**City University of Hong Kong**

CS3343 Software Engineering Practice

2023/24 Semester A

Analysis and Design

Project Title: Connect Four Game

Group 27

|  |  |
| --- | --- |
| **Name** | **Student Number** |
| Fong Tsz Wai | 57156110 |
| JI Xin | 40147182 |
| Ka Lam Mark Alexander LEE | 56759365 |
| LIN Xiaoyang | 57126676 |
| SONG Rui | 57126559 |

Table of Contents

[1. Design Constraints 3](#_Toc153218843)

[1.1. Compatibility with Windows 10 3](#_Toc153218844)

[1.2. Programming Language Selection 3](#_Toc153218845)

[1.3. Artificial Intelligence (AI) Component 3](#_Toc153218846)

[1.4. Performance Optimization 3](#_Toc153218847)

[1.5. Game Rules 3](#_Toc153218848)

[1.6. Schedule 3](#_Toc153218849)

[2. Requirement Specifications 4](#_Toc153218850)

[2.1. Basic Functions 4](#_Toc153218851)

[2.2. Advanced Functions 4](#_Toc153218852)

[3. Use Case 5](#_Toc153218853)

[3.1. Use Case Diagram 5](#_Toc153218854)

[3.2. Use Case Specification 5](#_Toc153218855)

[4. Class Diagram 7](#_Toc153218856)

[4.1. Use Case Specification 7](#_Toc153218857)

[5. Sequence Diagram 8](#_Toc153218858)

[5.1. Game Loop 8](#_Toc153218859)

[5.2. View Game Record 10](#_Toc153218860)

[6. Design Patterns & Design Principles 11](#_Toc153218861)

[6.1. Design Principles 11](#_Toc153218862)

[6.1.1 Single Responsibility Principle 11](#_Toc153218863)

[6.1.2 Law of Demeter 11](#_Toc153218864)

[6.1.3 Open Closed Principle 11](#_Toc153218865)

[6.2. Design Patterns 12](#_Toc153218866)

[6.2.1 Singleton Pattern 12](#_Toc153218867)

[6.2.2 Command Pattern 13](#_Toc153218868)

[6.2.3 Factory Pattern 14](#_Toc153218869)

# Design Constraints

## Compatibility with Windows 10

* The client specifically requires the program to run on command prompt on his Windows 10 machine.
* The application will be developed and tested to ensure seamless execution on the Windows 10 operating system.

## Programming Language Selection

* Considering the expertise of our team members, Java has been chosen as the programming language for application development.
* Java offers cross-platform compatibility, automatic memory management and object-oriented structure, making it suitable for our project requirements.

## Artificial Intelligence (AI) Component

* Given the presence of AI experts in our team, the AI module will be assigned to those teammates.
* The AI developer will leverage their expertise to implement efficient algorithms and optimize the game's AI functionality.

## Performance Optimization

* A key objective is to build the application to run as efficiently as possible.
* Best coding practices will be followed throughout the development process to ensure optimal performance and code expandability.
* The AI developer will specifically focus on selecting and implementing efficient algorithms to enhance the game's performance.

## Game Rules

* The game rules must be the same as the original Connect4, except the twist the client requested us to complete.

## Schedule

* The client has requested this project to be done in 3 months. The final delivery date has been set and agreed by the client and our members.

# Requirement Specifications

## Basic Functions

* **Connect4 Game Grid**: Implement a game grid for playing Connect4.
* **Human vs Human Mode**: Allow two human players to play against each other.
* **Game Score History**: Keep a record of the game scores for future reference.

## Advanced Functions

* **Human vs AI Mode with two difficulty levels**: Provide an option for a human player to play against an AI opponent with 2 difficulty levels (easy and difficult).
* **Configurable amount of AI suggestions for next move**: Allow the user to customize the number of AI suggestions for the next move.
* **Configurable amount of Undo in each game play**: Enable the user to decide the number of undo moves available during gameplay.
* **Entire Game Replay**: Allow users to replay the entire game, including all moves of each game in game history.
* **AI next move suggestion during Game Replay**: Provide AI-generated suggestions for the next move during game replays.

# Use Case

## Use Case Diagram

|  |
| --- |
|  |
| Fig 3.1 Use Case Diagram of Connect4 |

## Use Case Specification

|  |  |  |
| --- | --- | --- |
| **Use Case ID:** | UC 1000 | |
| **Use Case Name:** | Play Game | |
| **Actor(s):** | Player | |
| **Description:** | This use case represents the main flow of the Connect4 Game. | |
| **Trigger:** | The Player selects “Start New Game” | |
| **Normal Flows:** | **Actor Action** | **System Response** |
| 1. The user runs the game 2. The user enters 1 and presses enter | The game asks for game mode   1. Human vs Human) 2. Human vs AI) 3. AI vs Human)   The game asks for the users Name and Preferred Symbol |
| **Post-conditions:** | The game ends with either a win or draw condition. | |

|  |  |  |
| --- | --- | --- |
| **Use Case ID:** | UC 2000 | |
| **Use Case Name:** | Make Move | |
| **Actor(s):** | Player | |
| **Description:** | This use case represents the action of the player making a move by selecting a column to drop their game piece. | |
| **Trigger:** | The player is prompted to make a move. | |
| **Normal Flows:** | **Actor Action** | **System Response** |
| 1. The player selects a column to drop the coin | The game checks for the validity of the move and shows the new grid where the move is made. |
| **Post-conditions:** | The player's move is successfully made and reflected on the game grid, asks the next player for a move. | |

|  |  |  |
| --- | --- | --- |
| **Use Case ID:** | UC 3000 | |
| **Use Case Name:** | Check Game History | |
| **Actor(s):** | Player | |
| **Description:** | This use case represents the action of checking the game history. | |
| **Trigger:** | The player requests to check the game history. | |
| **Normal Flows:** | **Actor Action** | **System Response** |
| 1. The player selects 2 to Review Game History | The game shows the list of game histories including game score, time played and the score of the players |
| **Post-conditions:** | The game asks for the user if they would like to replay the game | |

|  |  |  |
| --- | --- | --- |
| **Use Case ID:** | UC 4000 | |
| **Use Case Name:** | Configure Game | |
| **Actor(s):** | Player | |
| **Description:** | This use case represents the action of configuring the game settings, including no. of undo chances and no of AI suggestions avalible. | |
| **Trigger:** | The player requests to configure the game. | |
| **Normal Flows:** | **Actor Action** | **System Response** |
| 1. The player selects the desired configurable item | The game Configures the specified items to the new parameters |
| **Post-conditions:** | Back to main menu | |

# Class Diagram

## Use Case Specification

|  |
| --- |
|  |
| Fig 4.1 Full Class Diagram of Connect4 |

Connect4 is comprised of 4 main parts, including command classes, Main Game, Referee, Players and Game Records.

**Command Classes**

Command classes include CmdMove, Cmd, Move and CmdList, and CmdFakeMove. These classes are responsible for handling and recording the details of each command. By using the command pattern in this part of the game, undo, redo and game replay is thus possible. CmdFake move is for the AI Algorithms to test the next best winning move.

**Main Game**

The Main game is comprised of Game Class. The Game class is the heart of each game, it records a CmdList for each game instance, so that the entire game can be recorded.

**Referee**

Referee class is for validating each move and checking winning, draw and grid full scenarios.

**Player**

Player classes include AI players and human players, they include functions to manipulate the connect4 moves, undo and suggest moves.

**Game Record**

Game record is a singleton. Responsible for reading and writing each game record.

# Sequence Diagram

## Game Loop

|  |
| --- |
|  |
| Fig 4.1 Sequence Diagram of gameLoop (Option –1,-2 and –3 and MakeMove) |

During gameplay, players can perform four different actions. They can input -1 to attempt an undo, -2 to request an AI suggestion, -3 to redo a previous move, or any other valid coordinates to specify their own move. Here's how each input is handled:

|  |  |
| --- | --- |
| -1 | Attempt an undo |
| If undo is not allowed, a "canNotUndoException" is thrown. If there are no more undo moves left, a "noMoreUndoException" is thrown. If undo is allowed, the player object invokes the "undocmd" function in the "CmdMove" class.  Print Grid | |

|  |  |
| --- | --- |
| -2 | Request AI Suggestion |
| If AI Suggestion is unavalible, a "noMoreSuggestionException" is thrown. If undo is allowed, invoke BataGo and GamaGo to print their suggestion | |

|  |  |
| --- | --- |
| -2 | Attempt a redo |
| If redo is not allowed, a "canNotRedoException" is thrown. If redo is allowed, the player object invokes the "redoOne" function in the "CmdMove" class.  Print Grid | |

|  |  |
| --- | --- |
| -3 | Attempt a redo |
| If redo is not allowed, a "canNotRedoException" is thrown. If redo is allowed, the player object invokes the "redoOne" function in the "CmdMove" class.  Print Grid | |

|  |  |
| --- | --- |
| Any other valid coordinate index | Attempt a move |
| Create a new CmdMove and add it into the cmdList  Print Grid | |

## View Game Record

|  |
| --- |
|  |
| Fig 4.2 Sequence Diagram of view game record |

In the main menu, users have the option to view Game History. If there are no records available, a "NoGameRecordException" is thrown to indicate the absence of any recorded games. However, if there are records present, the system displays all the game records and prompts the user to choose a specific record for game replay.

When viewing an individual game record, the player can navigate to the next move (2), previous move (1), or request the suggested next move from the AI (0). This AI suggestion provides insights into moves that give a higher chance of winning at any given point in the game replay.

|  |  |
| --- | --- |
| 2 | Next move |
| If it is already last move, print already last move exception  If it isn’t the last move, traverse it. | |

|  |  |
| --- | --- |
| 1 | Previous Move |
| If it is the first move, print already first move exception  If it isn’t the first move, traverse the previous move. | |

|  |  |
| --- | --- |
| 3 | AI Suggestion for next move |
| If it is the last move, show already last move exception.  If not the last move, invoke BetaGo and GamaGo to print the next best move suggestion. | |

# Design Patterns & Design Principles

## Design Principles

### Single Responsibility Principle

This code within this project implements single responsibility principle to ensure that each module or class has a well-defined and specific responsibility. For example, the handling of game win conditions, draw conditions, and move validity are solely handled by the Referee class. This ensures that any issues or bugs related to these aspects can be easily identified and located during the debugging process. In addition, each of the command functions are all handled by their respective command class, this ensures maximum code clarity and maintainability. Additionally, as the project moves forward, new functionality can be easily extended into the code without affecting other parts of the project.

### Law of Demeter

This project implements the Law of Demeter as much as possible by reducing dependencies of each class and eliminating the access of details of other classes. Through the use of dependency injection and setting appropriate visibility levels of each class, encapsulation is achieved and thus promoting lose coupling. During code refactoring, long chains of method calls are identified and refactored to ensure objects have maximum separation.

### Open Closed Principle

This project implements the Open-Closed Principle by encapsulating data and methods within each entity and defining boundaries and responsibilities for each. The entities within the project are designed to be open for extension but closed for modification. For example, the project utilizes abstraction to represent a player. By extending this abstract player, new types of players can inherit the functions of Player without having to modify existing code. This approach enables the addition of new player types without the need to modify the existing code, promoting extensibility and adhering to the Open-Closed Principle. By adhering to this principle, maintainability is enhanced, and new functions and code can be added by extending the original code while minimizing the need to modify existing code.

## Design Patterns

### Singleton Pattern

|  |
| --- |
|  |
| Fig. 5.1 Subset of the connect4 Class Diagram (Game Record) |

This project effectively incorporates the Singleton pattern in relevant areas. Specifically, the GameRecord class is utilized to write and retrieve game histories to and from a text file. To ensure a single global point of access and prevent multiple instances of GameRecord, the class is implemented using the Singleton design pattern. This guarantees that only one instance of GameRecord exists, and it is provided to the client whenever it is needed.

By applying the Singleton pattern, a single and global point of access to GameRecord is maintained. The Singleton design pattern ensures that multiple instances of GameRecord are not created, preventing potential conflicts and ensuring a centralized and controlled access point.

In short, this project effectively applies the Singleton pattern to ensure a single instance of the GameRecord class. By utilizing the Singleton design pattern, the project maintains a centralized and controlled access point for GameRecord, preventing the creation of multiple instances and potential conflicts.

### Command Pattern

|  |
| --- |
|  |
| Fig. 5.2 Subset of the connect4 Class Diagram (Commands) |

This project successfully applies the Command pattern in key areas. As shown in Figure 5.2, all move commands implement the CmdMove interface, ensuring a consistent structure for all types of moves, including the undo method. The executed moves are stored in a cmdList, enabling the recording of game history for each match. The state of each move is effectively recorded, allowing for easy replay at the user's discretion.

By utilizing the Command pattern, the project enables the addition of new commands by extending the command class and implementing the necessary functionality for undo and game replay. This approach enhances flexibility and maintainability, as new commands can be seamlessly integrated into the system without extensive modifications. For example, during phase II of the implementation, it is decided that CmdFakeMove is needed for the AI algorithm to calculate the best next move. CmdFakeMove was quickly added without modifying existing command codes. This successful addition highlights the benefits of using the Command pattern, as it allows for easy integration of new commands while maintaining the integrity and functionality of the existing system.

In summary, the project effectively incorporates the Command pattern to provide a structured framework for different types of moves, enabling undo capabilities and game replay. This design pattern promotes code reusability and extensibility, enhancing the overall functionality and user experience of the project.

### Factory Pattern

|  |
| --- |
|  |
| Fig 5.4 Subset of connect4 Class Diagram (Player) |

This project effectively incorporates the Factory pattern in crucial aspects. As depicted in Figure 5.4, the PlayerFactory is responsible for creating different types of players (humanPlayer, GamaGoAIPlayer, BetaGoAIPlayer) as needed. This centralizes the object creation process, providing a dedicated entity for handling player creation.

By utilizing the PlayerFactory, the project achieves code decoupling and promotes dependency injection. The client no longer needs to directly create player objects but instead relies on the PlayerFactory to handle the creation process. This separation of responsibilities enhances modularity and flexibility.

Furthermore, the implementation of the Factory pattern facilitated the seamless addition of GamaGoAIPlayer and BetaGoAIPlayer during the second phase of game implementation, we refactored the code to implement playerFactory. The PlayerFactory exclusively handles the creation of game players, allowing for easy extension with minimal modification of the existing codebase.

In summary, the project effectively applies the Factory pattern by utilizing the PlayerFactory to centralize player object creation. This promotes code decoupling, dependency injection, and simplifies the addition of new player types. The Factory pattern enhances maintainability, flexibility, and modularity in the project.