CONWAY'S GAME OF LIFE

Object-Oriented Concepts II project report

Submitted By

Namisa Najah Raisa

BSc in Software Engineering
Dept of Computer Science and Engineering

January 20, 2024

January 20, 2024

Contents

1 Preface

1.1 The project title

Before going in the project we need to know about "Conway's Game of Life".

The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is a zero-player game,meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves. It is Turing complete and can simulate a universal constructor or any other Turing machine.

1.2 About SOLID

What is SOLID?Why do we need it?

It's a popular set of design principles used in object-oriented software development. SOLID is an acronym for five key design principles:

1. **Single Responsibility Principle**:A class should have only one reason to change, meaning it should have only one responsibility.
2. **Open/Closed Principle**:Software entities (classes, modules, functions) should be open for extension but closed for modification.
3. **Liskov Substitution Principle**:Subtypes should be substitutable for their base types without altering the correctness of the program.
4. **Interface Segregation Principle**:A class should not be forced to implement interfaces it does not use. It promotes the creation of small, specific interfaces.
5. **Dependency Inversion Principle**:High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.

All five are commonly used by software engineers and provide important advantages to developers, hence their importance for this area of work.Without good design principles, software becomes rigid, brittle, and immobile. SOLID principles were developed to combat these problematic design patterns.

1.3 The objective of SOLID principles

The overall goal of SOLID principles is to reduce dependencies so that engineers can change one area of the software without affecting others.

In addition, they are intended to make designs easier to understand, maintain and extend. Ultimately, using these design principles makes it easier for software engineers to avoid problems and create software that is adaptable, efficient, and agile.

Although the principles have many advantages, following them often leads to writing longer and more complex code. This means that it can lengthen the design process and make development a bit more difficult. However, this extra time and effort is worth the work because they make the software much easier to maintain, test, and extend.

The principles have become popular because, if followed correctly, they improve code readability, maintainability, design patterns and testability.

2 UML diagram and explanation

2.1 UML before applying SOLID

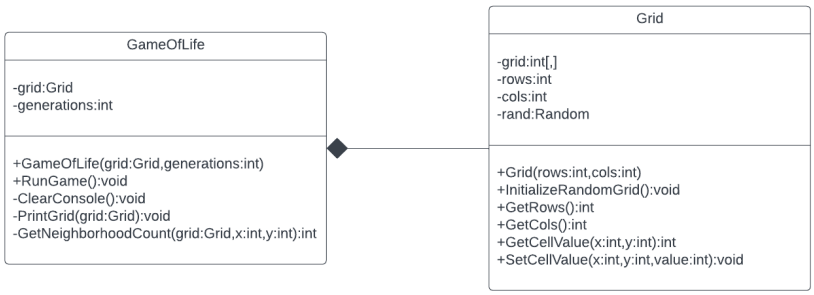

Fig:UML diagram for the version of the project before applying SOLID

In the version of the before applying SOLID principles the principles that it violates:

1. **Single Responsibility Principle**
    - The GameOfLife class is responsible for managing the game state (evolving generations) and interacting with the console to display the grid.
    - Violation: This class has multiple responsibilities, violating the SRP. It would be more maintainable and modular if these responsibilities were split into separate classes.
2. **Open-Closed Principle**

- The GameOfLife class does not provide a clear extension mechanism for modifying or adding game rules.
- Violation: It is not easily extensible without modifying existing code, violating the OCP.

3. **Dependency Inversion Principle**
   - The GameOfLife class directly depends on the Grid class, which is a concrete implementation.
   - Violation: It would be better to depend on abstractions (e.g., an interface) rather than concrete implementations, violating the DIP.

2.2 UML after applying SOLID

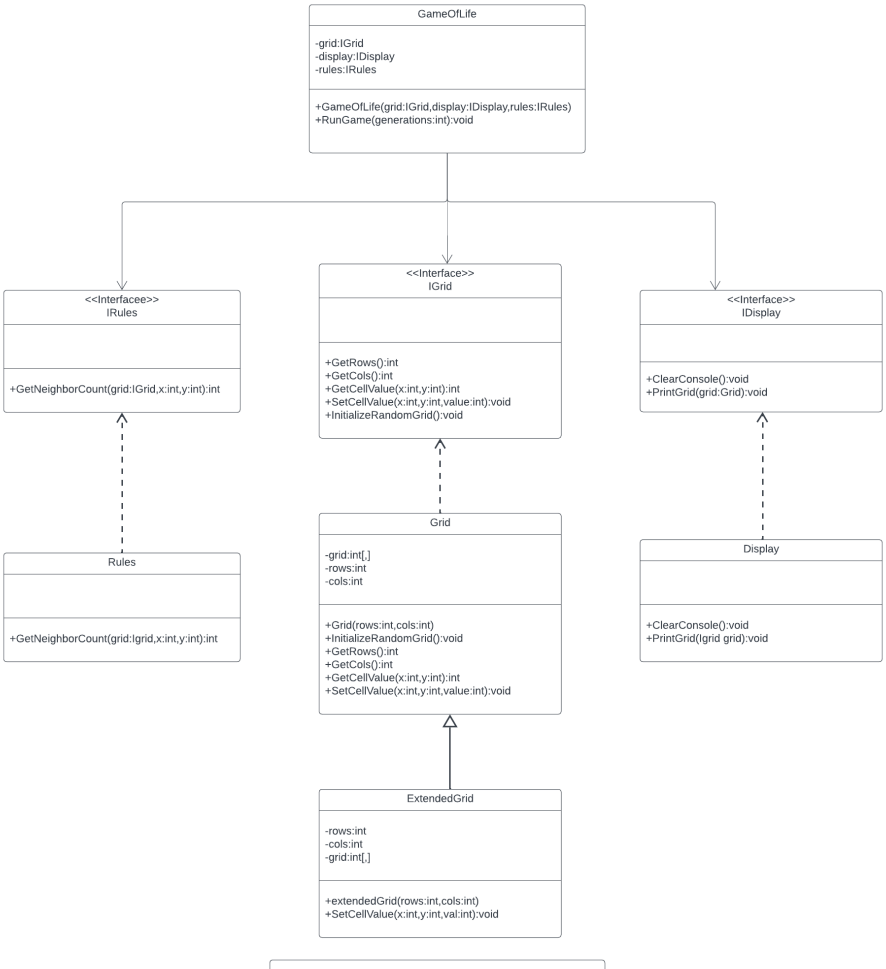

Fig:UML diagram for the version of the project after applying SOLID

In the version after refactoring the project with SOLID principles it's a much more efficient and more readable.

1. **Single Responsibility Principle**

   - **Display class:** The Display class is responsible for console display operations.Its primary function is to handle the display of the grid on the console.The class focuses solely on the presentation concerns related to displaying the grid.It doesn't concern itself with the game logic, grid representation, or rule application.
   - **ExtendedGrid class:** Represents an extended version of the grid with additional functionality.Inherits from the base Grid class, providing a specialized grid.Concentrates on the representation and behavior specific to an extended grid.Additional functionality or changes related to the extended grid are encapsulated here.Modifications to the extended grid behavior do not affect the base grid functionality.Allows for easy extension without altering the existing grid implementation.
   - **GameOfLife class:** Manages the game logic, including initializing the grid, applying rules, and displaying the grid.Orchestrates the overall flow of the Conway's Game of Life simulation.Encapsulates the entire game logic within the class.Changes to the game rules or display don't require modifications in other parts of the code.Provides a clear separation of concerns between game logic and other aspects of the application.Easier to maintain and extend since modifications are isolated within the GameOfLife class.

2. **Open-Closed Principle**

   - **ExtendedGrid class:** The ExtendedGrid class is designed to be open for extension, meaning it allows additional functionality specific to ExtendedGrid without modifying existing code.Developers can add new methods or behavior to ExtendedGrid without altering the behavior of the existing Grid class, following the principle of "open for extension."

     Suppose you want to add a method to perform a special operation on the extended grid. You can easily add this method to the ExtendedGrid class without modifying the existing Grid class or affecting its functionality.
   - **GameOfLife class:** The GameOfLife class adheres to the Open-Closed Principle by being open for extension. It can work with any class implementing the IGrid, IDisplay, and IRules interfaces without modifying its code.The class is not tightly coupled to specific implementations but relies on abstractions, allowing for flexibility in extending or replacing components.If there's a need to introduce a

new type of display or grid with different rules, it can be accomplished by creating new classes implementing the required interfaces without modifying the existing GameOfLife class.

Code stability: The existing code in ExtendedGrid or GameOfLife remains unchanged when new features are added.Encourages the creation of modular and easily extendable software.

3. **Liskov Substitution Principle**

   - **ExtendedGrid class:** ExtendedGrid inherits from the base class Grid.Objects of type ExtendedGrid can be used wherever objects of type Grid are expected.Instances of ExtendedGrid can be substituted for instances of Grid without altering the behavior of the program. This is because ExtendedGrid inherits the fundamental grid-related functionality from its base class.Code that expects an object of type Grid can seamlessly work with an object of type ExtendedGrid because ExtendedGrid is a subtype of Grid.

     For example, if there is a method that takes a Grid parameter, you can pass an instance of ExtendedGrid to that method, and it should behave as expected.
     Enhances code reusability: Code that works with the base class (Grid) can also work with its derived class (ExtendedGrid) without modification.
     Promotes polymorphism: Subtypes can be used interchangeably, allowing for more flexible and extensible code.

     If you have a function that works with a generic grid (Grid), you can use an ExtendedGrid object without modifying that function. This ensures that substitutability is maintained.

4. **Interface Segregation Principle**

   - **IDisplay, IGrid, IRules interfaces:** Each interface has a specific and minimal set of methods related to its responsibility. No class is forced to implement unnecessary methods.

     Each interface contains a minimal set of methods, focusing on a specific responsibility.Classes implementing these interfaces are not burdened with unnecessary methods.Classes can implement only the interfaces relevant to their functionality.For instance, a class dealing with grid display doesn't need to implement grid operation methods.

     No class is forced to implement methods it doesn't need, reducing the chances of unnecessary or irrelevant code in a class.

     Suppose you have a class specifically focused on grid operations but not concerned with displaying the grid. This class can implement

only the IGrid interface without being forced to implement IDisplay methods.

5. **Dependency Inversion Principle**

   - **GameOfLife class:** The GameOfLife class depends on abstractions by using the interfaces IGrid, IDisplay, and IRules. This is evident in its constructor where it receives instances of these interfaces.By depending on abstractions, the GameOfLife class is decoupled from specific implementations of IGrid, IDisplay, and IRules.This decoupling allows for flexibility in swapping implementations without modifying the GameOfLife class.

     For example, you can easily replace a concrete class that implements IGrid with another as long as it adheres to the same interface.

     Consider the instantiation of GameOfLife in the Program class:

```
IGrid grid = new ExtendedGrid(rows, cols); //
   Using the ExtendedGrid class
IDisplay display = new Display();
IRules rules = new Rules();
GameOfLife game = new GameOfLife(grid, display,
    rules);
```

     Here, specific implementations (ExtendedGrid, Display, Rules) are created and passed as abstractions (IGrid, IDisplay, IRules) to the GameOfLife constructor.

     The GameOfLife class is not tightly coupled to concrete implementations, making it easier to change or extend individual components without affecting the overall structure.

     Changes to the internal implementations of IGrid, IDisplay, or IRules do not necessitate modifications to the GameOfLife class.

     If a new grid representation or display method is developed, it can be seamlessly integrated into the GameOfLife simulation by creating a class that implements the corresponding interface without modifying the existing game logic.

3 Avoided code and design smells

1. **Rigidity:**

   - Rigidity occurs when a small change in the system requires modifications to many classes.

- The use of interfaces (IDisplay, IGrid, IRules) and dependency injection in the GameOfLife class helps in avoiding rigidity. Changes to implementations can be accommodated without modifying the GameOfLife class.

2. **Fragility:**
   - Fragility is the tendency of the system to break in many places due to a single change.
   - The separation of concerns and the use of interfaces contribute to a less fragile design. Each class has a specific responsibility, reducing the impact of changes.

3. **Immobility:**
   - Immobility occurs when it's challenging to reuse components in other systems.
   - The adherence to SOLID principles, particularly the Dependency Inversion Principle, enables the flexibility to replace implementations with minimal impact, promoting reusability.

4. **Viscosity:**
   - Viscosity refers to the resistance to change, making it easier to do the wrong thing.
   - The clear separation of concerns in classes (Display, ExtendedGrid, GameOfLife, Grid, Rules) and adherence to SOLID principles contribute to a design that is less resistant to change.

5. **God Class:**
   - A class that does too many things or has too much responsibility.
   - Each class (Display, ExtendedGrid, GameOfLife, Grid, Rules) has a clear and specific responsibility, adhering to the Single Responsibility Principle.

6. **Feature Envy:**
   - A class that uses methods of another class excessively, possibly indicating misplaced responsibilities.
   - By following SOLID principles, especially the Single Responsibility Principle, Feature Envy is minimized as each class is responsible for its own concerns.

7. **Shotgun Surgery:**
   - A situation where a single change requires modification in multiple places.

- The design, with its use of interfaces and abstractions, minimizes the impact of changes, preventing shotgun surgery.

8. **Data Clumps:**
   - Occurs when groups of variables are always used together.
   - The design of classes (Grid, ExtendedGrid) encapsulates related data, preventing data clumps and promoting encapsulation.

4 Github repositories(redirected links)

1. Project repository
2. The inspiration for this project

5 References used throughout the report(redirected links)

1. Conway's Game of Life information
2. Conway's game of life simulation
3. SOLID principles
4. Lucidchart:used for drawing the UML diagrams

...The End...