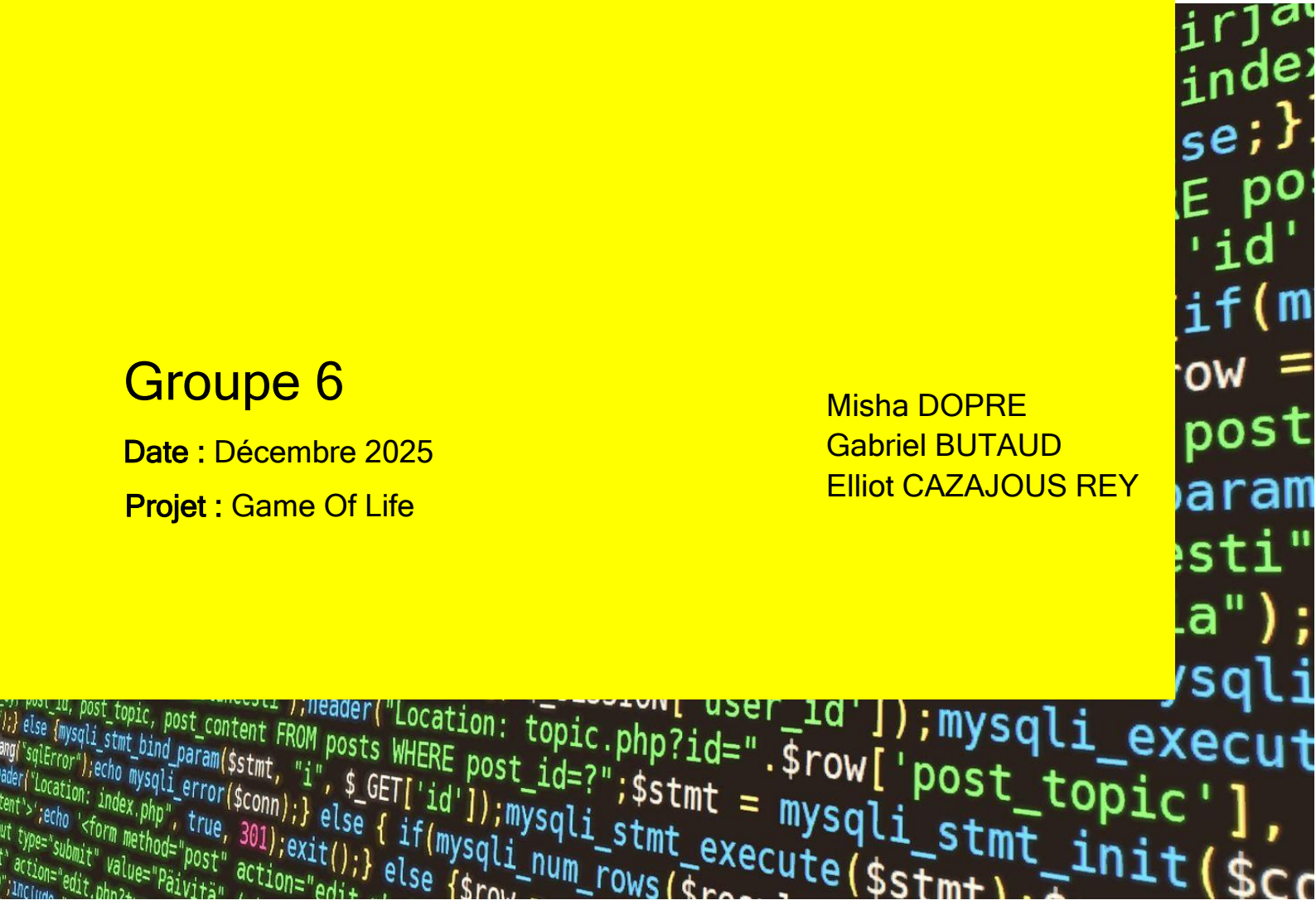
DOCUMENTATION DÉVELOPPEUR

Game Of Life



Groupe 6

Date : Décembre 2025

Projet : Game Of Life

Misha DOPRE
Gabriel BUTAUD
Elliot CAZAJOUS REY

CESI_
ÉCOLE D'INGÉNIEURS

Table des matières

1. Architecture du projet

Veuillez retrouver en annexes les diagrammes d'activité, de classes ainsi que de séquence décrivant plus en détails certains points de notre architecture.

1.1 Structure des fichiers

```
Game-Of-Life/
├─── include/              -> Headers (. h)
|    ├─── Cell.h           -> Classe cellule
|    ├─── State.h          -> États des cellules (polymorphisme)
|    ├─── Grid.h           -> Grille de jeu
|    ├─── Rule.h           -> Règles du jeu
|    ├─── Interface.h      -> Interfaces console/graphique
|    └─── FileManager.h    -> Gestion des fichiers I/O
├─── src/                  -> Implémentations (. cpp)
|    ├─── Cell.cpp
|    ├─── State.cpp
|    ├─── Grid.cpp
|    ├─── Rule. cpp
|    ├─── Interface.cpp
```

```
|   └──── FileManager.cpp
├──── main.cpp              -> Point d'entrée
├──── tests. cpp            -> Tests unitaires
├──── CMakeLists.txt        -> Configuration CMake
└──── input.txt             -> Exemple de grille
```

1.2 Dépendances

Dépendance	Version	Usage
SFML	2.6.1	Interface graphique (téléchargée automatiquement)
STL C++	C++20	Conteneurs, pointeurs intelligents

1.3 Application des principes SOLID

Principe	Application
Single Responsability	Cell gère une cellule, Grid gère la grille, FileManager gère les fichiers
Open/Closed	Nouvelles règles ajoutables via GameRule sans modifier Grid
Liskov Substitution	AliveState, DeadState, ObstacleState sont interchangeables
Interface Segregation	Interface expose uniquement run()
Dependency Inversion	Grid dépend de l'abstraction GameRule, pas de Conway

1.4 Design patterns utilisés

State Pattern pour les états des cellules :

- Chaque état (vivant, mort, obstacle) est une classe distincte
- Le comportement est encapsulé dans l'état, pas dans la cellule
- Facilite l'ajout de nouveaux états sans modifier Cell

Strategy Pattern pour les règles du jeu :

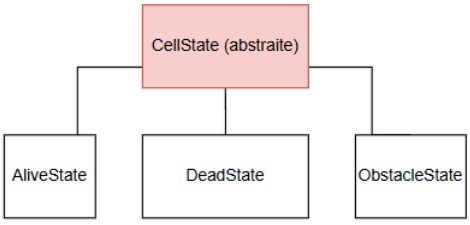
- Les règles sont externalisées dans une hiérarchie séparée
- Permet de changer de règles (Conway, HighLife.. .) sans toucher à la grille
- La grille délègue le calcul du prochain état à la règle

Template Method pour les interfaces :

- Interface définit le squelette (run())
- ConsoleInterface et GraphicInterface implémentent différemment

2. Hiérarchie des classes

2.1 Etats des cellules (State pattern)



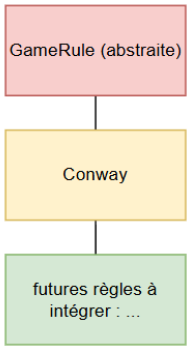
CellState, classe de base abstraite :

```
class CellState {
public:
    virtual bool estVivant() const = 0;      // Est-ce une cellule vivante ?
    virtual char symbole() const = 0;         // Symbole pour export fichier
    virtual std::unique_ptr<CellState> clone() const = 0;  // Deep copy
    virtual bool estStatique() const { return false; }     // Immuable ?
};
```

Classe	estVivant()	symbole()	estStatique()
AliveState	true	1	false
DeadState	false	0	false
ObstacleState	false	2	true

Pourquoi clone() ?
Les smart pointers (unique_ptr) ne peuvent pas être copiés directement. La méthode clone() permet de dupliquer un état sans copier le pointeur.

2.2 Règles du jeu (Strategy Pattern)



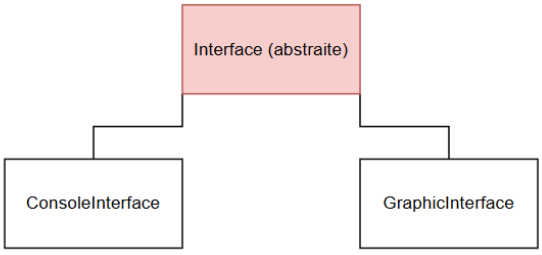
GameRule, interface des règles :

```
class GameRule {
public:
    virtual std::unique_ptr<CellState> apply(
        const CellState& currentState,
        int aliveNeighbors
    ) const = 0;
};
```

Conway, implémente les règles classiques :

- Cellule vivante + 2 ou 3 voisins -> reste vivante
- Cellule morte + exactement 3 voisins -> devient vivante
- Sinon -> meurt

2.3 Interfaces d'affichage (Template Method)



Classe	Responsabilité
ConsoleInterface	Sauvegarde chaque itération dans <fichier>_out/iter_X.txt
GraphicInterface	Affichage SFML temps réel avec contrôles clavier

2.4 Relations entre les classes

Voir annexe Diagramme de classes.

3. Fonctionnement interne

3.1 Cycle de vie d'une simulation

1. main. cpp
   └── Lecture des arguments (fichier, mode, itérations)
   └── Création de Grid

2. Grid::loadFile()
   └── FileManager::readGridFile() lit le fichier
   └── Création des Cell avec états appropriés (0->Dead, 1->Alive, 2->Obstacle)

3. Boucle de simulation (via Interface::run())
   └── Grid::parcoursGrille() -> Calcule les prochains états
   └── Grid::update() -> Applique les changements
   └── Affichage ou sauvegarde

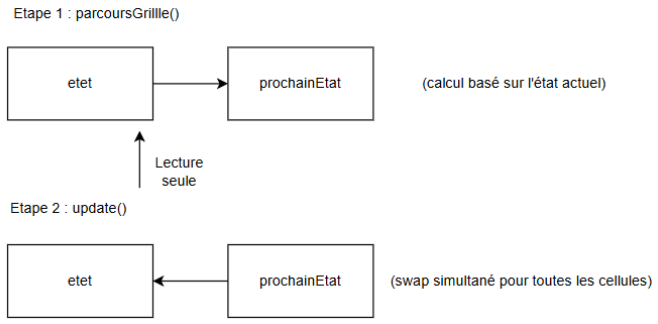
3.2 Double buffering

Le double buffering est essentiel pour éviter que les mises à jour n'influencent les calculs en cours.

Problème sans double buffering :

Cellule A calculée -> devient morte
Cellule B (voisine de A) calculée -> voit A morte alors qu'elle était vivante

Solution avec double buffering :



Ainsi, toutes les cellules sont calculées avec l'état de la génération précédente, puis toutes basculent en même temps.

3.3 Flux de calcul d'une cellule

```
// Dans Grid::parcoursGrille()
for (chaque cellule) {
    if (cellule.estObstacle()) {
        // Obstacle : garde le même état
        cellule.prepareProchainEtat(cellule.getState().clone());
        continue;
    }

    int voisins = compteurVoisin(x, y);  // Compte les voisins vivants
    auto nouvelEtat = rule->apply(cellule.getState(), voisins);  // Applique Conway
    cellule. prepareProchainEtat(std::move(nouvelEtat));
}

// Dans Grid::update()
for (chaque cellule) {
    cellule.majEtat();  // prochainEtat → etat
}
```

4. Fonctionnalités bonus

4.1 Grille torique

Dans une grille classique, les cellules aux bords ont moins de voisins. La grille torique connecte les bords opposés.
Implémentation dans Grid::compteurVoisin() :

```
int nx = (x + dx + rows) % rows;  // Wrap vertical
int ny = (y + dy + cols) % cols;  // Wrap horizontal
```

Explication :

- x + dx peut être négatif (-1) ou dépasser la taille (rows)
- On ajoute rows pour éviter les négatifs
- Le modulo % ramène dans les bornes [0, rows-1]

Exemple (grille 5x5) :

- Position (0, 0), voisin gauche : (0 + (-1) + 5) % 5 = 4 -> dernière colonne
- Position (4, 4), voisin bas : (4 + 1 + 5) % 5 = 0 -> première ligne

4.2 Cellules obstacles

Les obstacles sont des cellules immuables qui ne participent pas à la simulation.

Caractéristiques :

- estStatique() retourne true
- estVivant() retourne false (ne compte pas comme voisin vivant)
- Ignorées dans le calcul des règles (optimisation)

Dans Grid::parcoursGrille() :

```
if (cells[i][j]. estObstacle()) {
    cells[i][j]. prepareProchainEtat(cells[i][j].getState().clone());
    continue;  // Pas de calcul de voisins ni de règle
}
```

5. Tests unitaires

5.1 Structures des test

Les tests utilisent assert() et une macro TEST_PASSED pour l'affichage.

Test	Ce qu'il vérifie
testCell()	Création, changement d'état, transformation en obstacle
testFileManager()	Lecture correcte des valeurs 0, 1, 2
testGridLogic()	Logique torique + immutabilité des obstacles après itération
testStableShape()	Le bloc (2*2) reste stable après N itération

5.2 Exécution

.\build\Release\RunTests.exe

5.3 Ajouter un test

```
void testMonTest() {
    // Arrange : préparer les données
    Cell c(true);

    // Act : exécuter l'action
    c.prepareProchainEtat(std::make_unique<DeadState>());
    c.majEtat();

    // Assert : vérifier le résultat
    assert(c.estVivant() == false);

    TEST_PASSED("MonTest");
}

// Ajouter dans main() de tests. cpp :
testMonTest();
```