Exercice 23 - Automate

Remarque : Le but de ce TD est de se familiariser avec les notions d'association, d'agrégation et de composition entre classes et de leurs conséquences au niveau de l'implémentation des classes.

Le texte suivant s'inspire de la page WikipédiA sur les automates cellulaires que vous pouvez consulter.

Un automate cellulaire est un automate qui s'applique sur une grille régulière de « cellules », chaque cellule étant dans un « état » choisi parmi un ensemble fini. L'état d'une cellule au pas de temps $t+1$ est fonction de l'état au pas de temps t d'un ensemble défini de cellules de la grille que l'on appelle son « voisinage ». À chaque nouvelle unité de temps, les mêmes règles sont appliquées simultanément à toutes les cellules de la grille, produisant une nouvelle « génération » de cellules (*c.-à-d.* une nouvelle grille) dépendant entièrement de la génération précédente (*c.-à-d.* de la grille précédente).
Dans cet exercice, on souhaite développer une application destinée à **simuler l'exécution d'automates cellulaires élémentaires** où les grilles sont composées d'une seule ligne (il s'agit alors d'un automate à 1 dimension) et où les cellules peuvent prendre seulement deux états : « 0 » ou « 1 ». Le voisinnage d'une cellule donnée est constituée d'elle-même et des deux cellules qui lui sont adjacentes, *c.-à-d.* la cellule précédente et la cellule suivante. Chacune des cellules pouvant prendre deux états, il existe $2^3 = 8$ configuration possibles d'un tel voisinage. Il faut donc définir quel doit être l'état, à la génération suivante, d'une cellule pour chacunes de ces configurations. Puisqu'il y a $2^8 = 256$ façons différentes de faire ce choix, il existe donc 256 automates cellulaires élémentaires différents. On les désigne souvent par un entier entre 0 et 255 dont la représentation binaire est la suite des états pris par l'automate sur les configurations successives 111, 110, 101, 100, 011, 010, 001, 000.
Par exemple, considérons l'automate cellulaire élémentaire défini par la table suivante, qui donne la règle d'évolution :

Configuration initiale à l'instant t	1**1**1	1**1**0	1**0**1	1**0**0	0**1**1	0**1**0	0**0**1	0**0**0
Valeur suivante de la cellule centrale à l'instant $t+1$	0	0	0	1	1	1	1	0

Cela signifie que si par exemple, à un temps t donné, une cellule est à l'état « 1 », sa voisine de gauche à l'état « 1 » et sa voisine de droite à l'état « 0 » (configuration 1**1**0 du tableau), au temps $t+1$ elle sera à l'état « 0 ».
Par convention la règle précédente est nommée « règle numéro 30 », car 30 en décimale s'écrit 00011110 en binaire et 00011110 est la deuxième ligne du tableau ci-dessus, décrivant la règle d'évolution.

D'un point de vue implémentation, un automate cellulaire élémentaire correspond à un objet instance d'une classe `Automate` permettant de le manipuler. Les grilles sont des instances de la classe `Etat`. Un objet de la classe `simulateur` permettra d'appliquer un objet `Automate` sur plusieurs générations à partir d'un objet `Etat` représentant une grille de départ. Grâce à un buffer un objet `Simulateur` est capable de garder un nombre donné des derniers états générés.

La classe `Automate` comporte 2 attributs. L'attribut `numero` de type **`unsigned short`** désigne le numéro de la règle d'évolution de l'automate (30 dans l'exemple précédent). L'attribut `numeroBit` de type `string` représente aussi la règle d'évolution en utilisant une chaine de caractères de taille 8 ne contenant que des `'0'` et des `'1'` (`"00011110"` dans l'exemple précédent). Les méthodes `getNumero()` et `getNumeroBit()` permettent de connaître la valeur de ces attributs. La classe `Automate` propose deux constructeurs. L'un de ces constructeurs permet d'initialiser un automate avec le numéro de règle, l'autre permet d'initialiser un automate avec son numéro sous forme binaire. La méthode **`void`** `appliquerTransition(`**`const`** `Etat& dep, Etat& dest)`**`const`** permet d'appliquer la règle d'évolution sur un état désigné par `dep` pour obtenir un état qui sera désigné par `dest`. On peut écrire (afficher) un objet `Automate` sur un flux `ostream`.

La classe `Etat` possède un attribut `dimension` de type `size_t` qui représente le nombre de cellules impliquées dans la grille. Elle possède aussi un attribut `valeur` de type **`bool`**`*` qui pointera sur un tableau alloué dynamiquement, de taille `dimension`, et contenant des valeurs de type **`bool`**. Chaque valeur de ce tableau correspond à l'état d'une cellule. La classe possède un constructeur qui permet d'initialiser un objet `Etat` dont la dimension est précisé avec un paramètre de type `size_t` (de valeur 0 par défaut). Initialement, toutes les cellules de l'état sont de valeur **`false`** (correspondant à la valeur 0). La méthode « `size_t getDimension()`**`const`** » permet de connaître la dimension d'un état. La méthode « **`bool`** `getCellule(size_t i)`**`const`** » permet de connaître l'état de la cellule `i` (indice entre 0 et `dimension-1`). La méthode « **`void`** `setCellule(size_t i,` **`bool`** `val)` » permet de modifier l'état de la cellule `i` avec la valeur `val`. Il est possible de dupliquer un objet `Etat` par construction ou affectation à partir d'un autre état. On peut écrire (afficher) un objet `Etat` sur un flux `ostream`.

La classe `Simulateur` possède un attribut `automate` de type **`const`** `Automate&` référençant un objet de la classe `Automate`. Un attribut `depart` de type **`const`** `Etat*` pointe sur un éventuel état de départ (s'il a été donné). La méthode « **`void`** `setEtatDepart(`**`const`** `Etat& e)` » permet de modifier cet attribut avec un état représenté par `e`. Un attribut `nbMaxEtats` représente le nombre maximum des derniers états que l'objet `Simulateur` peut sauvegarder. Un attribut `etats` de type `Etat**` pointe sur un tableau alloué dynamiquement, de dimension `nbMaxEtats`, et contenant des valeurs de type `Etat*`. Un pointeur du tableau contient éventuellement l'adresse

d'un état alloué dynamiquement ou est égal à la valeur `nullptr`. Initialement tous les pointeurs du tableau `etats` sont initialisés avec la valeur `nullptr`.
Le constructeur « `Simulateur(const Automate& a, size_t buf = 2)` » permet d'initialiser un objet de la classe `Simulateur` qui simulera un automate référencé par `a` en utilisant un buffer de taille `buf` (2 par défaut). L'état de départ sera précisé plus tard avec la méthode `setEtatDepart()`. Le constructeur « `Simulateur(const Automate& a, const Etat& dep, size_t buffer = 2)` » permet, en plus, de préciser l'état de départ. Lorsque l'état de départ est modifié (avec le constructeur ou la méthode `setEtatDepart()`), `etats[0]` pointe sur une copie de l'état de départ et un attribut `rang` de type `size_t` est initialisé avec la valeur 0. Cet attribut permet de sauvegarder le rang du dernier état généré.
À la génération `rang`, la méthode `next()` permet de générer l'état à la génération `rang+1`. L'adresse de l'état généré à la génération `rang` est sauvegardé dans la cellule `rang % nbMaxEtats` du tableau `etats`, de sorte que seuls les `nbMaxEtats` derniers états sont sauvegardés. Les emplacements mémoire pointés par les pointeurs du tableau sont efficacement réutilisés au fur et à mesure des générations.
La méthode « `void run(size_t nbSteps)` » permet d'avancer de `nbSteps` générations en une seule fois. La méthode `reset()` permet de revenir à l'état de départ. La méthode « `const Etat& dernier()const` » permet d'accéder au dernier état généré. La méthode « `size_t getRangDernier()const` » permet de connaître le rang du denrier état généré.

Préparation : Créer un projet vide et ajouter trois fichiers `automate.h`, `automate.cpp` et `main.cpp`. Définir la fonction principale `main()` dans le fichier `main.cpp`. S'assurer que le projet compile correctement. Dans cet exercice, on tâchera de mener une approche "compilation séparée". Au fur et à mesure de l'exercice, on pourra compléter la fonction principale en utilisant les éléments créés. Les situations exceptionnelles seront gérées en utilisant la classe d'exception suivante (à recopier dans le fichier `automate.h`) :

```
#ifndef _AUTOMATE_H
#define _AUTOMATE_H
#include <string>
class AutomateException {
public:
 AutomateException(const std::string& message):info(message) {}
 std::string getInfo() const { return info; }
private:
 std::string info;
};
#endif
```

automate.h

Question 1

Identifier les différentes entités du monde décrit ci-dessus. Identifier les associations qui existent entre ces classes. Quel type de lien existe t-il entre un objet `Simulateur` et les objets `Etat` qu'il crée et auxquels il donne accès ? Quel type de lien existe t-il entre un objet `Simulateur` et l'objet pointé par l'attribut `depart` ? Quel type de lien existe-t-il entre un objet `Simulateur` et l'objet `Automate` auquel il est associé ? Établir un modèle UML où apparaissent les différentes classes utilisées dans l'application ainsi que les associations entre ces classes.

Question 2

Définir la classe `Etat` ainsi que l'ensemble de ses méthodes. La classe `Etat` nécessite-t-elle (a priori) un destructeur, un constructeur de recopie et/ou un opérateur d'affectation ? Expliquer. Définir ces méthodes seulement si nécessaire. Surcharger l'opérateur << de manière à pouvoir écrire un objet `Etat` sur un flux `ostream`.

Question 3

Définir la classe `Automate` ainsi que l'ensemble de ses méthodes. Quel est l'intérêt d'utiliser des références **const** pour le paramètre du constructeur qui permet d'initialiser un automate avec une règle d'évolution sous forme binaire ? La classe `Automate` nécessite t-elle (a priori) un destructeur, un constructeur de recopie et/ou un opérateur d'affectation ? Expliquer. Définir ces méthodes seulement si nécessaire. Surcharger l'opérateur << de manière à pouvoir écrire un objet `Automate` sur un flux `ostream`. On pourra utiliser les deux fonctions suivantes permettant de passer d'un `numero` à un `numeroBit` :

```
unsigned short NumBitToNum(const std::string& num) {
 if (num.size() != 8) throw AutomateException("Numero d'automate indefini");
 int puissance = 1;
 unsigned short numero = 0;
 for (int i = 7; i >= 0; i--) {
```

```
  if (num[i] == '1') numero += puissance;
  else if (num[i] != '0') throw AutomateException("Numero d'automate indefini");
  puissance *= 2;
 }
 return numero;
}

std::string NumToNumBit(unsigned short num) {
 std::string numeroBit;
 if (num > 256) throw AutomateException("Numero d'automate indefini");
 unsigned short p = 128;
 int i = 7;
 while (i >= 0) {
  if (num >= p) { numeroBit.push_back('1'); num -= p; }
  else { numeroBit.push_back('0'); }
  i--;
  p = p / 2;
 }
 return numeroBit;
}
```

Question 4

Définir la classe `Simulateur` ainsi que l'ensemble de ses méthodes. La classe `Simulateur` nécessite t-elle (a priori) un destructeur ? Expliquer. Définir cette méthode seulement si nécessaire.
Bien que la classe `Simulateur` ne gère a priori pas correctement (si on ne fait rien de plus) la construction par recopie et l'affectation, cet aspect sera ignoré dans le cadre de cet exercice repris plus tard.
À titre d'exemple, le programme

```
int main() {
 Automate a(30);
 std::cout << "automate " << a << "\n";
 Etat e(22); e.setCellule(11, true);
 Simulateur s(a,e);
 cout << e << "\n";
 for (size_t i = 0; i < 10; i++) {
  s.next(); cout << s.dernier()<<"\n";
 }
 return 0;
}
```

devrait produire l'affichage suivant (les cellules actives sont marquées avec `X`) :

```
automate 30 : 00011110

           X
          XXX
         XX  X
        XX XXXX
       XX  X   X
      XX XXXX XXX
     XX  X    X  X
    XX XXXX  XXXXXX
   XX  X   XXX     X
  XX XXXX XX  X   XXX
 XX  X    X XXXX XX  X
```

Exercice 24 - Problèmes de conception

Dans l'application, les objets `Automate` sont gérés par un module appelé `AutomateManager` qui est responsable de leur création (et destruction) et de leur sauvegarde. La classe possède deux méthodes `getAutomate()` qui permettent d'accéder (éventuelllement de créer) un automate dont le numéro (entier ou en binaire) est transmis en argument. Seul l'instance de la classe `AutomateManager` peut créer des objets `Automate`.

Question 1

Est-il possible de définir un tableau (alloué dynamiquement ou non) d'objets `Automate` sans fournir d'initialisateur ? Expliquer. Est-il possible de créer un tableau (alloué dynamiquement ou non) de pointeurs d'objet `Automate` sans fournir d'initialisateur ? Expliquer.

Question 2

Expliciter des intérêts de mettre en place le Design Pattern *Singleton* pour la classe `AutomateManager`. Développer la classe `AutomateManager` en implémentant ce design pattern. Mettre à jour le diagramme de classe.

Question 3

On remarque que la duplication malencontreuse d'un objet `Simulateur` pourrait poser des problèmes. Mettre en place les instructions qui permettent d'empêcher la duplication d'un objet `Simulateur`.

Question 4

Afin de pouvoir parcourir séquentiellement les derniers états générés et stockés dans le buffer d'un `Simulateur`, appliquer le design pattern *Iterator* à cette classe en déduisant son implémentation du code suivant. Ces itérateurs permettront de parcourir les derniers états générés depuis le dernier généré jusqu'au plus ancien encore présent dans le buffer.

```
void afficherEtModifierEtats(Simulateur& s){
  for(Simulateur::Iterator it= s.getIterator();!it.isDone();it.next()){
   std::cout<<it.current()<<"\n";
   it.current().setCellule(0, false); // modification possible
  }
}
//...
void afficherEtats(const Simulateur& sconst){
  for(Simulateur::ConstIterator it= sconst.getIterator();!it.isDone();it.next()){
    std::cout<<it.current()<<"\n"; //ok
    //it.current().setCellule(0, true); // modification impossible
  }
}
```

Question 5

Refaire la question précédente en proposant une interface d'itérateur similaire à celle utilisée par les conteneurs standards du C++ (STL), *c.-à-d.* qui permet de parcourir séquentiellement les différents états d'un objet `Simulateur` avec le code suivant :

```
void afficherEtModifierEtats(Simulateur& s){
 for(Simulateur::iterator it=s.begin();it!=s.end();++it)
   std::cout<<*it<<"\n";
}
void afficherEtats(const Simulateur& sconst){
 for(Simulateur::const_iterator it=sconst.begin();it!=sconst.end();++it)
   std::cout<<*it<<"\n";
}
```