Comprehensive Exercise Report

Team Softwarangers of Section <<000>>

Timur Daminov 211ADB053

Nweze Isabella Chinasa 240ADB034

Shane Micah Jayaraj 191ADB012

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

[**Requirements/Analysis**](#_uwgqwd5ezv2w) **2**

[Journal](#_lsityg2iq9m6) 2

[Software Requirements](#_2h0vru1u2mla) 3

[**Black-Box Testing**](#_prhaxdxmf8n8) **4**

[Journal](#_18f11w613jft) 4

[Black-box Test Cases](#_2xn4jzot820y) 5

[**Design**](#_24fdizefyocn) **6**

[Journal](#_esp2ocs9j6bk) 6

[Software Design](#_aifbl1x6rddt) 7

[**Implementation**](#_hya8f3jqkba6) **8**

[Journal](#_acupzfhai7gz) 8

[Implementation Details](#_ojhtwkms2z3b) 9

[**Testing**](#_3qvya3vi836q) **10**

[Journal](#_ckfs4xbl5pyr) 10

[Testing Details](#_bzt1547yxzxi) 11

[**Presentation**](#_hdjvrbf45b1p) **12**

[Preparation](#_xbiquwtmf36n) 12

[**Grading Rubric**](#_u0hfnmdgusmf) **13**

# 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 classic strategy game where two players strategically drop colored discs into a grid, aiming to be the first to connect four discs of their color either horizontally, vertically, or diagonally.

**The functional requirements:**

* + - * The game must provide a grid where players can drop discs.
      * Players can choose their color
      * Discs must stack upwards, horizontally, or diagonally upon being dropped into the grid.
      * The game must be able to detect win
      * The game should prevent players from dropping discs into a column that is already full.
      * It should display the current state of the game, including the grid and the positions of the discs.

**Non-functional requirements:**

* + - * The game interface should be intuitive and visually appealing, suitable for players of all ages.
* The connect 4 users may include children, parents, friends and families
* Describe how each user would interact with the software
  + Engage in single-player mode against the computer or play with friends or family in multiplayer mode.
* What features must the software have? What should the users be able to do?
  + Multiplayer mode
  + Selecting player who makes move first

## Software Requirements

The Connect 4 software project aims to deliver a digital adaptation of the classic strategy game, providing an engaging and intuitive experience for players of all ages. Players will strategically drop colored discs into a grid, aiming to connect four discs of their color either horizontally, vertically, or diagonally before their opponent. The software will feature both single-player mode against a computer opponent and multiplayer mode for playing with friends or family. It will offer customization options such as selecting board size and choosing the starting player, ensuring a personalized gaming experience. The interface will be designed to be visually appealing and user-friendly, meeting the needs of diverse user groups including children, parents, and casual gamers.

**Requirements:**

**Functional Requirements:**

* The software must provide a grid where players can drop discs.
* Players must be able to choose their color before starting the game.
* Discs must stack upwards, horizontally, or diagonally upon being dropped into the grid.
* The game must be able to detect a win condition when a player connects four discs of their color.
* The software should prevent players from dropping discs into a column that is already full.
* It should display the current state of the game, including the grid and the positions of the discs.

**Non-functional Requirements:**

* The game interface should be intuitive and visually appealing, suitable for players of all ages.

**User Stories:**

* As a player, I want to be able to choose starting player.
* As a parent, I want the game interface to be visually appealing and easy to navigate so that my children can enjoy playing independently.
* As a casual gamer, I want the option to play against the computer or against friends and family in multiplayer mode for added variety and challenge.

# Implementation:

# 

# 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)?
  + selected column
* What does output for the software look like (e.g., what type of data, how many pieces of data)?
  + updated game board after each player's move.
* What equivalence classes can the input be broken into?
  + input column can be broken into valid and invalid classes
* What boundary values exist for the input?
  + Lower boundary: Column 1 (minimum valid input).
  + Upper boundary: Column 7 (maximum valid input).
  + Values beyond these boundaries would be invalid inputs.
* Are there other cases that must be tested to test all requirements?
  + Testing for a win condition
  + Testing different game states
* Other notes:
  + It's important to make sure that the game behaves correctly under various scenarios, such as different sequences of moves leading to different outcomes.

## 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.

|  |  |  |  |
| --- | --- | --- | --- |
| **Test ID** | **Description** | **Expected Results** | **Actual Results** |
| 1 | Verify that the game initializes correctly and displays the start screen. | The start screen should appear with options to choose the starting player (Red or Yellow). | The start screen starts with opportunity to choose starting color, after that the game starts |
| 2 | Check the winning condition horizontally, vertically and diagonally. | The game should announce winner after 4 circles the same color connected horizontally vertically or diagonally | The winning condition function works as expected |
| 3 | Check for opportunity to restart the game or quit if player wishes | Game should ask player whether he wants to restart game or quit | Both restart function and quit function implemented as buttons in popup window and works as expected |

# 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.
  + Grid
  + Players
  + Discs
  + Color
  + Win condition
  + Column
  + State
  + Interface
* Which nouns potentially may represent a class in your design?
  + Grid
  + Players
  + Discs
* Which nouns potentially may represent attributes/fields in your design? Also list the class each attribute/field would be a part of.
  + Color (belonging to Disc class)
  + Win condition (belonging to Game class)
  + State (belonging to Game class)
  + Column (belonging to Grid class)
  + Interface (belonging to Game class)
* 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.
  + Design Option 1:
  + Grid Class:
    - Attributes: Rows, Columns
  + Player Class:
    - Attributes: Color
  + Disc Class:
    - Attributes: Position, Color
  + Game Class:
    - Attributes: State, Win condition, Current player
  + Methods: dropDisc(), detectWin(), switchPlayer()
  + Design Option 2:
  + GameBoard Class:
    - Attributes: Grid (2D array), Players
  + Player Class:
    - Attributes: Color
  + Game Class:
    - Attributes: State, Win condition, Current player
* Which design do you plan to use? Explain why you have chosen this design.
  + We plan to use Design Option 1 because it provides a more modular and flexible approach with separate classes for managing the grid, players, discs, and game logic. This design allows for better encapsulation and separation of concerns, making the codebase easier to maintain and extend in the future.
* List the verbs from your requirements/analysis documentation.
  + Provide
  + Choose
  + Stack
  + Detect
  + Prevent
  + Display
* Which verbs potentially may represent a method in your design? Also list the class each method would be part of.
  + DropDisc()
  + DetectWin()
  + SwitchPlayer()

## Software Design

**1. Game Class:**

* **Fields:**
  + state: GameState (Enumeration)
  + currentPlayer: Player
* **Methods:**
  + **dropCircle(column: int): boolean**: Drops a disc into the specified column if valid, returns true if successful.
  + **winningMove(): boolean**: Checks if the current player has won the game.
  + **switchPlayer(): void**: Switches the turn to the next player.

**2. Player Class:**

* **Fields:**
  + color: Color (Enumeration)

**3. Disc Class:**

* **Fields:**
  + position: Position (Tuple representing row and column)
  + color: Color (Enumeration)

**4. Grid Class:**

* **Fields:**
  + cells: Cell[7][6] (2D array representing the game board)

**Additional Notes:**

* **Enums:**
  + **GameState**: Enumerates the possible states of the game (e.g., Ongoing, Draw, Won).
  + **Color**: Enumerates the possible colors of discs (e.g., Red, Yellow).
* **Classes:**
  + The **Game** class manages the game logic, including turn-taking, win detection, and disc dropping.
  + The **Player** class represents each player in the game, storing their chosen color.
  + The **Disc** class represents individual discs on the game board, storing their position and color.
  + The **Grid** class manages the game board, including initializing cells and checking for valid moves.

# Implementation

Instructions: Week 8

## Journal

**Programming concepts used:**

**1. Object-Oriented Programming (OOP)**

Classes and Objects: The Connect4 class encapsulates the entire game logic and UI components. An instance of this class (game) is created to start the application.

Methods: Functions within the Connect4 class (e.g., start\_screen, start\_game, reset\_game, draw\_board) encapsulate behaviors and game logic.

Encapsulation: The game state and logic are encapsulated within the Connect4 class, preventing direct access from outside the class.

**2. Graphical User Interface (GUI) Programming**

Tkinter Library: The project uses the Tkinter library to create and manage the GUI. Components like Tk, Canvas, Button, and Label are used for user interaction.

Event Handling: Mouse events (<Motion>, <Button-1>) are used to interact with the game board, updating the display based on user actions.

**3. Numpy Library**

Array Manipulation: The numpy library is used to create and manipulate the game board, which is a 2D array representing the state of the game.

## 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.>>

# 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.
  + <<Insert answer>>
* List the classes of your implementation. For each class, list equivalence classes, boundary values, and paths through code that you should test.
  + Initialization and Start Screen
    - Equivalence Class: Correct initialization and display of start screen
    - Test: Verify that the start screen is displayed correctly with "Red" and "Yellow" buttons.
  + Starting the Game
    - Equivalence Class: Game starts with either Red or Yellow player
    - Test: Verify that the game board is displayed and the correct player's piece follows the cursor.
  + Dropping a Piece
    - Equivalence Class: Valid column selection
    - Test: Verify that a piece is dropped in the lowest available row in the selected column.
    - Equivalence Class: Invalid column selection (full column)
    - Test: Verify that a piece cannot be dropped in a full column.
  + Game Play
    - Equivalence Class: Alternating turns
    - Test: Verify that turns alternate correctly between Red and Yellow players.
    - Equivalence Class: Detecting win conditions
    - Test: Verify detection of horizontal, vertical, and diagonal win conditions for both players.
    - Test: Verify that the game detects a draw when the board is full.
  + Resetting the Game
    - Equivalence Class: Reset game functionality
    - Test: Verify that the game board resets and the start screen reappears when the reset button is pressed.
  + Boundary Values
    - Dropping a Piece
    - Boundary Value: First column (0)
    - Boundary Value: Last column (COLUMN\_COUNT - 1)
    - Test: Verify pieces can be dropped in the first and last columns.
    - Boundary Value: Top row (ROW\_COUNT - 1)
    - Boundary Value: Bottom row (0)
    - Test: Verify pieces are placed correctly in the top and bottom rows.
    - Win Conditions
    - Boundary Value: Winning move with the minimum number of pieces (four in a row)
    - Test: Verify win is detected with exactly four pieces in a row horizontally, vertically, and diagonally.
* 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.>>

# 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
* Describe your requirement assumptions/additions.
* Describe your design options and decision. How did you weigh the pros and cons of the different designs to make your decision?
* Describe your tests (e.g., what you tested, equivalence classes).
* What lessons did you learn from the comprehensive exercise (i.e., programming concepts, software process)?
* What functionalities are you going to demo?
* 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.)
* Other notes:

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