Comprehensive Exercise Report

Team <<X>> of Section <<000>>

Wilfried RUIZ, ID = 240ADM002
Guy Leonard KAMGA FOTSO WAFO, ID = 240ADM005
Elias CREMNITER, ID = 240AEB020
Kevin RANDRIANIMANANA, ID = 240AEB027

NOTE: You will replace all placeholders that are given in <<>>

Requirements/Analysis

Week 2

Journal

The following prompts are meant to aid your thought process as you complete the requirements/analysis portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

- After reading the client's brief (possibly incomplete description), write one sentence that describes the project (expected software) and list the already known requirements.
    - Connect 4 is a game which is played by 2 players in which the winner is the player who puts the first 4 discs in the grid.
    - A grid to put the discs in; the grid has a sliding bar in the bottom; red discs and yellow discs; put the disc on top and drop it to let it falling; the players play each their turn; to get first 4 discs successive (row, column or diagonal)

- After reading the client's brief (possibly incomplete description), what questions do you have for the client? Are there any pieces that are unclear? After you have a list of questions, raise your hand and ask the client (your instructor) the questions; make sure to document his/her answers.
    - How many rounds in this game ? How to play if there are more than 2 players ? What could be the consequences for the loser and the awards for the winner ?

- Does the project cover topics you are unfamiliar with? If so, look up the topics and list your references.
    - There aren't any topics we are unfamiliar with.

- Describe the users of this software (e.g., small child, high school teacher who is taking attendance).
    - Everybody can play this game except children who are less than 4 years old.

- Describe how each user would interact with the software
    - On a computer, the players can place their discs by clicking on the desired column; ; players can use controllers to navigate and select the column, then press a button to place their disc

- What features must the software have? What should the software be able to do?
- A graphical interface to allow to players understanding quickly how to play; the players play in the same computer (local interface)

- Other notes:
    - The rules/descriptions are based on the official rules (link of official rules on Ortus)

Software Requirements

<<Use your notes from above to complete this section of the formal documentation by writing a detailed description of the project, including a paragraph overview of the project followed by a list of requirements (see lecture for format of requirements). You may also choose to include user stories.>>

Project Overview:

The project involves implementing the rules of the game Connect Four in a computer environment. Connect Four is a strategic game for two players, where each player takes turns selecting a column to drop a token of their color (red or yellow) into the grid. The goal is to align four tokens of one's own color horizontally, vertically, or diagonally before the opponent does. The game is played on a 6x7 grid. The game ends when the grid is full or when a player successfully aligns four tokens. The player who aligns four tokens first wins the game.

Project Requirements:

1. The game must allow two players to take turns.
2. Players must be able to place their token in a non-full column of the grid.
3. The game must detect and signal when a player aligns four tokens.
4. The game must end when the grid is full or when a player aligns four tokens.
5. The game must clearly display the current state of the grid after each move.
6. Players must be able to start a new game after the end of the previous one.
7. The game must include a user-friendly interface for players to select columns to place their tokens.
8. The game must handle draw cases when the grid is full without any player aligning four tokens.
9. The program must be developed in an appropriate programming language with suitable data structures to represent the grid and tokens.

User Stories:

1. As a player, I want to be able to select a column to place my token so that I can take my turn.
2. As a player, I want to be notified when I align four tokens to win the game.
3. As a player, I want to see the current state of the grid after each move to understand the game's progress.
4. As a player, I want to be able to start a new game at any time to continue playing.
5. As a player, I want the game to automatically detect a draw when the grid is full without any player aligning four tokens.

Black-Box Testing

Instructions: Week 4

Journal

Remember: Black box tests should only be based on your requirements and should work independent of design.

The following prompts are meant to aid your thought process as you complete the black box testing portion of this exercise. Please review your list of requirements and respond to each of the prompts below. Feel free to add additional notes.

- What does input for the software look like (e.g., what type of data, how many pieces of data)?
    - The inputs are the click done with the mouse on the column select number on the screen

- What does output for the software look like (e.g., what type of data, how many pieces of data)?
    - The software will show some evolutions (disc placed) on the screen and text by moment.

- What equivalence classes can the input be broken into?
    - A Disc and player class.

- What boundary values exist for the input?
    - The Column 1 and 7 are the boundaries values for the game board and players can’t place a disc outside.

- Are there other cases that must be tested to test all requirements?
    - Yes, we should assure that if a column is full the player can’t place a disc. The number of turns should not exceed the number of cases on the gameboard. One disc and change turn for the player.

- Other notes:
    - <<Insert notes>>

Black-box Test Cases

Use your notes from above to complete the black-box test plan section of the formal documentation by writing black box test cases (other than actual results since no program currently exists). Remember to test each equivalence class, boundary value, and requirement.

Valid partitions for input:
TCOND1: NbDisc = positive Integers and 0<=NbDisc<=6
Invalid partitions for input:
TCOND2: NbDisc = real number with a fractional part
TCOND3: NbDisc = alphabetic
TCOND4: NbDisc= special character
TCOND5: NbDisc = positive Integers and NbDisc> 7

Valid partitions for output:
TCOND6: the disc is displayed on the column if there is space.
TCOND7: Disc isn't displayed on the column and it's still the turn of the same player
Invalid partitions for output (could possibly occur):
TCOND8: Disc not displayed whereas we have place on the column
TCOND9: Disc displayed over the column

Test ID	**Description**	**Expected Results**	**Actual Results**
1	Try to put a disc on the right column. In column 3	If column number 3	SUCCESS
2	The first player place his disc then the second player	If the disc is placed, it announces the end of the turn	SUCCESS
3	Testing place on the column (if full)	If column full and player try putting disc NO DISC IS PUT	SUCCESS
4	Test if game board full (draw)	If the number of tokens is 49 the game ends	SUCCESS
5	Test end game	If 4 discs are aligned, show the winner screen.	SUCCESS

Design

Instructions: Week 6

Journal

Remember: You still will not be writing code at this point in the process.

The following prompts are meant to aid your thought process as you complete the design portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

- List the nouns from your requirements/analysis documentation.
    - Game, Player, Board, Token, Grid, Color, column

- Which nouns potentially may represent a class in your design?
    - Game → Game
    - Player → Player with underclass HumanPlayer and BotPlayer
    - Board → GameBoard

- Which nouns potentially may represent attributes/fields in your design? Also list the class each attribute/field would be a part of.
    - In Game class : gameboard, players
    - In Player class : token_color, name
    - In GameBoard class : grid, columns and rows

- Now that you have a list of possible classes, consider different design options (***lists of classes and attributes***) along with the pros and cons of each. We often do not come up with the best design on our first attempt. Also consider whether any needed classes are missing. These two design options should not be GUI vs. non-GUI; instead you need to include the classes and attributes for each design. Reminder: Each design must include at least two classes that define object types.
-
    - Option1:
      Classes: Game, Player, Board
      Pros: - Straightforward representation of entities involved in the game;
      - Each class represents a distinct entity, making it easy to understand and manage.
      Cons: - Might lead to tight coupling between classes if not designed properly.
      - Changes in one class might necessitate changes in multiple other classes.

    - Option 2 :
      - Model package with :
        · Player (with name and color attributes)
        · BotPlayer (implement player)
        · HumanPlayer (implement player)
        · Board (with grid attribute containing Piece objects)
        · Game
      - Vue package with :
        · Connect4 screen selection mode
        · Vue screen game board
      - Controller to make link between class in Model and Vue packages

Pros: - Utilizes inheritance, reducing redundancy in code.
- Allows for easier extension and modification of classes.
- More modular approach, potentially easier to maintain.

Cons: - Inheritance hierarchy could become complex if additional features are added.
- Overuse of inheritance might lead to less flexibility in design modifications.

- Which design do you plan to use? Explain why you have chosen this design.
  We plan to use Design Option 2 because it provides a more modular and extensible approach to representing the entities involved in the game. The use of inheritance allows for better organization of attributes and methods, making it easier to manage and extend the codebase as needed.

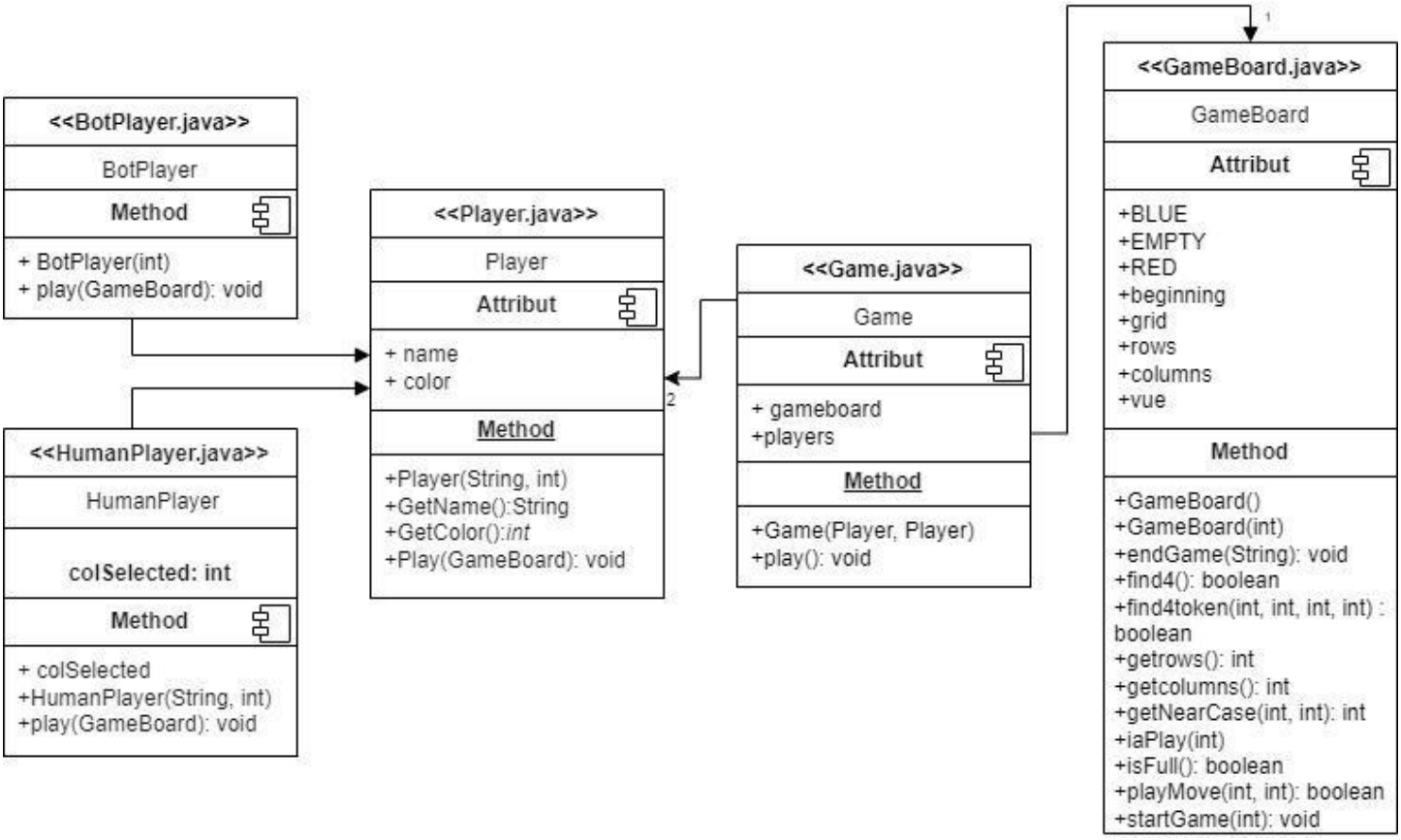
- List the verbs from your requirements/analysis documentation.
  - Play, Drop, Check, Win, Start, Take, Detect, End, Display, Draw, Move

- Which verbs potentially may represent a method in your design? Also list the class each method would be part of.
  - Game : play(),
  - Player : play()
  - HumanPlayer : play()
  - BotPlayer : play()
  - GameBoard : check(),
  - starGame(),
  - placeToken(),
  - playmove(),
  - find4Token(),
  - endGame(),
  - drawCase()

- Other notes:
  - Consider implementing a check method in the Game class to determine if a player has won when aligning 4 tokens.
  - The BotPlayer and HumanPlayer will implement the class play() in player to have their own move.

Software Design

<<Use your notes from above to complete this section of the formal documentation by planning the classes, methods, and fields that will used in the software. Your design should include UML class diagrams along with method headers. ***Prior to starting the formal documentation, you should show your answers to the above prompts to your instructor.***>>



Implementation

Instructions: Week 8

Journal

The following prompts are meant to aid your thought process as you complete the implementation portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

- What programming concepts from the course will you need to implement your design? Briefly explain how each will be used during implementation.
    - For the implementation we use the MVC (Model View Controller) pattern. We started with the model which are the class representing the game then we made the view which represents the interface and we linked the two with the controller

Implementation Details

<<Use your notes from above to write code and complete this section of the formal documentation with a README for the user that explains how he/she will interact with the system.>>

Game Interface:

Upon running the game, you will see a graphical interface representing the Connect Four game board. Here's how you can interact with it:

1. Choosing Game Mode:
    - You could choose to play against a bot (the computer play the other player) or against a true player which is with you in the real world.
    - When your game mode is chosen you could write your name and if in game mode player vs player your name and the name of the other player.
2. Game Board:
    - The game board consists of a grid of cells. Each cell can be empty, filled with a yellow token (player 1), or filled with a red token (player 2).
    - Players take turns dropping tokens into columns. Tokens fall straight down until they land on the lowest unoccupied cell in the column.
3. Player Interaction:
    - Players interact with the game by clicking on the buttons above the game board. Each button represents a column where the player can drop their token.
    - To make a move, simply click on the button corresponding to the column where you want to drop your token.
4. Game Progression:
    - The game progresses as players take turns making moves. The objective is to be the first to form a horizontal, vertical, or diagonal line of four tokens of your color.
    - Once a player achieves this, the game will end, and a message will be displayed indicating the winner. If the game board is filled without a winner, it will result in a draw.
5. Ending the Game:
    - The game can end in one of three ways:
        - One player achieves a winning combination.
        - The game board is filled without a winner (draw).
        - The user manually closes the game window.
6. Restarting the Game:
    - If you wish to play another round, you can simply restart the game by running the program again.

Testing

Instructions: Week 10

Journal

The following prompts are meant to aid your thought process as you complete the testing portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

- Have you changed any requirements since you completed the black box test plan? If so, list changes below and update your black-box test plan appropriately.

No, we have not changed any requirements since the black box test plan was completed. The requirements remain focused on validating the core functionalities of the Connect Four game, as listed in the five test cases:

1. Placing a disc in a specified column.
2. Alternating turns between two players.
3. Preventing disc placement in a full column.
4. Detecting when the game board is full (draw condition).
5. Identifying a winning condition when four discs are aligned.

- List the classes of your implementation. For each class, list equivalence classes, boundary values, and paths through code that you should test.

    - **GameBoard**
        - **Equivalence Classes:**
            - Columns: 1-7 (valid columns for disc placement)
            - Rows: 1-6 (valid rows for disc placement)
            - Disc colors: "Red", "Yellow" (valid disc colors)
        - **Boundary Values:**
            - Columns: 1<=NbDisk<=7
            - Rows: 1<=NbDisk<=6
            - Disc colors: null, empty string, invalid colors
        - **Paths through Code to Test:**
            - Adding discs to valid columns
            - Attempting to add discs to invalid columns/rows
            - Checking if a column is full
            - Checking if the board is full
            - Checking for a win condition
        - **Needed Tests:**
            - Place a disc in a valid column
            - Place a disc when it's another player's turn
            - Place a disc in a full column
            - Check if the board is full
            - Check if the game identifies a winning condition

    - **Player**

        - **Equivalence Classes:**
            - Player names: non-empty strings
            - Disc colors: valid colors ("Red", "Yellow")
        - **Boundary Values:**
            - Player names: empty string, null
            - Disc colors: null, empty string, invalid colors
        - **Paths through Code to Test:**
            - Creating players with valid/invalid names
            - Assigning valid/invalid disc colors to players
        - **Needed Tests:**
            - Create a player with a valid name and disc color
            - Create a player with an empty name (invalid case)
            - Assign a valid disc color to a player
            - Attempt to assign an invalid disc color to a player (invalid case)

- **HumanPlayer (extends Player)**
    - **Equivalence Classes:**
        - Same as Player
    - **Boundary Values:**
        - Same as Player
    - **Paths through Code to Test:**
        - Creating a human player
        - Assigning a disc color to a human player
    - **Needed Tests:**
        - Create a human player with valid name and disc color
        - Assign a disc color to a human player

- **BotPlayer (extends Player)**
    - **Equivalence Classes:**
        - Same as Player
    - **Boundary Values:**
        - Same as Player
    - **Paths through Code to Test:**
        - Creating a bot player
        - Assigning a disc color to a bot player
        - Bot making a move (additional path)
    - **Needed Tests:**
        - Create a bot player with valid name and disc color
        - Assign a disc color to a bot player
        - Test bot player making a move

- **Game**
    - **Equivalence Classes:**
        - Turn sequences: valid and invalid moves

        - Player actions: valid and invalid moves
    - **Boundary Values:**
        - Valid moves: placing a disc in an empty column
        - Invalid moves: placing a disc in a full column
        - Game states: initial, in-progress, draw, win
    - **Paths through Code to Test:**
        - Initializing the game
        - Taking turns in sequence
        - Handling invalid moves
        - Declaring winners
        - Handling a draw condition
    - **Needed Tests:**
        - Initialize the game correctly
        - Simulate a series of valid turns
        - Attempt an invalid move
        - Declare the correct winner when conditions are met
        - Handle a draw condition when the board is full

- **Vue**
    - **Equivalence Classes:**
        - Display states: start, player turn, end
    - **Boundary Values:**
        - Empty board
        - Full board
        - Winning state
    - **Paths through Code to Test:**
        - Displaying the game board at various states
        - Showing messages for start, player turn, and end states
    - **Needed Tests:**
        - Display the game board at the start
        - Update the display after a player's turn
        - Show the winner screen when a player wins
        - Display a draw message when the board is full
- **Connect4**
    - **Equivalence Classes:**
        - Command-line arguments: valid and invalid
        - Game start/stop: valid initialization and termination
    - **Boundary Values:**
        - No arguments
        - Invalid arguments
    - **Paths through Code to Test:**
        - Starting the game with/without valid arguments
        - Handling invalid command-line arguments
        - Properly terminating the game
    - **Needed Tests:**
        - Start the game with valid arguments
        - Handle no arguments scenario (default behavior)

- Handle invalid arguments gracefully
- Terminate the game properly

- Other notes:
    - <<Insert notes>>

Testing Details

<<Use your notes from above to write your test programs and complete this section of the formal documentation by creating a list of your test programs along with descriptions of what they are testing. You will also complete the black-box test plan by running the program and filling in the Actual Results column.>>

Test Programs and Descriptions

1. **Test ID 1: testPlayMove**
    - **Description:** Tests placing a disc in a specific column (column 3) to verify if the disc is correctly placed in the bottom-most position of that column.
    - **Expected Result:** The disc is placed in column 3 at row 0.
2. **Test ID 2: testTurnAnnouncement**
    - **Description:** Tests the alternation of turns between two players. Ensures that after the first player places a disc, the game announces the turn for the second player.
    - **Expected Result:** After placing a disc, the next player's turn is correctly announced.
3. **Test ID 3: testPlaceOnFullColumn**
    - **Description:** Tests attempting to place a disc in a full column. Ensures that the game prevents adding a disc to an already full column.
    - **Expected Result:** No disc is placed when attempting to add a disc to a full column.
4. **Test ID 4: testIsFull**
    - **Description:** Tests if the game correctly identifies when the board is full by filling the entire board with discs.
    - **Expected Result:** The game correctly identifies that the board is full and no more moves can be made.
5. **Test ID 5: testEndGame**
    - **Description:** Tests if the game correctly identifies a winning condition by placing four discs in a row for one player.
    - **Expected Result:** The game identifies the winning condition and declares the winner.

Presentation

Instructions:Week 12

Preparation

The following prompts are meant to aid your thought process as you complete the presentation portion of this exercise. It is recommended that you examine the previous sections of the journal and your reflections as you work on the presentation as it is likely that you have already answered some of the following prompts elsewhere. Please respond to each of the prompts below and feel free to add additional notes.

- Give a brief description of your final project
    - Our final project is the game Connect 4 in which we use Java as a programming language. This game allows two players to compete against each other or against an AI on a virtual board. Players take turns colored discs into a vertical grid, and the first player to align four of their discs horizontally, vertically, or diagonally wins the game.

- Describe your requirement assumptions/additions.
  Requirement assumptions :
    - 1. The game must allow two players to take turns.
    - 2. Players must be able to place their token in a non-full column of the grid.
    - 3. The game must detect and signal when a player aligns four tokens.
    - 4. The game must end when the grid is full or when a player aligns four tokens.
    - 5. The game must clearly display the current state of the grid after each move.
    - 6. Players must be able to start a new game after the end of the previous one.
    - 7. The game must include a user-friendly interface for players to select columns to place their tokens.
    - 8. The game must handle draw cases when the grid is full without any player aligning four tokens.
    - 9. The program must be developed in an appropriate programming language with suitable data structures to represent the grid and tokens.

  Requirement additions :
    1.The game should support two modes : player vs player and player vs ai
    2.The board is a standard 7x6 grid
    3.The ai should have different difficulty levels

- Describe your design options and decision. How did you weigh the pros and cons of the different designs to make your decision?
  Design Option 1:

    - Classes: Game, Player, Board, Piece, BotPlayer
    - Pros:
        - Clear separation of responsibilities.
        - Easier to maintain and debug.
    - Cons:
        - Potential for tight coupling between classes.

  Design Option 2:

- Model package with :
  - Player (with name and color attributes)
  - BotPlayer (implement player)
  - HumanPlayer (implement player)
  - Board (with grid attribute containing Piece objects)
  - Game
- Vue package with :
  - Connect4 screen selection mode
  - Vue screen game board
- Controller to make link between class in Model and Vue packages

Pros: - Utilizes inheritance, reducing redundancy in code.
- Allows for easier extension and modification of classes.
- More modular approach, potentially easier to maintain.

Cons: - Inheritance hierarchy could become complex if additional features are added.
- Overuse of inheritance might lead to less flexibility in design modifications.

Decision:
We choose Design Option 2 because it provides a more modular and extensible approach to representing the entities involved in the game. The use of inheritance allows for better organization of attributes and methods, reduces code redundancy, making it easier to manage and extend the codebase as needed and functionalities like adding new bot difficulty levels.

- How did the extension affect your design?
  - The extension required us to introduce additional features like different bot difficulty levels. This necessitated more detailed class structures and interactions, especially around the BotPlayer class. We also had to ensure our design could handle game state persistence efficiently.

- Describe your tests (e.g., what you tested, equivalence classes).
  - Put a disc in the grid (class Game, GameBoard)
  - The first player begins the game then the second player (players' turn) (class Player, BotPlayer, HumanPlayer)
  - Test which player wins the game (class Game, Player, BotPlayer, HumanPlayer)
  - Test if the game board is full (it's a draw) (class Game, GameBoard)
  - Test if possible to put a disc on a full column (class Game, GameBoard)

- What functionalities are you going to demo?
  - A game against the bot player (randomly put a disc on the board) and the winner with the method who verified it

- Who is going to speak about each portion of your presentation? (Recall: Each group will have ten minutes to present their work; minimum length of group presentation is seven minutes. Each student must present for at least two minutes of the presentation.)
  - Elias : intro and requirement
  - Kevin : requirement and choice of design
  - Wilfried : design and demo
  - Guy-Leonard : test and conclusion

- Other notes:

- <<Insert notes>>

<<Use your notes from above to complete create your slides and plan your presentation and demo.>>