# IFN 563 Object Oriented Design

## Assessment 2

## Final Design and Implementation

## Submitted by

## 

|  |  |
| --- | --- |
| SCOTT BILAU | N11916095 |
| NORA JOHN | N11890223 |

## **Table of Contents**

## Overview……………………………………………………………………………………

## Principles and patterns…………………………………….…………………………

## Class Diagram………………………………………………….…………………………

## Sequence Diagram…………………………………………...…………………………

## Object diagram………………………………………………...…………………………

## Program Execution Guide………………………………….…………………………

## Declaration……………………………………………………...…………………………

## Overview

The final design streamlines implementation and addresses the challenges of developing a console-based multi-board game by incorporating feedback from the preliminary design. The program is organized into distinct classes that represent key game elements, such as players, boards, game states, and turn management. This structure also integrates features like saving and loading game states, managing moves through stacks, and displaying results on a leader-board.

In the preliminary design, a single Board class was used to represent any type of board game, with a Rule class checking the validity of moves. However, this approach was overly abstract and difficult to implement, as each game has unique characteristics requiring distinct classes. To address this, the new design includes child classes under the Board class, each tailored to a specific game.

Additionally, the new design introduces a separate GameState class to manage the functionality of pausing and resuming games. This class includes methods for saving the game state to an external file and resuming from where the game left off . . Central to the architecture is the **Game** class, which manages the overall game flow, including starting the game, handling player turns, checking for game-over conditions, and managing commands like undo and redo. The **Board** class represents the game board, encapsulating board-related operations such as displaying the board, checking for empty positions, and determining win conditions. Players are represented by the **Player** class, an abstract class that provides a common interface for different player types, such as **HumanPlayer** and **ComputerPlayer**.

This design results in a robust, modular, and scalable game system that can easily be extended with new player types, game variations, or additional features.

**OOP design patterns and principles used in this implementation:**

* **Abstraction**:
  + The Player class is abstract, providing a common interface for different player types.
  + Why: It allows for different implementations of players while ensuring they all have the necessary methods.
* **Encapsulation**:
  + Each class encapsulates its data and behavior.
  + Why: It helps in maintaining the integrity of the data and provides a clear interface for interaction.
* **Inheritance**:
  + HumanPlayer and ComputerPlayer inherit from the Player class.
  + Why: It promotes code reuse and allows for specialized implementations of different player types.
* **Polymorphism**:
  + The GetMove method is implemented differently in HumanPlayer and ComputerPlayer.
  + Why: It allows for different behavior based on the player type while using a common interface.
* **Single Responsibility Principle**:
  + Each class has a single, well-defined responsibility (e.g., Board manages the game board, Game manages the game flow).
  + Why: It makes the code more modular, easier to understand, and easier to maintain.
* **Open/Closed Principle**:
  + The design allows for easy addition of new player types or game variations without modifying existing code.
  + Why: It makes the system more extensible and reduces the risk of introducing bugs in existing functionality.
* **Strategy Pattern**:
  + Different player types (Human and Computer) implement different strategies for making moves.
  + Why: It allows for easy swapping of player types and addition of new player strategies.
* **Composition**:
  + The Game class composes Board, Player, and Piece objects.
  + Why: It creates a flexible structure that can be easily modified or extended.
* **Factory Method (simplified)**:
  + The Game class creates player objects based on user input.
  + Why: It centralizes object creation and allows for easy modification of the creation process.
* **Iterator Pattern (implicit)**:
  + The use of List<Board> and iterating over it in various methods.
  + Why: It provides a standard way to access elements of the collection without exposing its underlying structure.

These patterns and principles work together to create a flexible, maintainable, and extensible design for the Notakto game. They allow for easy modifications, such as adding new player types or changing game rules, while keeping the core structure intact.

---

Here's an analysis of the design principles and patterns used:

**Design Principles**

* **Single Responsibility Principle (SRP)**:
  + Each class has a specific responsibility. For example, NotaktoBoard handles board-related operations, Player manages player actions, and Game controls the game flow.
* **Open/Closed Principle (OCP)**:
  + The Player class is abstract and can be extended (e.g., HumanPlayer and ComputerPlayer) without modifying existing code.
* **Liskov Substitution Principle (LSP)**:
  + Subclasses like HumanPlayer and ComputerPlayer can be used interchangeably with the Player base class.
* **Interface Segregation Principle (ISP)**:
  + While not explicitly using interfaces, the classes have focused methods that align with this principle.
* **Dependency Inversion Principle (DIP)**:
  + High-level modules (like Game) depend on abstractions (like Player) rather than concrete implementations.

**Design Patterns**

* **State Pattern**:
  + Participating Classes: GameState, Game
  + Important Operations: PauseGame(), ResumeGame()
  + Justification: Manages the game's state (paused, resumed) and allows for easy state transitions.
* **Strategy Pattern**:
  + Participating Classes: Player (abstract), HumanPlayer, ComputerPlayer
  + Important Operations: GetMove()
  + Justification: Allows for different player strategies (human input vs. computer algorithm) to be interchangeable.
* **Command Pattern**:
  + Participating Classes: Move, Game
  + Important Operations: Undo(), Redo()
  + Justification: Encapsulates move actions as objects, allowing for easy undo/redo functionality.
* **Template Method Pattern**:
  + Participating Classes: Board (abstract), NotaktoBoard
  + Important Operations: PlacePiece(), CheckForWin()
  + Justification: Defines the skeleton of board operations in the abstract class, with specific implementations in the concrete class.
* **Singleton Pattern** (implied):
  + Participating Classes: Board (with static AllBoards list)
  + Important Operations: AddBoard(), ResetAllBoards()
  + Justification: Maintains a single collection of all boards across the game.
* **Memento Pattern** (partially implemented):
  + Participating Classes: GameState, Game
  + Important Operations: SaveGame(), LoadSavedGame()
  + Justification: Allows for saving and restoring game state.

**Classes/Interfaces to be Reused from Existing Libraries and Frameworks**

* **System Collections:**
  + List<T>
  + Stack<T>
* **System IO:**
  + File
* **System Text Json:**
  + JsonSerializer
* **System:**
  + Console
  + Random
  + Serializable attribute

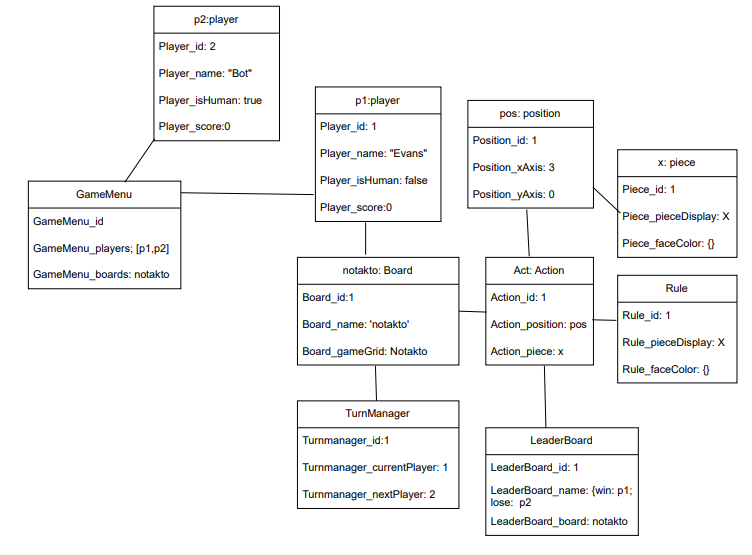
These classes and interfaces are used for data structures, file operations, JSON serialization, console I/O, and random number generation, providing essential functionality for the game implementation without reinventing the wheel.

**Class diagram**

The class diagram is updated with a program design which features a streamlined **Game** class, with some of its responsibilities now shifted to a newly introduced **GameState** class. The **Game** class continues to manage core game functions such as initiating the game, handling player turns, and determining when the game is over. However, state-related operations are now encapsulated within the **GameState** class.The **GameState** class introduces methods like PauseGame(Game game), UpdateGameInstance(), ResumeGame(Game game), LoadSavedGame(Game game), SaveGame(Game game), and CheckGameHistory(). These methods handle the dynamic aspects of game states, including pausing and resuming games, saving and loading game progress, and maintaining a history of game states.

The **Board** class represents the game board, handling the display of the board, checking for empty positions, determining win conditions, and managing the board's state (e.g., marking it as dead if needed). Specialized boards, like **NotaktoBoard** and **GomokuBoard**, inherit from this class and implement specific rules and behaviors for different games. Players in the game are represented by the **Player** class, which is an abstract base class for different player types, such as **HumanPlayer** and **ComputerPlayer**. These subclasses implement the GetMove() method according to the specific logic for human input or AI-driven moves.

**Object diagram**



The diagram represents a modular game system with classes for managing players, the game board, positions, pieces, actions, and rules. Key components include GameMenu for setup, TurnManager for turn management, and LeaderBoard for tracking results. No changes where made to the object diagram.

**Program Execution Guide**

· S**et Up the Development Environment:**

* · Ensure you have .NET installed on your system. If not, download and install it from the [official .NET website](https://dotnet.microsoft.com/download" \t "_new).

· **Download or Clone the Project:** · Obtain the source code of the game. You can download it as a ZIP file or clone it using Git if it's hosted on a version control platform like GitHub.

* 1. Navigate to the project folder using your file explorer or command prompt.
* 2. Once inside the project folder, use the ‘cd’ command to navigate to the root directory of the project. Execute the following command to run the game: ‘ dotnet run ‘

**Starting NOTAKTO**

- After launching the project, choose ‘start game’ to start the game, ‘help’ for instructions,or ‘exit’ to play Notakto, as it is the first game implemented in the project.

-When the game is started select your opponent by entering ‘1’ for a human player or ‘2’ for a computer/Bot opponent.

- Enter your name when prompted.

- Then the board and the current status will be displayed.

- Enter a number less than or equal to the board length to place your piece.

- Use the following keys during your turn:

- ‘SAVE’ to save a move

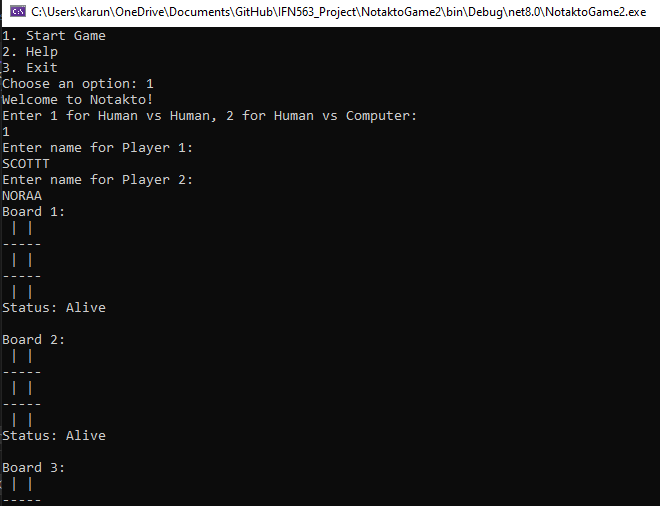
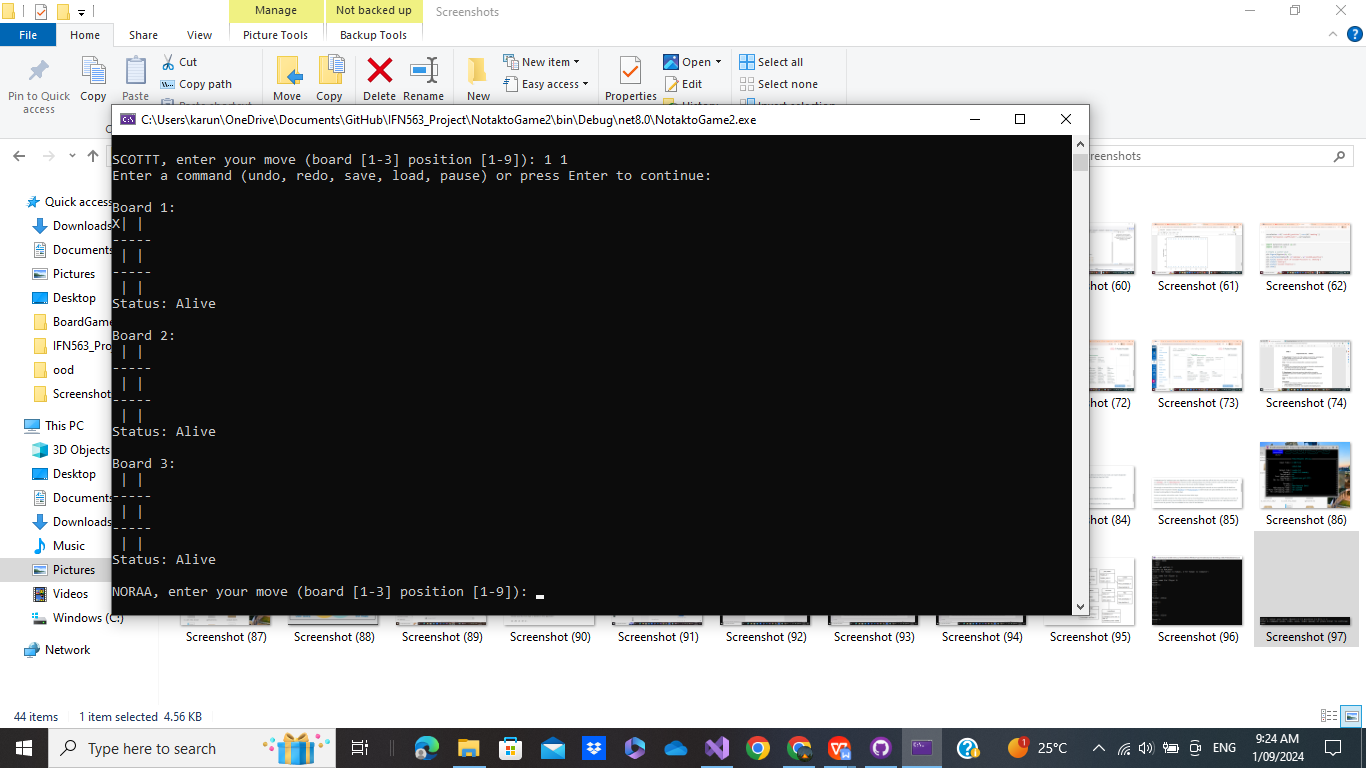
- ‘UNDO’ or ‘REDO’ to undo or redo a move respectively.

- ‘PAUSE to pause the game and ‘LOAD’ to load the game

- Or press ‘ENTER’ to confirm your move.

Let your opponent take their turn. If the opponent is the computer, the move will be made automatically and displayed on the board.- Continue taking turns until the game concludes.

- Finally the winner will be displayed!

Here is a screenshot of the game of Human vs Human

**Declaration**

We, [SCOTT] and [NORA], declare that this assignment was completed collaboratively, with equal participation and contribution from both of us. Each member of the group has actively engaged in all aspects of the assignment. Both members have contributed equally to the development and execution of the assignment. All sources and references used in this assignment have been properly cited, and the work presented is solely our own. Each participant has been involved in all phases of the project, from initial planning to final submission. We acknowledge that any misrepresentation of our contributions could be subject to review under the institution's academic integrity policies.

We communicated in each stage to make it final version. Also discussed the feedback after each part.

Name : Scott Bilau

Student number : N11916095

Name : Nora John

Student number: n11890223