Jeu de la vie — Architecture en couches

Projet C++17

1 Objectif

Décrire l'architecture en quatre couches adoptée pour l'application Jeu de la vie, en séparant clairement la logique métier, l'application, l'infrastructure et la présentation.

2 Vue d'ensemble

- **Domaine (domain/)** : logique pure, modèles et règles du jeu de Conway. Pas de dépendances I/O ou UI.
- **Application (application/)** : orchestration des cas d'usage. Ports (interfaces) et façade de service.
- **Infrastructure (infrastructure/)** : implémentations techniques des ports (accès fichiers, export).
- **Présentation (ui/)** : interfaces utilisateur (console, SFML), dépend de l'application, pas de logique métier directe.

3 Couches en détail

3.1 Domaine

- `CellState` (abstraite), `AliveState`, `DeadState` : états clonables d'une cellule.
- `Rule` (stratégie) et `ConwayRule` : règle B3/S23 (naissance sur 3 voisins, survie sur 2 ou 3).
- `Cell` : encapsule position et état.
- `Grid` : stockage 2D, comptage des voisins, comparaison.
- `GameOfLife` : exécute les étapes, conserve l'itération courante, détecte la stabilité.

3.2 Application

- Ports **IGridLoader** et **IGridExporter** : interfaces pour charger et exporter une grille.
- `SimulationService` : façade qui orchestre une simulation (charge la grille, instancie `GameOfLife`, exécute `step()`, exporte si un exporter est fourni).

- `SimulationConfig` : paramètres d'exécution (fichier d'entrée, itérations max, torique, etc.).
- Dépend uniquement du domaine et des interfaces (pas de dépendance concrète à l'I/O ou à SFML).

3.3 Infrastructure

- `FileGridLoader` (wrap `InitialStateLoader`) : implémente **IGridLoader** pour lire une grille depuis un fichier texte.
- `FileGridExporter` (wrap `GridExporter`) : implémente **IGridExporter** pour écrire une grille vers un fichier texte.
- Ces classes concrètes réalisent les ports définis dans l'application.

3.4 Présentation

- `ConsoleRunner`, `GraphicRunner`, `ui/main.cpp`.
- Créent et injectent les implémentations infrastructure dans `SimulationService`.
- Aucun accès direct aux détails du domaine, uniquement via la façade applicative.

4 Dépendances

- Domaine : n'importe aucune autre couche.
- Application : dépend du domaine et des interfaces (définies dans application/).
- Infrastructure : dépend des ports (application) et des types du domaine qu'elle sérialise (`Grid`).
- Présentation : dépend de l'application (façade/ports) et peut référencer des types du domaine pour l'affichage (lecture seule du `Grid`).

5 Organisation des fichiers

- **domain**/ : `CellState.h`, `AliveState.h`, `DeadState.h`, `Cell.h`, `Grid.h/.cpp`, `Rule.h`, `ConwayRule.h`, `GameOfLife.h/.cpp`.
- **application**/ : `SimulationConfig.h`, `IGridLoader.h`, `IGridExporter.h`, `SimulationService.h/.`
- **infrastructure**/ : `InitialStateLoader.h/.cpp`, `GridExporter.h/.cpp`, `FileGridLoader.h`, `FileGridExporter.h`.
- **ui**/ : `main.cpp`, `ConsoleRunner.cpp/.h`, `GraphicRunner.cpp/.h`.

6 Flux d'exécution

1. L'UI construit un `SimulationConfig` et des implémentations de ports (`FileGridLoader`, `FileGridExporter`).
2. Elle instancie `SimulationService` avec ces dépendances (et éventuellement une `Rule`).
3. `SimulationService` charge la grille, crée `GameOfLife`, expose `step()`, `currentGrid()`, `isStable()`, `hasFinished()`.
4. Chaque `step()` applique la règle et déclenche un export si un exporter est fourni (console).
5. L'UI affiche/contrôle en se basant sur la façade (pas de logique métier dans l'UI).

7 Avantages

- Séparation nette des responsabilités : logique pure vs. I/O vs. présentation.
- Testabilité accrue : domaine testé sans I/O ; application testable via mocks des ports.
- Évolutivité : ajouter un autre loader/exporter (réseau, base de données) sans toucher au domaine ni à l'UI.
- Maintenabilité : l'UI reste focalisée sur l'expérience utilisateur (console/SFML), la logique reste dans le domaine.

8 Prochaines étapes

- Ajouter des interfaces pour le timing (pas de temps) si besoin (`IClock`).
- Étendre les tests application avec des mocks `IGridLoader/IGridExporter` pour vérifier l'orchestration.
- Documenter la DI (injection des implémentations dans l'UI) et envisager un conteneur léger si le projet grandit.