ChargePoint Take Home Assignment

Game of Life

Software Design Document

Revision Date	Reviewed By	Comments
16/02/2025	Arshdeep Singh	Created design document

TABLE OF CONTENTS

1.0 INTRODUCTION

1.1 Project Name

Game of Life

1.2 Purpose

The Game of Life is a simple simulation of how cells live, die, or multiply based on a set of rules. It starts with an initial pattern and evolves over multiple generations. This project builds a console-based version of the game, handling the logic, user input, and display of the grid dynamically.

1.3 Scope

This project focuses on implementing Game of Life algorithm, as a console-based application. It includes functionalities for initializing the game grid, running the simulation for a specified number of generations, and displaying the evolving patterns. The design follows a modular approach with separate components for game logic, user input handling, and display. Future enhancements may include a GUI version or additional game variations.

2.0 HIGH LEVEL ARCHITECTURE

The **Game of Life** project is designed in a structured and modular way to keep things clean, maintainable, and easy to extend. The architecture is divided into different layers, each handling a specific responsibility.

1. **Main Layer (main)**
   - This is the entry point of the application.
   - Application: Starts the game by setting up the required components.

2. **Game Logic Layer (game)**
   - This layer contains the core logic of the game.
   - Game: An interface that defines how any game should behave (start, run, stop).
   - GameOfLife: Implements the Game interface and contains the logic to evolve generations based on rules.
   - GameType: Used to define different types of games.

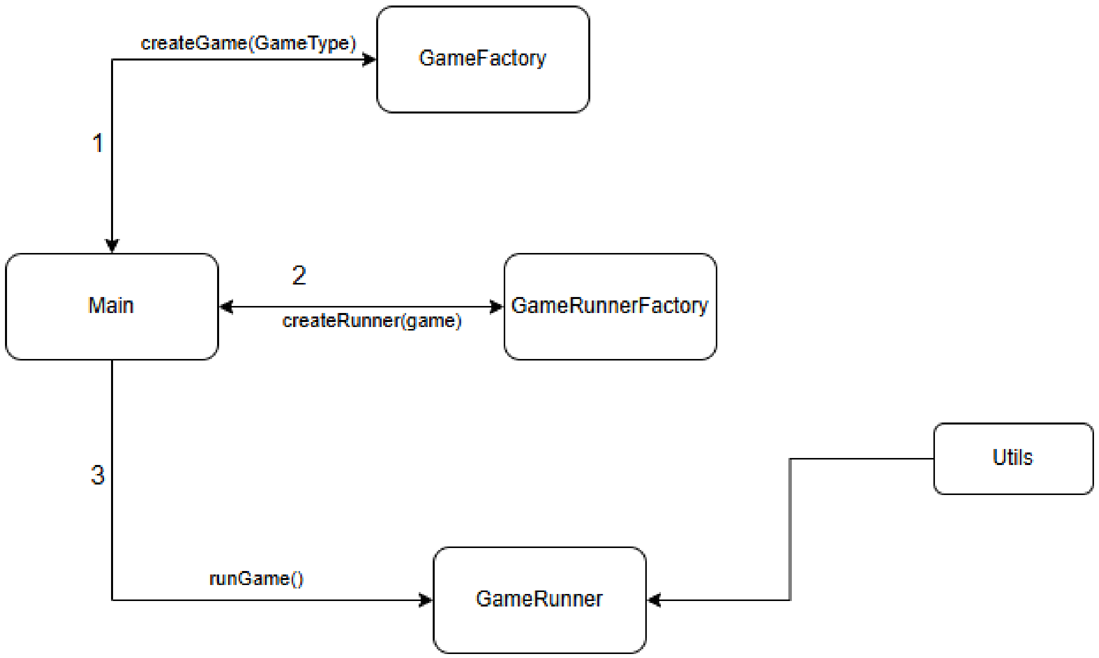
3. **Game Runner Layer (gamerunner)**
   - Responsible for managing and running the game.
   - GameRunner: An interface to define how a game should be executed.
   - GameOfLifeRunner: Implements GameRunner to handle game execution.

4. **Factory Layer (factory)**
   - Helps in creating objects dynamically without tightly coupling the code.
   - GameFactory: Responsible for creating instances of different Game implementations.
   - GameRunnerFactory: Creates instances of GameRunner.

5. **Utility Layer (util)**
   - A set of helper classes for various operations.
   - GameOfLifeUtils: Generates the initial Glider pattern for the game, Manages how the game board is printed to the console.
   - MatrixUtils: Provides matrix-related functions like deep copying.
   - InputUtils: Handles user input validation.

6. **Constants Layer (constants)**
   - Stores commonly used constants.
   - Constants: Holds values like grid size, delay times, and symbols used in the game.

3.0 FLOW OF EXECUTION



4.0 CLASS DESIGN AND RESPONSIBILITIES

The project follows a **modular design**, with each class handling a specific responsibility. This keeps the codebase clean, maintainable, and easy to extend.

1. Game (Interface)

- Defines the core structure of a game.
- Any game implementation (like **Game of Life**) must implement start(), run(), and stop().
- Promotes flexibility for future game variations.

2. GameOfLife (Implementation of Game)

- Implements **Game of Life** logic.
- Initializes the grid, runs generations, and applies game rules.
- Uses helper utility classes for **matrix operations** and **neighbour calculations**.

3. GameRunner (Interface)

- A generic interface to **run any game**.
- Decouples game logic from execution, allowing different game runners.

4. GameOfLifeRunner (Executes GameOfLife)

- Ensures only GameOfLife is run.
- Calls start(), run(), and stop() on the game.

5. Utility Classes (Helper Functions)

GameOfLifeUtils

- Generates an initial **Glider pattern** for the game board.
- Calculates **live neighbors** for each cell.
- Handles **grid display in the console** using symbols for alive and dead cells.

MatrixUtils

- Provides a **deep copy function** for the 2D grid to avoid modifying original data.

InputUtils

- Takes and **validates user input** (e.g., grid size, generations).
- Prevents invalid values (negative numbers, non-integer inputs).

6. Constants (Static Values)

- Stores fixed values like **neighbor positions, symbols, and sleep delays**.
- Centralized constants make updates easier.

5.0 KEY DESIGN DECISIONS

1. **Modular Design**
   The system is divided into multiple components such as Game, GameRunner, and Utils to ensure separation of concerns and better maintainability.

2. **Factory Pattern for GameRunner, Game**
   A GameRunnerFactory is used to create the appropriate GameRunner for a given Game, making the system more extensible in case additional game types are introduced in the future.

3. **Utility Classes for Reusability**
   Common functionalities like matrix operations (MatrixUtils), game board initialization (GameOfLifeUtils), and input handling (InputUtils) are moved to separate utility classes to keep core game logic clean and reusable.

4. **Thread Sleep Handling in Game Execution**
   The game introduces delays using Thread.sleep() to control the speed of simulation updates, enhancing user experience. Exception handling is in place to avoid abrupt crashes.

5. **Constants for Maintainability**
   Game-related constants such as symbols, sleep delay, and grid configurations are placed in a dedicated Constants class to make future changes easier without modifying multiple parts of the code.

6. **Prevention of Utility Class Instantiation**
   All utility classes have private constructors to prevent instantiation, following best practices for static helper classes.

7. **Simplified Game Interface**
   The Game interface enforces a standard structure (start(), run(), stop()) for any game implementation, ensuring consistency and making it easy to add new games in the future.

6.0 FUTURE ENHACEMENTS

Right now, the game is a **console-based application** because of time constraints. However, there were three possible output options considered:

1. **Console-based (Current Choice)** : Chosen for simplicity and faster development.
2. **Swing-based UI** : A Java Swing graphical version for a more interactive experience.
3. **Web-based UI (React, etc.)** : A modern, user-friendly version with a web-based frontend.

In the future, we can expand this project by adding more features:

1. **User-defined Patterns** :
   Instead of always starting with a Glider, let users create their own initial board setup, either through command-line input, a file, or even a graphical editor.

2. **Game Speed Control** :
   Right now, the simulation runs at a fixed speed. We can allow users to **adjust speed dynamically** or even pause and resume the game.

3. **Better Performance** :
   Right now, the whole grid is stored in memory. We can optimize this by only keeping track of live cells to handle **larger grids efficiently**.

4. **Saving & Resuming Games** :
   Let users save a game state and continue later. This could be stored in a **file (JSON/XML) or even a database**.

5. **Logging & Debugging** :
   Add logs for tracking each step of the simulation and debugging tools for analysis.