base de données

La requête SQL pour créer une table est classique, et il suffit de la mettre entre guillemets en paramètre de la méthode **exec()** de l'objet **QSqlQuery**.

A.1.2 Sauvegarde

La sauvegarde consiste en une requête SQL d'insertion. On récupère les attributs utiles de l'automate et on les sauvegarde dans la bonne table. Pour bien utiliser l'objet **QSqlQuery**, on commence avec la méthode *prepare*() où on y insère entre guillemets une requête SQL du type INSERT INTO *nom de la table* VALUES (:"attribut 1", :"attribut 2", ...). Ensuite, on appelle la méthode *bindValue(" :attribut", *une méthode get pour récupérer l'attribut*)* autant de fois qu'il y a d'attributs à enregistrer. Enfin, on fait appel à la méthode *exec()* pour exécuter la requête SQL.

A.1.3 Chargement

Le chargement consiste en une requête SQL de sélection. On récupère tous les tuples de la table. L'application se chargera de choisir l'automate utile. La requête SQL est classique et on fait appel à la méthode *exec()* de l'objet **QSqlQuery** avec comme paramètre la requête entre guillemets.
Pour passer d'un tuple à l'autre on utile la méthode *next()*, pour accéder à un attribut c'est la méthode *value(*numéro colonne*).to*type de la variable**