Jeu de la vie

Documentation technique complète

Auteurs

Abderrahmane Frigaa
Mehdi Benahmed

Date

December 7, 2025

Version : December 7, 2025

Contents

1 Introduction

Ce document détaille l'ensemble de la base de code du projet **Jeu de la vie** (C++17, SFML), ses modules, ses flux et ses choix d'architecture. L'application implémente l'automate cellulaire de Conway avec deux modes (console et SFML) et une architecture en quatre couches pour séparer la logique métier, l'orchestration, l'infrastructure et la présentation.

2 Objectifs et principes

- Séparation des responsabilités (4 couches) pour testabilité et évolutivité.
- Pas de dépendances I/O ou SFML dans le domaine.
- Ports (interfaces) pour le chargement/export, consommation via une façade applicative.
- UI réactive et autonome (console ou SFML) basée sur la façade.

3 Architecture en couches

- **Domaine (domain/)** : logique pure du Jeu de Conway (modèles, règles, évolution).
- **Application (application/)** : ports (interfaces) et `SimulationService` pour orchestrer la simulation.
- **Infrastructure (infrastructure/)** : implémentations techniques des ports (lecture/écriture de fichiers).
- **Présentation (ui/)** : interfaces utilisateur (console, SFML) consommant la façade.

4 Dépendances et build

- C++17, `g++`/`clang++`.
- SFML 3.x (modules graphics, window, system) pour l'UI graphique.
- `pkg-config` pour récupérer les flags SFML.
- **Makefile** : `make` (binaire `jeu`), `make clean`, `make test` (runner sans SFML).

5 Couche Domaine (domain/)

5.1 États et cellules

- `CellState` : interface abstraite (`isAlive()`, `clone()`); contrat commun pour tout état.
- `AliveState`, `DeadState` : états concrets vivants/morts, clonables; utilisés par défaut pour Conway.
- `Cell` : position (ligne, colonne) + pointeur possédé vers `CellState`. Copie profonde (clone) et move; `setState` remplace l'état (delete de l'ancien).

5.2 Règles

- `Rule` : stratégie d'évolution (`nextState(const Cell&, const Grid&)`) renvoie un nouvel état alloué; point d'extension pour d'autres automates.
- `ConwayRule` : implémente B3/S23 (naissance sur 3 voisins, survie sur 2 ou 3) et instancie les `CellState` correspondants.

5.3 Grille

- `Grid` : matrice de `Cell`, dimensions, flag torique; initialise des cellules mortes par défaut.
- `countAliveNeighbors` : comptage des voisins (avec ou sans torus via modulo).
- `equals` : comparaison binaire vivante/morte cellule par cellule.

5.4 Simulation

- `GameOfLife` : détient la grille courante et la précédente, une règle, itération courante et itération max.
- `step()` : copie la grille, applique la règle à chaque cellule, incrémente l'itération.
- `isStable()` : vraie si la grille courante égale la précédente.
- `hasFinished()` : vrai si stable ou itérations max atteintes.

5.5 Hypothèses et limites du domaine

- Pointeurs bruts dans `Cell` (fragilité potentielle, pas de smart pointers).
- Coût : deux copies de grille par `step()` (précédente + prochaine).
- Règle unique par simulation (pas de règles multiples en parallèle).

6 Couche Application (application/)

6.1 Configuration

- `SimulationConfig` : `inputFile`, `outputBaseName`, `maxIterations` (défaut 100), `toroidal`, `graphicMode`.

6.2 Ports (interfaces)

- `IGridLoader` : `loadGrid(path, toroidal)` (chargement initial).
- `IGridExporter` : `exportGrid(grid, baseName, iteration)` (persistance d'itérations).

6.3 Façade `SimulationService`

- Construit `GameOfLife` via un `IGridLoader` et une `Rule` (fallback `ConwayRule`).
- `step()` : délègue à `GameOfLife`, déclenche `IGridExporter` si présent (pour la console).
- Expose `currentGrid()`, `isStable()`, `hasFinished()`, `currentIteration()`.

6.4 Dépendances

- Dépend du domaine (grilles, règles) et des interfaces de chargement/export.
- Ne dépend pas de SFML ou d'I/O concrètes.

7 Couche Infrastructure (infrastructure/)

7.1 Chargement

- `InitialStateLoader` : lit un fichier texte (première ligne `rows cols`, puis matrice 0/1), construit la `Grid`.
- `FileGridLoader` : implémente `IGridLoader` en déléguant à `InitialStateLoader`.
- Validation stricte : exceptions `std::runtime_error` si fichier introuvable ou format invalide.

7.2 Export

- `GridExporter` : écrit la grille au format texte (dimensions + matrice 0/1) dans `<base>_<iter>.txt`.
- `FileGridExporter` : implémente `IGridExporter` en déléguant à `GridExporter`.

7.3 Dépendances

- Dépend des ports (interfaces) et des types domaine (`Grid`) pour sérialiser/désérialiser.
- Pas de dépendance SFML.

8 Couche Présentation (ui/)

8.1 Console

- `ConsoleRunner` : crée `FileGridLoader`/`FileGridExporter`, instancie `SimulationService`, boucle jusqu'à `hasFinished()`, exporte chaque itération.
- CLI : `console <input> <baseName> [maxIter]`.

8.2 Graphique (SFML)

- `GraphicRunner` : menu SFML (titre, champs fichier/itérations, boutons start/quit), fond animé (grille scintillante), overlay d'itération.
- Contrôles : Espace (pause), N (pas à pas), fermeture fenêtre.
- CLI : `graphic [input] [maxIter]` (défauts sinon).
- Utilise `SimulationService` sans exporter (affichage direct de la grille courante).

8.3 Point d'entrée

- `ui/main.cpp` : parse arguments, construit `SimulationConfig`, redirige vers console ou graphique.

9 Format des données

- Entrée : première ligne `<rows> <cols>`, puis `rows` lignes de `cols` entiers (0/1) séparés par des espaces.
- Sortie (console) : même format, fichier par itération `<base>_<n>.txt`.

10 Flux d'exécution

1. L'UI lit la configuration (arguments, champs du menu).
2. L'UI crée les implémentations des ports (chargement/export) et instancie `SimulationService`.
3. `SimulationService` charge la grille initiale, crée `GameOfLife`.
4. Chaque `step()` applique la règle, incrémente l'itération, exporte si applicable.
5. Arrêt si `isStable()` ou itération max atteinte (console), ou boucle tant que la fenêtre est ouverte (graphique).

11 Tests

- `tests/main.cpp` : harness léger (expect/log) pour bloc stable, blinker, cellule isolée.
- `make test` : construit `tests/test_runner` (pas de SFML).
- Sortie attendue : cas listés avec `OK`, puis "All tests passed.".

12 Conventions et erreurs

- Erreurs de chargement/export : exceptions `std::runtime_error`.
- Iterations max : `SimulationConfig::maxIterations` (défaut 100), peut être surchargé via CLI.
- Mode torique : `toroidal` (désactivé par défaut) pris en compte dans `Grid::countAliveNeighbors`.

13 Analyse SOLID et limites

- SRP : respecté par modules séparés (règle, grille, UI, I/O, service).
- OCP/DIP : ports pour loader/exporter, règle injectée ; fallback concret `ConwayRule` reste couplé par défaut.
- LSP/ISP : interfaces fines pour loader/exporter ; UI dépend directement de la grille pour le rendu.
- Limites : pointeurs bruts dans `Cell`, copies coûteuses de grilles à chaque `step`, dépendance UI sur le modèle (pas de DTO).

14 Pistes d'amélioration

- Smart pointers ou états immuables pour `Cell`.
- Double buffer sans clone par cellule pour accélérer `step()`.
- Port `IClock` pour gérer le temps dans l'UI graphique.
- DTO/vues pour limiter l'exposition du domaine à l'UI.
- Tests supplémentaires : mode torique, motifs complexes, erreurs de parsing, intégration UI (sans SFML via mocks).

15 Synthèse des fichiers

- **domain/** : `CellState.h`, `AliveState.h`, `DeadState.h`, `Cell.h`, `Rule.h`, `ConwayRule.h`, `Grid.h/.cpp`, `GameOfLife.h/.cpp`.
- **application/** : `SimulationConfig.h`, `IGridLoader.h`, `IGridExporter.h`, `SimulationService.h/.c`
- **infrastructure/** : `InitialStateLoader.h/.cpp`, `GridExporter.h/.cpp`, `FileGridLoader.h`, `FileGridExporter.h`.
- **ui/** : `main.cpp`, `ConsoleRunner.cpp/.h`, `GraphicRunner.cpp/.h`.
- **tests/** : `main.cpp` (runner).
- **docs/** : `architecture_layered.tex`, `tests_explained.tex`, `codebase_overview.tex`.