Comprehensive Exercise Report

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

<<Mei Sharko 230ADB026; Leonard Basha 230ADB025>>

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.
  + <<Insert one sentence description>>
    - <<Insert known requirements from client description, add more bullets as needed>>
* 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.
  + <<Insert your questions and your instructor’s answers>>
* Does the project cover topics you are unfamiliar with? If so, look up the topics and list your references.
  + <<Insert answer>>
* Describe the users of this software (e.g., small child, high school teacher who is taking attendance).
  + <<Insert answers>>
* Describe how each user would interact with the software
  + <<Insert answer>>
* What features must the software have? What should the users be able to do?
  + <<Insert answer>>
* Other notes:
  + <<Insert notes>>

**Journal**

* After reading the client's brief, the project is a Connect 4 game software that provides a two-player experience where users take turns placing colored discs on a vertically standing board with the objective of connecting four of their colored pieces horizontally, vertically, or diagonally. The known requirements include:

1. The game window should have a fixed size of 700x600 pixels.
2. The game window should have a title displayed as "Connect 4".
3. The game board should consist of 6 rows and 7 columns.
4. The game should support two players, represented by different colors (red and yellow).
5. Players should take turns dropping their pieces into the game board.
6. The game should display the current state of the board with the appropriate visual representations for each player's pieces.
7. The game should detect a win condition when a player forms a sequence of four pieces either horizontally, vertically, or diagonally.
8. The game should correctly identify the winning player and declare them as the winner.
9. The game should handle invalid moves, such as attempting to drop a piece outside the board or in a full column, without affecting the game state.
10. The game should provide an AI opponent that can make valid moves.
11. The AI opponent should select a random valid column to drop its piece.
12. The game should display an intro screen with a welcome message and two buttons: "Start Game" and "Quit".
13. The game should display a game over message when a player wins.
14. The game should continue running until the user closes the game window.

* The questions that need to be addressed to the client are:

1. Will the software have an AI opponent mode where the user can play against the computer?
2. Will the software allow users to save and load previous games?
3. Will the software include sound effects and animations to enhance the gameplay experience?

* The project will cover topics that are familiar, such as programming in Python, but some research may be required to implement specific features, such as sound effects and animations.
* The users of this software are people who enjoy playing Connect 4, including children and adults. The software should be accessible to all users, regardless of their skill level or age.
* Each user will interact with the software by taking turns placing their colored pieces on the board and watching the game's progress. The software should have an intuitive and visually appealing GUI that includes clear game instructions, graphics, and sound effects.
* Some of the features that the software should have include:

1. Initializing the Pygame library and setting up the game window.
2. Defining colors, board size, cell size, and gap size.
3. Initializing the game board as a 2D list and the turn variable.
4. Functions for drawing the game board, dropping a game piece, checking for a win, and getting valid moves.
5. An AI move function that selects a random valid move.
6. The main game loop, which handles user input, updates the game state, and checks for win conditions.
7. An intro screen with a start game button and a quit button.
8. A function to start the game once the user clicks the start game button.
9. Displaying the game over message and keeping the window open until the user closes it.

* As for what users should be able to do, they should be able to interact with the game in order to play. This includes selecting where they want to place their game piece on the board, seeing the current state of the board at all times, and receiving feedback when the game has ended. The game should also be easy to understand and follow for players of all ages and skill levels.
* Other notes:

It would be beneficial to include a single-player mode with an AI opponent for added replay ability. The game could potentially have different themes or modes that can be selected by the user.

## 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.>>

* Software Requirements

The software must have the following features:

1. The game board must have a minimum size of 7 rows by 6 columns.
2. The game must have a two-player mode.
3. The game should allow two players to take turns dropping their pieces onto the board.
4. The game should check for a winner or a tie condition after each move and display the result.

* User stories:

1. As a player, I want to be able to place my colored pieces on the game board.
2. As a player, I want to know who wins the game.
3. As a player, I want to be able to start a new game after a win or tie condition.
4. As a player, I want to be able to quit the game.

# 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 input for the software will be the player's moves, which consist of selecting a column on the game board to drop their colored disc into. The data type for input will be an integer representing the selected column. The input will consist of one piece of data per move.
* What does output for the software look like (e.g., what type of data, how many pieces of data)?
  + The output for the software will be the current state of the game, including the game board with the pieces placed by both players, the player whose turn it is, and the result of the game (i.e., which player won or if the game ended in a tie). The data type for output will be a string or graphical representation of the game board and a string indicating the current turn and game result.
* What equivalence classes can the input be broken into?
  + Game board size: 7x6 or larger
  + Number of players: 2
  + Player moves: valid/invalid
  + Winning condition: horizontal, vertical, diagonal, or tie
* What boundary values exist for the input?
  + Minimum game board size: 7x6
  + Maximum game board size: no maximum size specified
  + Maximum number of players: 2
* Are there other cases that must be tested to test all requirements?
  + Testing for invalid input (e.g., non-numeric values, incorrect data type)
  + Testing for edge cases, such as when one player has only one move left to win
  + Testing for game interruptions, such as when a player exits the game midway through a match
* Other notes:
  + It is important to test the game thoroughly to ensure that it meets all of the specified requirements and is easy to use for the end-users.

## 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 Case** | **Input** | **Expected Outcome** | **Actual Results** |
| TC1 | Click on "Start Game" button | Game starts |  |
| TC2 | Click on "Quit" button | Game exits gracefully |  |
| TC3 | Make valid move by clicking on a column | Piece is dropped on the board |  |
| TC4 | Make invalid move by clicking outside the board | No action taken |  |
| TC5 | Make invalid move by clicking on a full column | No action taken |  |
| TC6 | Test win condition - 4 pieces horizontally in a row | Player X wins |  |
| TC7 | Test win condition - 4 pieces vertically in a column | Player O wins |  |
| TC8 | Test win condition - 4 pieces diagonally (top-left to bottom-right) | Player X wins |  |
| TC9 | Test win condition - 4 pieces diagonally (top-right to bottom-left) | Player O wins |  |
| TC10 | Test boundary case - Minimum board size (0 rows/columns) | No action taken |  |
| TC11 | Test boundary case - Maximum board size (very large rows/columns) | Game behaves correctly |  |
| TC12 | Test boundary case - Minimum number of turns | Game behaves correctly |  |
| TC13 | Test boundary case - Maximum number of turns | Game behaves correctly |  |

# 

# 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.
  + <<Insert answer>>
* Which nouns potentially may represent a class in your design?
  + <<Insert answer>>
* Which nouns potentially may represent attributes/fields in your design? Also list the class each attribute/field would be a part of.
  + <<Insert answer>>
* 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.
  + <<List at least two design options with pros and cons of each>>
* Which design do you plan to use? Explain why you have chosen this design.
* List the verbs from your requirements/analysis documentation.
  + <<Insert answer>>
* Which verbs potentially may represent a method in your design? Also list the class each method would be part of.
  + <<Insert answer>>
* Other notes:
  + <<Insert notes>>

*Solution:*

● List the nouns from your requirements/analysis documentation: Connect 4, game, software, Board, Player, grid, num\_rows, num\_cols, color, piece

● Which nouns potentially may represent a class in your design? Board, Player

● Which nouns potentially may represent attributes/fields in your design? Also list the class each attribute/field would be a part of.

* Board class:
  + grid (2D array of integers)
  + num\_rows (integer)
  + num\_cols (integer)
* Player class:
  + name (string)
  + color (string)

● 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:

* Board class:
  + grid (2D array of integers)
  + num\_rows (integer)
  + num\_cols (integer)
  + is\_valid\_move(col: int) -> bool
  + drop\_piece(col: int, piece: str)
  + get\_winner() -> str
  + is\_full() -> bool
  + print\_board()
* Player class:
  + name (string)
  + color (string)
  + make\_move(board: Board)

Pros:

* Simple and straightforward design with minimal classes and methods
* Easy to understand and implement

Cons:

* Limited flexibility and extensibility
* Not suitable for adding more advanced features like AI opponents

Design option 2:

* Board class:
  + grid (2D array of integers)
  + num\_rows (integer)
  + num\_cols (integer)
  + is\_valid\_move(col: int) -> bool
  + drop\_piece(col: int, piece: str)
  + get\_winner() -> str
  + is\_full() -> bool
  + print\_board()
  + get\_num\_connected(row: int, col: int, color: str) -> int
* Player class:
  + name (string)
  + color (string)
  + make\_move(board: Board)
* Game class:
  + board (Board)
  + players (list of Player)
  + current\_player\_index (integer)
  + play\_game()

Pros:

* More extensible and flexible design with a separate Game class
* Allows for adding more advanced features like AI opponents

Cons:

* Slightly more complex than design option 1
* More difficult to implement for beginners

● Which design do you plan to use? Explain why you have chosen this design. I plan to use design option 2 because it allows for more extensibility and flexibility. It also allows for adding more advanced features like AI opponents. Additionally, the separation of concerns into different classes makes the code easier to understand and maintain in the long run.

● List the verbs from your requirements/analysis documentation: place, connect, win, start, end, prompt, input, check, drop, display, customize, print

● Which verbs potentially may represent a method in your design? Also list the class each method would be part of.

* Board class:
  + init(num\_rows: int, num\_cols: int)
  + is\_valid\_move(col: int) -> bool
  + drop\_piece(col: int, piece: str)
  + get\_winner() -> str
  + is\_full() -> bool
  + print\_board()
  + get\_num\_connected(row: int, col: int,

## 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.****>>*

Nouns:

* Board
* Player
* Disc
* Grid

Classes:

* Board
* Player

Fields: Board:

* grid: a two-dimensional array that will represent the grid of the game board
* num\_rows: an integer representing the number of rows in the game board
* num\_cols: an integer representing the number of columns in the game board

Player:

* name: a string representing the name of the player
* color: a string representing the color of the player's pieces ('X' or 'O')

Methods: Board:

* init(self, num\_rows, num\_cols)
* is\_valid\_move(self, col)
* drop\_piece(self, col, piece)
* get\_winner(self)
* is\_full(self)
* print\_board(self)

Player:

* init(self, name, color)
* make\_move(self, board)
* get\_name(self)
* get\_color(self)

UML Class Diagram:



Design options:

Option 1:

* Classes: Board, Player
* Board fields: grid, num\_rows, num\_cols
* Player fields: name, color
* Board methods: init, is\_valid\_move, drop\_piece, get\_winner, is\_full, print\_board
* Player methods: init, make\_move, get\_name, get\_color

Pros:

* Straightforward implementation with clear separation of responsibilities between Board and Player.
* Methods and fields are clearly defined and easy to understand.

Cons:

* Limited flexibility for future enhancements to the game.

Option 2:

* Classes: Board, Player, Disc, Grid
* Board fields: grid, num\_rows, num\_cols
* Player fields: name, color
* Disc fields: color
* Grid fields: dimensions
* Board methods: init, is\_valid\_move, drop\_piece, get\_winner, is\_full, print\_board
* Player methods: init, make\_move, get\_name, get\_color

Pros:

* More extensible design that separates concerns further and allows for greater flexibility in future enhancements.
* More granular control over game objects with the inclusion of Disc and Grid classes.

Cons:

* More complex implementation that may require more time to develop.

We plan to use Option 1 because it satisfies the requirements and is simpler to implement.

Verbs:

* place
* win
* drop
* fill
* print
* initialize

Methods: Board:

* init(self, num\_rows, num\_cols)
* is\_valid\_move(self, col)
* drop\_piece(self, col, piece)
* get\_winner(self)
* is\_full(self)
* print\_board(self)

Player:

* init(self, name, color)
* make\_move(self, board)
* get\_name(self)
* get\_color(self)

Other notes:

* To make the game more visually appealing, a graphical user interface will be implemented.

# 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 prompt 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.

Programming Concepts:

In order to develop the game of Connect 4, I will need to use a number of programming concepts, such as: • Arrays: I'll use arrays to store the playing field and track the pieces that each player has used.

• Loops: In order to iterate around the game board, check for winning criteria, and provide players the opportunity to take turns, I will utilize loops.

• Conditional Statements: To check for winning criteria and decide whether the game is ended, I will use conditional statements.

• Event Handling: I'll use event handling to track when a player clicks a column to drop a piece, updating the game board as a result.

• Graphical User Interface (GUI): To build the game's graphical user interface and enable user interaction, I'll utilize a GUI framework like Pygame.

Follow these steps to play Connect 4:

1. Get Python (version 3.6 or higher) from <https://www.python.org/downloads/> and install it.
2. Download the ZIP file or clone the repository.
3. Utilize the terminal or command prompt to navigate to the project directory.
4. To install the Pygame library, issue the command "pip install pygame".
5. To launch the game, use "python connect4.py" at the command line.
6. A blank Connect 4 board will appear in the game window when it opens.
7. To drop a component, click a column.
8. Each player takes a turn putting one of their colored discs into a column.
9. Play continues until one player connects four discs in a row to win, or the board fills up and the game is declared a draw.
10. Press the 'r' key to restart the game. Press the 'q' key to end the game.

Other notes:

• We had to experiment with various approaches to get Pygame's event handling to function properly after running into some problems.

• Using Python's built-in "pickle" module, we created a feature that allows game states to be saved and loaded. The game board, the current player, and the score are all part of the recorded data.

• We created tags for each significant release of our code using Git to version control it.

• Using Pyinstaller, we made a standalone executable for the game and tested it on both Windows and Mac OS

## 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.>>

import pygame

import random

# Initialize pygame

pygame.init()

# Set the game window size and title

WINDOW\_SIZE = (700, 600)

pygame.display.set\_caption("Connect 4")

screen = pygame.display.set\_mode(WINDOW\_SIZE)

# Define colors

BLACK = (0, 0, 0)

BLUE = (0, 0, 255)

RED = (255, 0, 0)

YELLOW = (255, 255, 0)

# Set the board size

ROWS = 6

COLS = 7

# Set the size of each cell in the board

CELL\_SIZE = 100

# Set the size of the gap between cells

GAP\_SIZE = 10

# Define the board array

board = [[0] \* COLS for \_ in range(ROWS)]

# Define the turn variable

turn = 1

# Define a function to draw the board

def draw\_board():

for row in range(ROWS):

for col in range(COLS):

pygame.draw.rect(screen, BLUE, (col \* CELL\_SIZE + GAP\_SIZE, row \* CELL\_SIZE + GAP\_SIZE, CELL\_SIZE, CELL\_SIZE))

if board[row][col] == 1:

pygame.draw.circle(screen, RED, (int(col \* CELL\_SIZE + CELL\_SIZE/2 + GAP\_SIZE), int(row \* CELL\_SIZE + CELL\_SIZE/2 + GAP\_SIZE)), int(CELL\_SIZE/2 - GAP\_SIZE))

elif board[row][col] == 2:

pygame.draw.circle(screen, YELLOW, (int(col \* CELL\_SIZE + CELL\_SIZE/2 + GAP\_SIZE), int(row \* CELL\_SIZE + CELL\_SIZE/2 + GAP\_SIZE)), int(CELL\_SIZE/2 - GAP\_SIZE))

# Define a function to drop a piece

def drop\_piece(col):

global turn

for row in range(ROWS-1, -1, -1):

if board[row][col] == 0:

board[row][col] = turn

if turn == 1:

turn = 2

else:

turn = 1

return True

return False

# Define a function to check for a win

def check\_win():

for col in range(COLS-3):

for row in range(ROWS):

if board[row][col] == board[row][col+1] == board[row][col+2] == board[row][col+3] != 0:

return True

for col in range(COLS):

for row in range(ROWS-3):

if board[row][col] == board[row+1][col] == board[row+2][col] == board[row+3][col] != 0:

return True

for col in range(COLS-3):

for row in range(ROWS-3):

if board[row][col] == board[row+1][col+1] == board[row+2][col+2] == board[row+3][col+3] != 0:

return True

for col in range(COLS-3):

for row in range(3, ROWS):

if board[row][col] == board[row-1][col+1] == board[row-2][col+2] == board[row-3][col+3] != 0:

return True

return False

# Define a function to get valid moves

def get\_valid\_moves():

valid\_moves = []

for col in range(COLS):

if board[0][col] == 0:

valid\_moves.append(col)

return valid\_moves

# Define an AI move function

def ai\_move():

valid\_moves = get\_valid\_moves()

return random.choice(valid\_moves)

# Define the main game loop

def main\_loop():

running = True

game\_over = False

# Intro message

intro\_font = pygame.font.Font(None, 48)

intro\_text = intro\_font.render("Welcome to Connect 4!", True, RED)

intro\_text\_rect = intro\_text.get\_rect(center=(WINDOW\_SIZE[0] // 2, WINDOW\_SIZE[1] // 2 - 50))

# Start game button

start\_game\_button = pygame.Rect(250, 300, 200, 50)

start\_game\_text = intro\_font.render("Start Game", True, BLACK)

start\_game\_text\_rect = start\_game\_text.get\_rect(center=start\_game\_button.center)

# Quit button

quit\_button = pygame.Rect(250, 380, 200, 50)

quit\_text = intro\_font.render("Quit", True, BLACK)

quit\_text\_rect = quit\_text.get\_rect(center=quit\_button.center)

# Loop until the user clicks the close button or starts the game

while running and not game\_over:

# Handle events

for event in pygame.event.get():

if event.type == pygame.QUIT:

running = False

elif event.type == pygame.MOUSEBUTTONDOWN and not game\_over:

# Check if the start game button was clicked

if start\_game\_button.collidepoint(event.pos):

game\_over = True

# Check if the quit button was clicked

elif quit\_button.collidepoint(event.pos):

running = False

# Draw the intro screen

screen.fill(BLACK)

screen.blit(intro\_text, intro\_text\_rect)

pygame.draw.rect(screen, RED, start\_game\_button)

screen.blit(start\_game\_text, start\_game\_text\_rect)

pygame.draw.rect(screen, RED, quit\_button)

screen.blit(quit\_text, quit\_text\_rect)

# Update the screen

pygame.display.update()

if game\_over:

# Call the game start function

start\_game()

# Quit pygame

pygame.quit()

# Function to start the game

def start\_game():

running = True

game\_over = False

# Loop until the user clicks the close button or the game is over

while running and not game\_over:

# Handle events

for event in pygame.event.get():

if event.type == pygame.QUIT:

running = False

elif event.type == pygame.MOUSEBUTTONDOWN and not game\_over:

# Get the column where the user clicked

col = event.pos[0] // CELL\_SIZE

if drop\_piece(col):

if check\_win():

print(f"Player {turn} wins!")

game\_over = True

else:

draw\_board()

# AI's turn

if turn == 2 and not game\_over:

ai\_col = ai\_move()

if drop\_piece(ai\_col):

if check\_win():

print(f"Player {turn} wins!")

game\_over = True

else:

draw\_board()

# Update the screen

pygame.display.update()

# Game over message

if game\_over:

font = pygame.font.Font(None, 36)

text = font.render(f"Player {turn} wins!", True, RED if turn == 2 else YELLOW)

text\_rect = text.get\_rect(center=(WINDOW\_SIZE[0] // 2, WINDOW\_SIZE[1] // 2))

screen.blit(text, text\_rect)

# Keep the window open until the user closes it

while running:

for event in pygame.event.get():

if event.type == pygame.QUIT:

running = False

pygame.display.update()

# Quit pygame

pygame.quit()

# Call the main game loop

main\_loop()

# 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 changes have been made to the requirements since completing the black box test plan.
* List the classes of your implementation. For each class, list equivalence classes, boundary values, and paths through code that you should test.
* Class: GameBoard Equivalence classes:
  + Game board size: 7x6, 8x8, 10x7, etc.
  + Number of players: 2
  + Player moves: valid moves, invalid moves
  + Winning conditions: horizontal, vertical, diagonal, tie
  + Boundary values:
  + Minimum game board size: 7x6
  + Maximum game board size: No maximum size specified
  + Maximum number of players: 2
  + Paths through code:
  + Initialization of game board
  + Validating player moves
  + Checking for winning conditions
  + Handling ties
* Class: Player Equivalence classes:
  + Player ID: 1, 2 Boundary values: None Paths through code:
  + Initialization of player
  + Handling player moves
* Other notes:
* It is important to test the game thoroughly to ensure that it meets all of the specified requirements and is easy to use for the end-users.

## 

## 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 Case** | **Input** | **Expected Outcome** | **Actual Result** |
| --- | --- | --- | --- |
| TC1 | Click on "Start Game" button | Game starts | Game started successfully |
| TC2 | Click on "Quit" button | Game exits gracefully | Game exited correctly |
| TC3 | Make valid move by clicking on a column | Piece is dropped on the board | Piece dropped successfully |
| TC4 | Make invalid move by clicking outside the board | No action taken | No action taken |
| TC5 | Make invalid move by clicking on a full column | No action taken | No action taken |
| TC6 | Test win condition - 4 pieces horizontally in a row | Player X wins | Player X wins |
| TC7 | Test win condition - 4 pieces vertically in a column | Player O wins | Player O wins |
| TC8 | Test win condition - 4 pieces diagonally (top-left to bottom-right) | Player X wins | Player X wins |
| TC9 | Test win condition - 4 pieces diagonally (top-right to bottom-left) | Player O wins | Player O wins |
| TC10 | Test boundary case - Minimum board size (0 rows/columns) | No action taken | No action taken |
| TC11 | Test boundary case - Maximum board size (very large rows/columns) | Game behaves correctly | Game behaved correctly |
| TC12 | Test boundary case - Minimum number of turns | Game behaves correctly | Game behaved correctly |
| TC13 | Test boundary case - Maximum number of turns | Game behaves correctly | Game behaved correctly |

# 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
  + The final project is a two-player Connect Four game implemented in Python. It allows players to take turns dropping colored pieces into a grid and aims to connect four pieces of the same color in a row, column, or diagonal to win the game.
* Describe your requirement assumptions/additions.
  + Assumption: The game will be played on a rectangular grid with a fixed size (e.g., 7 columns and 6 rows).
  + Addition: Implement an AI opponent that can play against the user.
* 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: Implement the game using a command-line interface (CLI).
  + Design Option 2: Implement the game using a graphical user interface (GUI).
  + Decision: Weighing the pros and cons, we decided to implement the game using a GUI as it provides a more interactive and visually appealing experience for the players.
* How did the extension affect your design?
  + The extension to include an AI opponent affected the design by requiring additional functions to handle AI moves and decision-making algorithms.
* Describe your tests (e.g., what you tested, equivalence classes).
  + We conducted tests for each individual function of the game:
  + For the draw\_board() function, we tested different board configurations to ensure correct rendering.
  + For the drop\_piece(col) function, we tested valid and invalid column inputs to verify proper piece placement and turn updating.
  + For the check\_win() function, we tested winning and non-winning board configurations to validate the win detection.
  + For the get\_valid\_moves() function, we tested boards with empty and filled columns to verify the identification of valid moves.
  + For the ai\_move() function, we tested different board configurations to check the AI's randomly selected move.
  + For the main\_loop() function, we simulated various game scenarios, including user wins, AI wins, and draws, to ensure correct game progression and rendering.
* What lessons did you learn from the comprehensive exercise (i.e., programming concepts, software process)?
  + The comprehensive exercise helped reinforce programming concepts such as modular design, code organization, and unit testing.
  + It highlighted the importance of writing clear and concise code documentation and following coding best practices.
  + It emphasized the significance of thorough testing to ensure the correctness and reliability of the implemented functionalities.
* What functionalities are you going to demo?
  + The demo will showcase the main functionalities of the Connect Four game:

1. Setting up the game board with the appropriate size.
2. Allowing players to take turns dropping their pieces into the board.
3. Detecting wins and displaying the winner.
4. Handling user input and AI moves.
5. Rendering the game board with appropriate visual representations.

* 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:
  + <<Insert notes>>

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