LO21 Rapport AutoCell

Par Samy Nastuzzi, Simon Bazin et Alexandre Brasseur

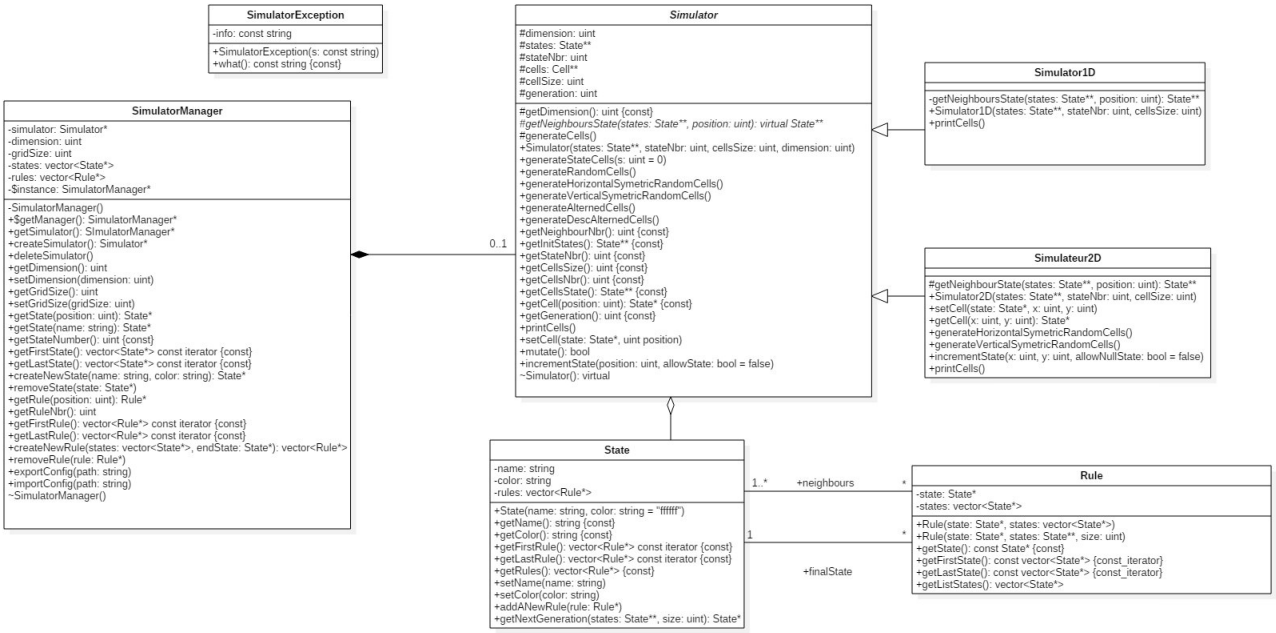
Préambule

L'application AutoCell a pour but d'implémenter des automates cellulaires à 1 et 2 dimensions afin de réaliser des simulations. Ces simulations peuvent être utiles notamment dans le domaine médicale, avec le développement de bactéries afin de prévoir des situations. Un automate est une grille virtuelle, dont les dimensions peuvent être sélectionnées. De plus, l'évolution d'étape en étape suit des règles prédéfinies. Cette évolution concerne des cellules, ici des cases du tableau, qui possèdent des états. Ces états sont à définir et réprésentent un état de vie d'une cellule. Ainsi, les états basiques sont les états morts et vivant, qui indiquent le statut de la cellule. Pour passer d'un état à un autre, des règles sont définies et concernent le nombre de voisins entourant chaque case. Cet automate concerne le 1D, 2D, 3D ou toute autre dimension.

Dans le cadre de ce projet, des automates cellulaires à 1 dimension et à 2 dimensions sont réalisés. Les états, règles et états de départs sont laissés au libre choix de l'utilisateur. De plus, le célèbre Jeu de la Vie proposé par le mathématicien britannique John Horton Conway est implémenté.

Au sein de ce rapport, la définition de l'implémentation, la présentation de la structure du projet, et une explication des méthodes réalisées seront énoncées. La modularité d'une implémentation diverse ou supplémentaire sera également justifiée.

Simulateur



La construction des simulateurs a été réalisée de manière à séparer les fonctionnalités propres grâce à l'héritage.

Description des classes

La classe `Rule` correspond aux règles de transition d'un état à un autre. L'état de départ est l'état auquel est attachée la règle. L'état d'arrivée est celui spécifié dans la règle. Une règle est définie par un vecteur d'états qui caractérisent le nombre de cellules à l'état spécifié qui doivent entourer la cellule initiale. Si la règle est respectée, la cellule passera d'un état à un autre.

La classe `State` correspond aux différents états possibles pour les cellules. Un état est défini par un nom et une couleur, la couleur qui sera affichée lors de la simulation. A ces attributs est associé un vecteur de règles.

La classe `Simulator` est une classe mère abstraite. La méthode `getNeighboursState` par exemple est virtuelle pure, et la classe ne peut donc pas être instanciée. Elle définie les fonctions essentielles au fonctionnement de tous simulateurs, peut importe sa dimension. Les attributs de cette classe définissent la dimension du simulateur, les états et le nombre, les cellules et le nombre, ainsi que le nombre de générations. Tout ceci est commun à tous les simulateurs, et des fonctions permettent de gérer les attributs et actions à réaliser lors des simulations.

Pour définir un simulateur fonctionnel, il faut l'implémenter avec la bonne simulation en héritant de la classe `Simulator`. Nous avons donc

De cette classe mère abstraite héritent les classes filles `Simulator1D` et `Simulator2D`. Les fonctions virtuelles pures, telles que `printCells()` doivent être définies dans les classes filles car la méthode n'est pas implémentée dans la classe mère.

La classe `Simulator1D` ne contient pas d'attributs propres, mais seulement quelques méthodes, dont le constructeur, qui appelle le constructeur de la classe mère, puisque c'est un héritage.

Pour la classe `Simulator2D`, des fonctions de la classe mère sont redéfinies, car elles ne correspondent pas tout à fait pour cette classe: par exemple, la position est indiqué par deux coordonnées (2D) et non une seule, comme déclaré dans la classe mère.

Une classe `SimulatorException` a été contruite pour gérer les erreurs qui pourraient survenir notamment lors de la mauvaise manipulation de l'utilisateur. Cette classe permet de gérer et d'envoyer des messages d'erreurs qui sont rattrapé par l'interface.

Enfin, la classe `SimulatorManager` est un singleton qui permet de gérer l'instanciabilité des simulateurs. De ce fait, il n'est plus possible de créer, détruire ou copier des simulateurs. Un seul simulateur est instancié durant la simulation et son accessibilité se réalise seulement via le singleton. De même, certains attributs du simulateur peuvent être accédés au travers de ce manager, qui possède des accesseurs. Ce manager gère aussi les états et les règles.

Implémentation

La Programmation Orientée Objet (POO) possède un principe fondamental qui est le principe d'encapsulation. Ce principe consiste à empêcher toute classe extérieure d'accéder et de modifier les attributs d'une classe. Ceci permet de masquer l'implémentation, et d'autoriser l'accès seulement à travers des méthodes implémentées. Ceci permet de garantir l'intégrité des données qui sont propres à une instance de classe.

Ainsi, pour toutes les classes, les attributs sont de type protégé (pour faciliter l'héritage) ou de typé privé. Le type protégé indique qu'une classe dérivée ou une classe définie comme amie peut accéder aux attributs de ce type, sans avoir à passer par des méthodes. Des attributs de type privé sont inaccessibles pour toute autre classe. Pour y accéder ou les modifier, des accesseurs(*getters*) et des modificateurs(*setters*) sont implémentés. C'est ce qui a été réalisé au sein des différentes classes de ce projet.

Ces méthodes sont reconnaissables de par leur nom: les getters ont un nom de fonction composé du préfixe *get* tandis que les setters sont composés du préfixe *set*.

D'autre part, des méthodes ont été implémentées pour permettre la sauvegarde et le chargement de fichiers JSON à exporter ou importer depuis le répertoire. Nous avons donc prédéfini des automates tels que le Jeu de la Vie en 2 dimensions dans le fichier `life_game.json`.

⚠ Le répertoire où sont placés automatiquement les fichiers contenant les sauvegardes des simulateurs est le fichier `projet/saves/`.

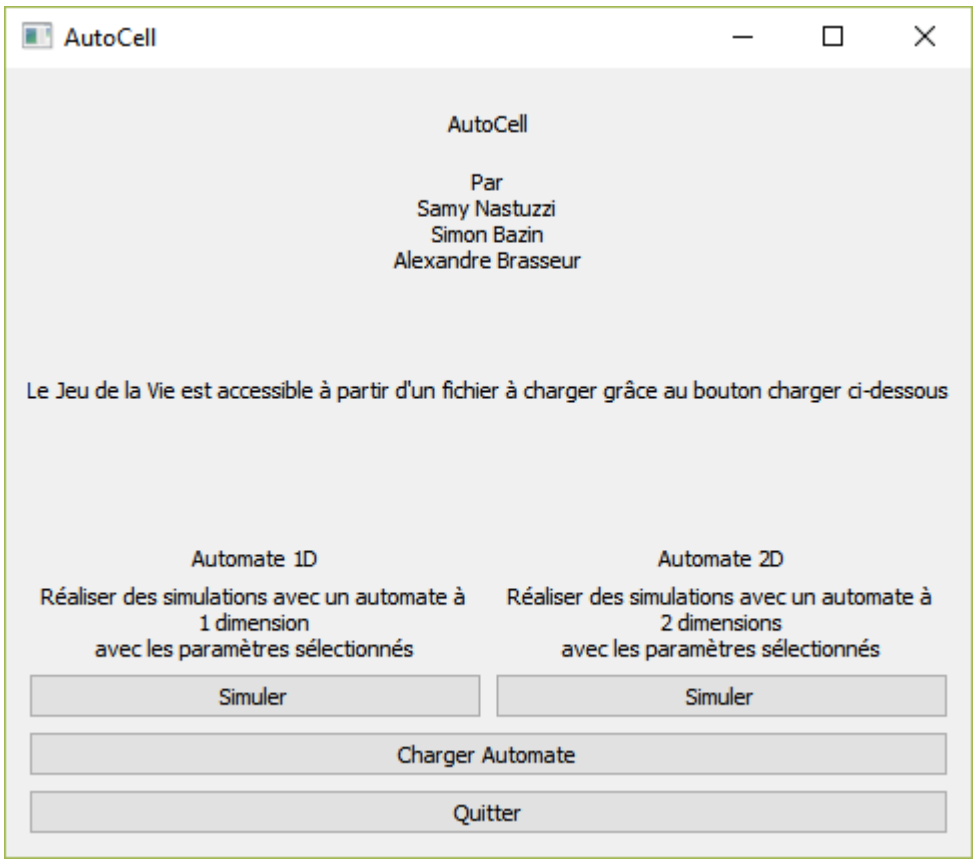
Interface

Présentation de l'interface

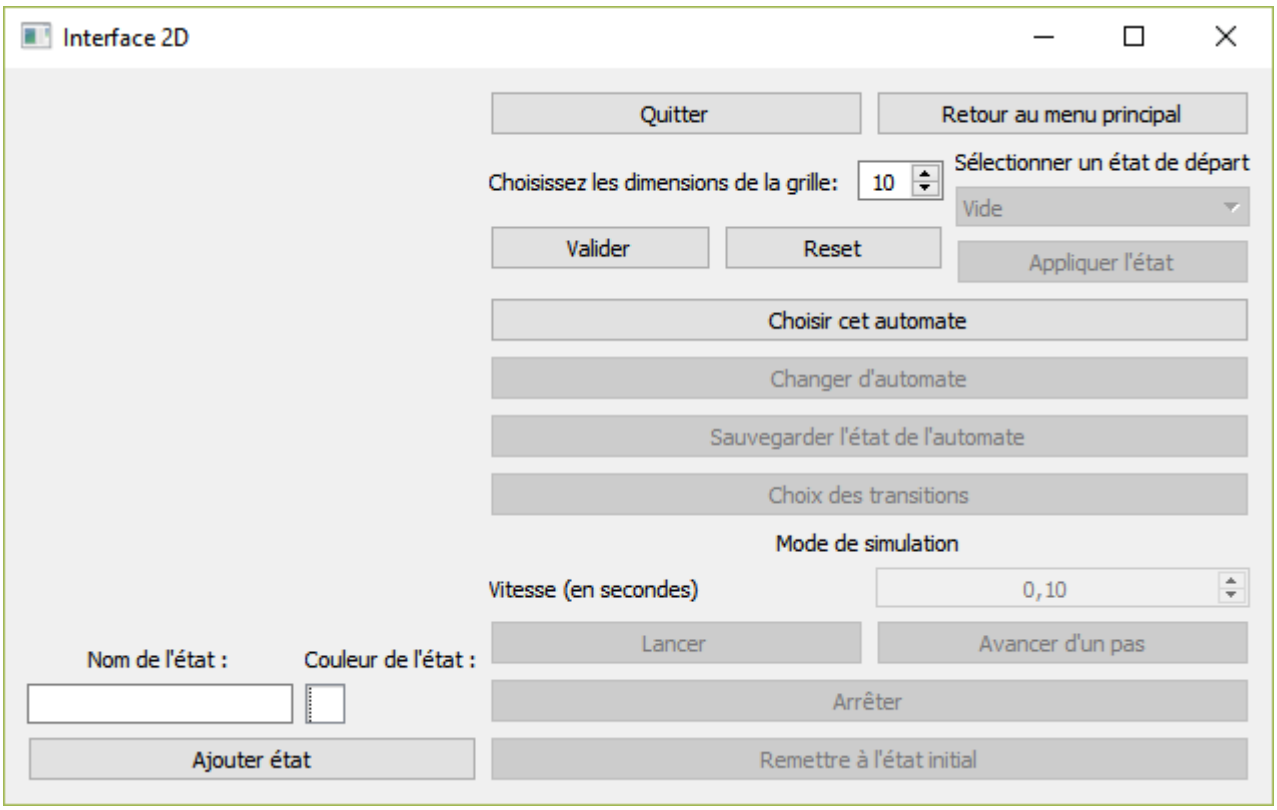
L'interface est composé de différentes classes qui permettent de définir des fenêtres. Nous avons choisi d'implémenter la simulation sous plusieurs fenêtres.

Fenêtre principale

Tout d'abord, la fenêtre principale permet de choisir quel type de simulation réaliser: en 1 dimension ou en 2 dimensions (voire plus). De plus, sur cette première fenêtre, il est possible de charger un fichier qui comporte des données de simulateur et ces données seront directement intégrées pour pour voir être lancées directement: la dimension, la taille, les états et les transitions définies dans le fichier seront générés automatiquement pour pouvoir lancer le simulateur.



Cependant, si l'utilisateur ne veut pas charger de fichier, il peut sélectionner l'automate qu'il veut simuler. En cliquant sur l'un des boutons, la fenêtre se ferme et une fenêtre s'ouvre, selon le simulateur sélectionné. Dans les deux cas, une fenêtre s'ouvre avec une interface très similaire. La seule différence est que pour l'interface 1D, il est possible de sélectionner la taille du buffer.



Sur cette fenêtre, il est possible d'ajouter et supprimer des états, dans la partie gauche. Il est nécessaire de sélectionner des états avec des noms et des couleurs différentes pour pouvoir l'ajouter. Ensuite, il est possible de changer les dimensions de la grille et, pour l'automate à 1 dimension, il est possible de changer la taille du buffer.

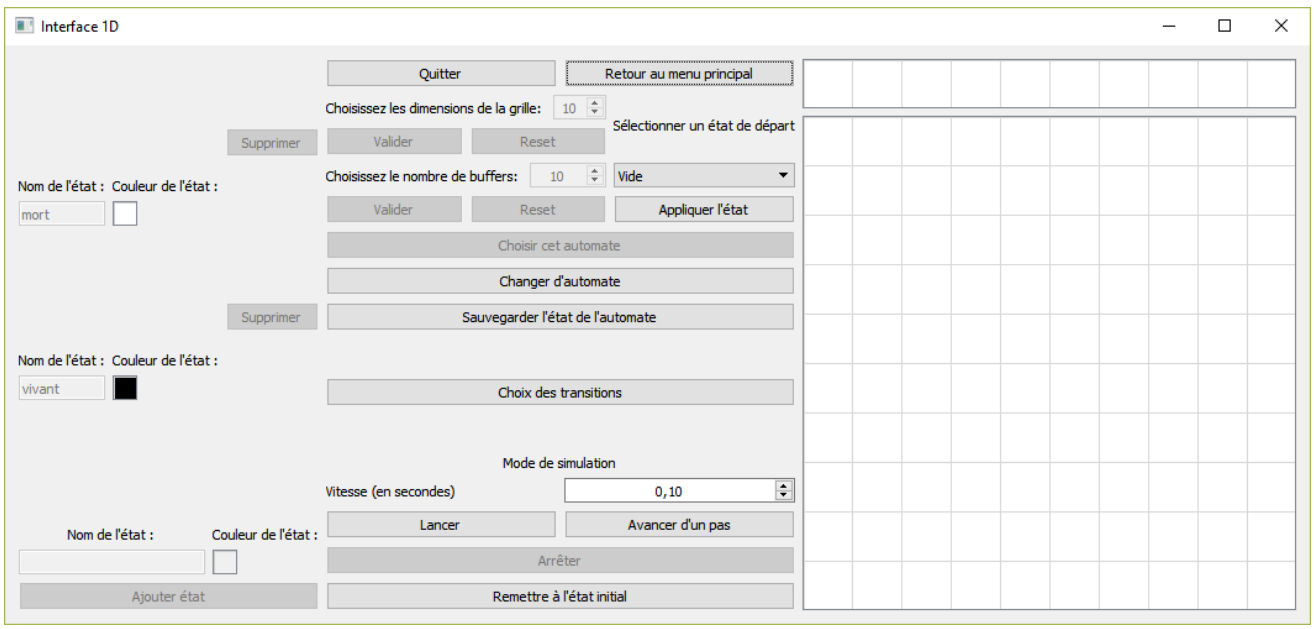
Une fois toutes ces actions réalisées, il faut cliquer sur `Choisir cet automate` pour pouvoir accéder à la simulation; les paramètres sont alors bloqués.

Une fois les paramètres sélectionnés, la fenêtre change et devient différente selon les dimensions.

Automate à 1 dimension

Si l'automate sélectionné est celui à 1 dimansion, la fenêtre change et afficher sur la partie droite un premier tableau à 1 dimension en haut puis, en-dessous, un tableau à deux dimensions. Le nombre de colonnes correspond à la dimension de la grille préselectionnée, alors que le nombre de lignes du deuxième tableau correspond à la taille du buffer. Les grilles sont initialisées avec le premier état inséré.

Voici l'affichage:



Il est alors possible de sélectionner différentes options, de changer les états du tableau initial, puis de lancer la simulation, en modifiant la vitesse ou en choisissant le mode *pas à pas*.

Le buffer correspond aux états précédent. La première ligne ne changera pas: c'est l'état initial. Ensuite, une fois la simulation lancée, la dernière ligne non vide est la dernière étape de la simulation. Ainsi, tant que la grille n'est pas pleine, on aperçoit toutes les étapes. Mais, une fois qu'elle est pleine, tous les états remonte d'une ligne pour laisser apparaître le dernier état simulé sur la dernière ligne.

Automate à 2 dimensions

L'interface est assez similaire à l'automate à 1 dimension. La principale différence est qu'il n'y a plus qu'une grille, qui simule les étapes. On ne possède plus les différentes étapes; seule l'étape actuelle est montrée à travers la grille.

Jeu de la Vie

Nous avons fait le choix de gérer le *Jeu de la Vie* depuis cette option de chargement de fichiers. Ainsi, pour simuler le Jeu de la Vie de John Horton Conway, il faut se rendre sur la fenêtre principale, sélectionner `Charger le fichier` et sélectionner le fichier `life_game.json` dans le répertoire `projet/saves`.

Notre automate : Apocalypse

Tout comme pour le *Jeu de la Vie* nous avons créer un automate *Apocalypse*. Il s'agit d'un automate à 4 états en 2 dimensions. Les 4 états sont :

- mort (blanc)
- humain (noir)
- militaire (bleu)
- zombi (vert)

Les règles sont les suivantes :

- un mort devient zombi s'il est entouré de 3 zombis ou plus
- un mort devient humain s'il est entouré de 3 humains ou plus
- un humain devient mort s'il est entouré d'aucun humain
- un humain devient militaire s'il est entouré de 3 militaires ou plus
- un humain devient zombi s'il est entouré de 2 zombis ou plus
- un militaire devient zombi s'il est entouré de 3 zombis ou plus
- un zombi devient mort s'il est entouré de 2 militaires ou plus
- un zombi devient mort s'il est entouré de 3 humains ou plus

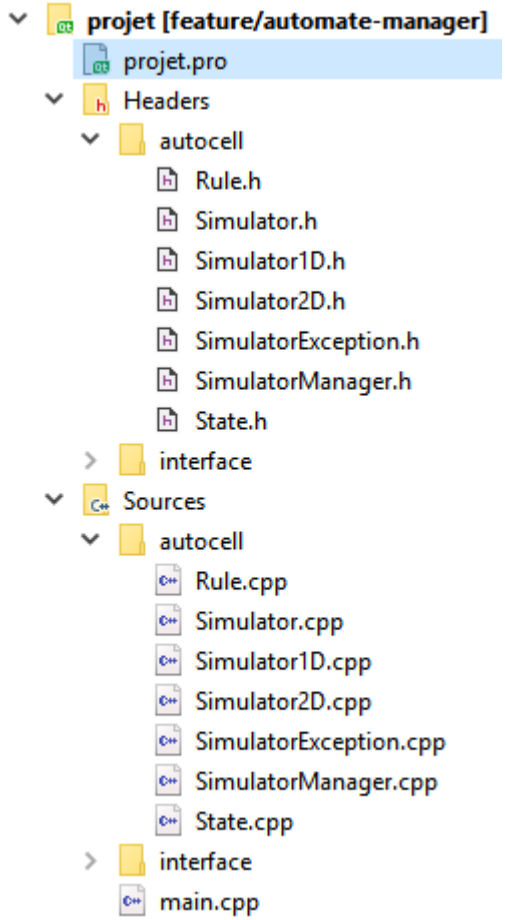
Comme pour le Jeu de la Vie, il faut se rendre sur la fenêtre principale, sélectionner `Charger le fichier` et sélectionner le fichier `apocalypse.json` dans le répertoire `projet/saves`.

Architecture du projet

Tous les fichiers C++ du projet se trouvent dans le dossier `projet/` :

- `interface/` : composants de l'interface Qt
- `autocell/` : l'ensemble des classes modélisant le problème Autocell

Voici l'architecture:



Architecture MVC

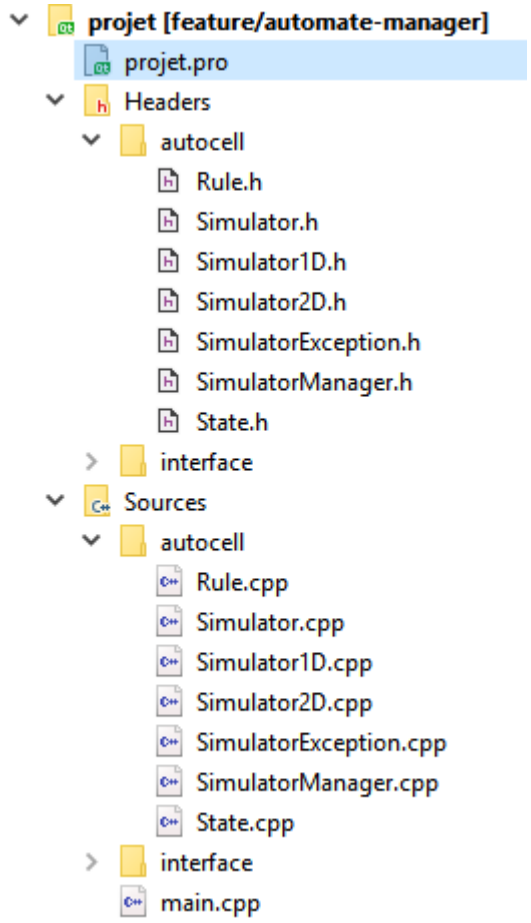
L'architecture de notre projet est basé sur une architecture MVC: Model, View, Controller.

Les fichiers du dossier `autocell/` sont propres à l'implémentation du backend. Ils correspondent au Modèle de données (Model). Ils permettent de gérer les différentes structures de données.

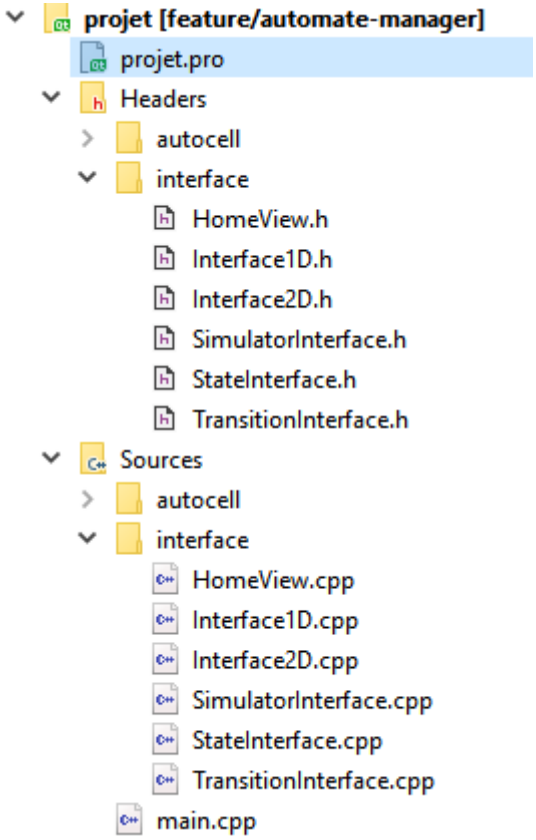
Les fichiers du dossier `interface/` sont utilisés pour la construction de l'interface qui a été réalisée avec le logiciel Qt. Ceci représente la Vue (View). Toute l'interface graphique est réalisée au sein de ces fichiers seulement.

Une liaison entre l'interface et les structures permet de rendre dynamique l'utilisation. Le `SimulatorManager` permet de contrôler (Controller) et superviser la bonne utilisation des automates. En effet, il est le seul à pouvoir autoriser la création, modification, suppression ou implémentation d'automates et autres attributs lui correspondant.

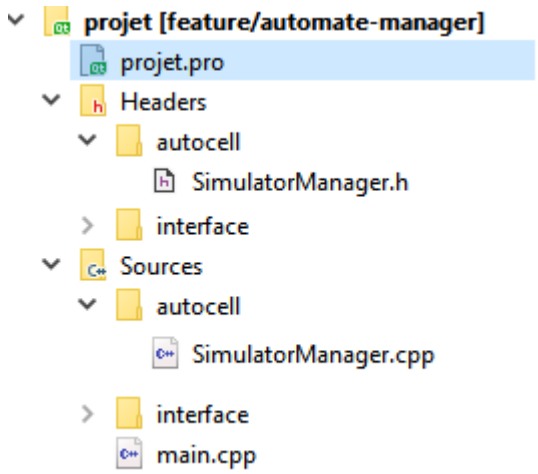
Pour le simulateur (Model):



Pour l'interface (View):



Pour le Manager (Controller):



Une architecture modulable

Tout d'abord, l'architecture MVC permet de séparer les différents éléments structurels et ainsi, laisse les développeurs se concentrer sur les aspects souhaités de manipulation. L'architecture a donc été réalisée pour être la plus extensible et flexible possible.

De plus, toutes les classes ont été implémentées avec un maximum d'héritage. Ceci permet d'avoir des classes abstraites regroupant un maximum de fonctions et attributs communs à toutes ses spécialisations. De ce fait, si de nouveaux automates doivent être implémentés, ils pourront hériter des méthodes d'une classe mère qui leur apportera une implémentation légère mais facilement extensible. Au niveau de l'interface comme du modèle de données, le maximum de méthodes et d'attributs ont été implémentés au sein de la classe mère (`SimulatorInterface` comme `Simulator`).

De même, des *template methods* ont été implémentées, notamment au niveau de l'interface. Ce sont des méthodes qui sont définies dans les classes mères, mais qui utilisent également des méthodes qui sont propres aux classes filles . Les méthodes générales sont souvent virtuelles, pouvant être redéfinies par les classes filles, et celles propres aux classes filles sont virtuelles pures : elles sont définies uniquement dans les classes filles, et la classe mère contient seulement la déclaration. Par exemple `SimulatorInterface::chosenAutomate` utilise la fonction `redrawGrid` qui est virtuelle pure.

D'autre part, l'utilisation d'un *singleton* avec le contrôlleur SimulatorManager, permet d'empêcher toute mauvaise manipulation depuis l'interface. Les méthodes et attributs ne sont pas accessibles et instanciables depuis n'importe quel endroit et par n'importe quelle méthode. Seul le manager permet l'instanciabilité de certains éléments. Ainsi, il est plus facile d'implémenter de nouvelles fonctionnalités et de gérer leur utilisation en empêchant les utilisateurs d'utiliser des outils pas encore développés.

Egalement, des vecteurs ont été utilisés au sein des classes. Nous avons donc utilisé des itérateurs pour agir sur ces différentes classes, ce qui permet de savoir sur quoi nous agissons.

Enfin, une classe d'exception `SimulatorException` a été créée pour gérer les éventuelles erreurs et afficher les messages d'erreur.

Conclusion

Au cours de ce projet, nous avons pu mettre en pratique toutes les compétences acquises lors des Travaux Dirigés et des Cours de LO21. Nous avons pu améliorer nos compétences en C++, ainsi que les principes de la Programmation Orientée Objet. Nous avons également utilisé des architectures et design pattern importants, qui nous permettent de gérer plus facilement certains cas.

De plus, nous avons découvert Qt, qui permet de faire des interfaces graphiques et de rendre des programmes dynamiques et interactifs avec une belle présentation.

Enfin, nous avons découvert les projets informatiques à plusieurs. Nous avons pour cela appris à manipuler des logiciels, tels que Git, qui permettent un travail collaboratif. De plus, nous nous sommes adaptés à la gestion de deadlines. Il a alors fallu mettre en place un planning en essayant au maximum de prévoir et de respecter des dates importantes avec objectifs fixés. Cela n'a pas été très facile puisque la majorité de l'implémentation a été apprise vers la fin du semestre.

Les différents outils utilisés nous ont aidés mais aussi posés quelques soucis et nous avons donc appris à les gérer ou les surmonter.