Development & Testing the Connet-4 Game

Stage 1: Developing the Main Gameplay

To develop the main gameplay of connect-4 we use three classes: the board class, the player class, and the game class.

1. Board class: This class will handle the drawing of the board and the placement of the game pieces on the board.
2. Player class: This class will handle the players' moves, such as selecting the column to place their game piece.
3. Game class: This class will handle the game's overall logic, such as checking for a win or a draw and alternating turns between the players.

Develop the Board class:

The "Board" class will represent the game board for Connect-4.

Date:

Planning:

1. Set up the board: Initialize the board with 6 rows and 7 columns for Connect-4. And create a 2D array to represent the board. and initialize the 2D array with zeros, which represents an empty cell.
2. Modify the board: Implement methods to allow players to place their pieces on the board, and to check if a certain location on the board is empty or occupied.
3. Update the board: Create a method that updates the board with a player's move. The method should take a player's piece (either "1" or "2") and the column number to place the piece.
4. Check for win conditions: Implement methods to check if a player has won the game by getting four pieces in a row horizontally, vertically, or diagonally.
5. Create a method that Draw the current state of the board to the pygame window, with each row and column labeled.
6. Initialize the board: Create a method to initialize the board with empty spaces or zeros, which represents an empty cell.
7. Test the class: Test the class with different scenarios to ensure that it's working correctly. You can create a simple game loop that alternates between the players and prompts them to enter their moves until the game ends.

Implementation:

```
import pygame
import numpy as np

class Board:
    """
    This class represents the Board for connect - 4
    """
    def __init__(self, rows, columns):
        self.rows = rows
        self.columns = columns
        self.cell_size = 90
        self.radius = 30
                # creates a 2D array that represents the game board
        self.grid = np.zeros((self.row, self.column))

    def draw(self, screen):

        # Draw the main board
        for c in range(7):
            for r in range(6):
                pygame.draw.rect(screen, (222, 226, 230), (c*self.cell_size + 20, r*self.cell_size+self.cell_size,
self.cell_size, self.cell_size))
                pygame.draw.circle(screen, (255,255,255), (int(c*self.cell_size+self.cell_size/2) + 20,
int(r*self.cell_size+self.cell_size+self.cell_size/2)), self.radius)

        # Draw the game pieces
        for c in range(7):
            for r in range(6):
                if self.grid[r][c] == 1:
                    pygame.draw.circle(screen, (24, 188, 156), (int(c*self.cell_size+self.cell_size/2) + 20 , 630-
int(r*self.cell_size+self.cell_size/2)), self.radius)
                elif self.grid[r][c] == 2:
                    pygame.draw.circle(screen, (44, 62, 80), (int(c*self.cell_size+self.cell_size/2) + 20, 630-
int(r*self.cell_size+self.cell_size/2)), self.radius)

    def is_valid_move(self, column):
                    # checks whether a given move is valid or not
        return self.grid[5][column] == 0

    def drop_piece(self, row, column, piece):
        # places a game piece (e.g., a colored disc) in the lowest
        self.grid[row][column] = piece

    def get_next_empty_row(self, col):
                    # returns the index of the lowest empty row in the selected column
        for r in range(6):
            if self.grid[r][col] == 0:
                return r

    def get_winner(self):
        """
        checks if a given move has resulted in a player
```

```python
        winning the game by connecting four game pieces of the same color
        vertically, horizontally, or diagonally.
        """

        # Check horizontally
        for i in range(self.rows):
            for j in range(self.columns - 3):
                if self.grid[i][j] == self.grid[i][j+1] == self.grid[i][j+2] == self.grid[i][j+3] != 0:
                    # return self.grid[i][j]
                    return True

        # Check  vertically
        for i in range(self.rows - 3):
            for j in range(self.columns):
                if self.grid[i][j] == self.grid[i+1][j] == self.grid[i+2][j] == self.grid[i+3][j] != 0:
                    # return self.grid[i][j]
                    return True

        # Check diagonally
        for i in range(self.rows - 3):
            for j in range(self.columns - 3):
                if self.grid[i][j] == self.grid[i+1][j+1] == self.grid[i+2][j+2] == self.grid[i+3][j+3] != 0:
                    # return self.grid[i][j]
                    return True

        for i in range(3, self.rows):
            for j in range(self.columns - 3):
                if self.grid[i][j] == self.grid[i-1][j+1] == self.grid[i-2][j+2] == self.grid[i-3][j+3] != 0:
                    # return self.grid[i][j]
                    return True


    def is_full(self):
        # retrun the board is full or not
        for i in range(6):
            for j in range(7):
                if self.grid[i][j] == 0:
                    return False
        return True

    def reset(self):
        # reset the game board
        self.grid = [[0 for _ in range(self.columns)] for _ in range(self.rows)]
```

Explanation:

The **Board** class represents the game board for Connect-4 game. It has the following methods:

1. **__init__(self, rows, columns)**: The constructor method initializes the game board by creating a 2D array of zeros with dimensions **rows** x **columns**. It also initializes several other variables related to the game board, such as the cell size, radius, and colors.
2. **draw(self, screen)**: This method is responsible for drawing the game board on the screen. It loops over the rows and columns of the board and uses **pygame** functions to draw rectangles and circles.
3. **is_valid_move(self, column)**: This method checks whether a given move (i.e., placing a game piece in a column) is valid or not. It returns **True** if the move is valid (i.e., the bottom row in the selected column is empty), and **False** otherwise.
4. **drop_piece(self, row, column, piece)**: This method places a game piece (e.g., a colored disc) in the lowest empty row of the selected column.
5. **get_next_empty_row(self, col)**: This method returns the index of the lowest empty row in the selected column.
6. **get_winner(self)**: This method checks if a given move has resulted in a player winning the game by connecting four game pieces of the same color vertically, horizontally, or diagonally. It returns **True** if a player has won and **False** otherwise.
7. **is_full(self)**: This method checks if the game board is full or not. It returns **True** if the board is full and **False** otherwise.
8. **reset(self)**: This method resets the game board by setting all elements of the 2D array to zero.

Overall, the **Board** class provides the basic functionality required to play Connect-4, such as checking for valid moves, placing game pieces, checking for a winner, and resetting the game board.

Develop the Player class:

The Player class will handle the players' moves, such as selecting the column to place their game piece.

Date:

Implementation:

```
class Player:
    def __init__(self, name):
        self.name = name

    def make_move(self, board, column, piece):
        if board.is_valid_move(column):
            row = board.get_next_empty_row(column)
            board.drop_piece(row, column, piece)
            return True
        return False
```

Explanation:

1. **__init__(self, name)**: This is the constructor method that initializes a **Player** object with a **name** attribute. The **name** attribute is passed as an argument when the **Player** object is created. For example, **player1 = Player('Compcode')**.
2. **make_move(self, board, column, piece)**: This method allows the player to make a move on the Connect-4 board. It takes three arguments: **board**, **column**, and **piece**. The **board** argument is an instance of the **Board** class, which represents the Connect-4 board. The **column** argument is an integer that represents the column in which the player wants to drop their piece. The **piece** argument is a string that represents the player's game piece (e.g., '1' or '2').
3. Inside the method, it first checks if the move is valid by calling the **is_valid_move(column)** method of the **board** object. If the move is valid, it calls the **get_next_empty_row(column)** method to get the row where the player's piece should be dropped. Then, it calls the **drop_piece(row, column, piece)** method of the **board** object to update the board with the player's move. Finally, it returns **True** if the move was successful and **False** if the move was invalid.

In summary, the **Player** class contains a constructor method that initializes a **Player** object with a name attribute, and a **make_move** method that allows the player to make a move on the Connect-4 board by updating the board with the player's move.

Develop the Game class:

Date:

Implementation:

```
import pygame
from borad import Board
from player import Player

class Game:

    def __init__(self, player1, player2):

        pygame.init()
        self.board = Board(6, 7)
        self.players = [Player(player1), Player(player2)]
        self.current_player = self.players[0]
        self.piece = 1
        self.game_over = False
        self.winner = None
        self.clock = pygame.time.Clock()
        self.font = pygame.font.SysFont("comicsansms", 30)
        self.font2 = pygame.font.SysFont("comicsansms", 80)
        self.timer_event = pygame.USEREVENT+1
        pygame.time.set_timer(self.timer_event, 1000)
        self.timer_paused = False
        self.time_left = 120
        self.width = 7 * 90 + 300
        self.height = (6+1) * 90

    def switch_player(self):
        if self.current_player == self.players[0]:
            self.current_player = self.players[1]
            self.piece = 2
        else:
            self.current_player = self.players[0]
            self.piece = 1
        self.time_left = 120

    def check_winner(self,screen):
        if self.board.get_winner():
            self.game_over = True
            self.board.draw(screen) #issues
            self.winner = self.current_player.name
            return True
        elif self.board.is_full():
            self.game_over = True
            return True
        return False

    def draw_timer_button(self,screen):
        # Draw the button
        self.button_width = 180
        self.button_height = 60
        self.button_x = screen.get_rect().right - (self.button_width+40)
        self.button_y = screen.get_rect().centery
        button_color = (0, 255, 0) if not self.timer_paused else (255, 0, 0)
        button_text = "Pause" if not self.timer_paused else "Resume"
        button_text_color = (255, 255, 255)
        button_text_pos = (self.button_x + self.button_width // 2, self.button_y + self.button_height // 2)
        button_text_surface = self.font.render(button_text, True, button_text_color)
        button_text_rect = button_text_surface.get_rect(center=button_text_pos)
        pygame.draw.rect(screen, button_color, (self.button_x, self.button_y, self.button_width,
self.button_height))
        screen.blit(button_text_surface, button_text_rect)

    def draw_players(self,screen):
        # Draw the players name
        width = 160
        height = 50
        x = screen.get_rect().right - (self.button_width+40)
        y1 = screen.get_rect().centery - 180
        y2 = screen.get_rect().centery + 180
        text1 = self.players[0].name
        text2 = self.players[1].name
        color_active = (255, 255, 255)
        color_pause = (50, 50, 50)
        text_pos1 = (x+width//2, y1+height//2)
        text_pos2 = (x+width//2, y2+height//2)
        if self.piece == 1:
            text_surface1 = self.font.render(text1, True, color_active)
            text_surface2 = self.font.render(text2, True, color_pause)
        else:
            text_surface1 = self.font.render(text1, True, color_pause)
            text_surface2 = self.font.render(text2, True, color_active)

        text_rect1 = text_surface1.get_rect(center=text_pos1)
        text_rect2 = text_surface2.get_rect(center=text_pos2)
        if self.piece == 1:
            pygame.draw.rect(screen, (69, 123, 157), (x, y1, width, height))
            pygame.draw.rect(screen, (229, 229, 229), (x, y2, width, height))
            pygame.draw.circle(screen, (24, 188, 156), (x-20, y1+25), 10)
            pygame.draw.circle(screen, (255, 255, 255), (x-20, y2+25), 10)
        else:
            pygame.draw.rect(screen, (229, 229, 229), (x, y1, width, height))
            pygame.draw.rect(screen, (69, 123, 157), (x, y2, width, height))
            pygame.draw.circle(screen, (44, 62, 80), (x-20, y2+25), 10)
            pygame.draw.circle(screen, (255, 255, 255), (x-20, y1+25), 10)


        screen.blit(text_surface1, text_rect1)
        screen.blit(text_surface2, text_rect2)




    def run(self):
        pygame.init()
        screen = pygame.display.set_mode((self.width, self.height))
        pygame.display.set_caption("Connect-4")
        screen.fill((255, 255, 255))
        self.board.draw(screen)

        # Start the main loop for the game
        while not self.game_over:

            # screen.fill((255, 255, 255))
            # self.board.draw(screen)

            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    quit()



                if event.type == self.timer_event:
                    if not self.timer_paused:
                        self.time_left -= 1
                elif event.type == pygame.MOUSEBUTTONDOWN:
                    # if event.key == pygame.K_SPACE:
                    #     self.timer_paused = not self.timer_paused
                    if self.button_x <= mouse_pos[0] <= self.button_x + self.button_width and \
                            self.button_y <= mouse_pos[1] <= self.button_y + self.button_height:
                        # Pause/resume the timer
                        self.timer_paused = not self.timer_paused

                if event.type == pygame.MOUSEMOTION:
                    mouse_pos = pygame.mouse.get_pos()
                    if (mouse_pos[0] >= 20 and mouse_pos[0] <= (int(6*90+90/2) + 40)):
                        pygame.draw.rect(screen, (255,255,255), (0,0, 930, 90))
                        posx = event.pos[0]
                        if (posx >= 0 and posx <= (int(6*90+90/2) + 20)) and (self.piece%2)==1:
                            pygame.draw.circle(screen, (24, 188, 156), (posx+30, int(90/2)), 30)
                        elif (posx >= 0 and posx <= (int(6*90+90/2) + 20)) and (self.piece%2)==0:
                            pygame.draw.circle(screen, (44, 62, 80), (posx+30, int(90/2)), 30)
                pygame.display.update()

                if event.type == pygame.MOUSEBUTTONDOWN and not self.timer_paused:
                    mouse_pos = pygame.mouse.get_pos()
                    if (mouse_pos[0] >= 20 and mouse_pos[0] <= (int(6*90+90/2) + 40)):
                        column = mouse_pos[0]//90
                        pygame.draw.rect(screen, (255,255,255), (0,0, 930, 90))
                        if self.current_player.make_move(self.board, column, self.piece):
                            if self.check_winner(screen):
                                break
                            self.switch_player()

            timer_text = self.font.render(f"Time Left: {self.time_left} seconds", True, (0, 0, 0))
            # Remove Previous drawn screen
            pygame.draw.rect(screen, (255,255,255), (680,280, 300, 100))
            screen.blit(timer_text, (680, 280))
            self.draw_timer_button(screen)
            self.draw_players(screen)
            self.board.draw(screen)
            pygame.display.update()
            # self.clock.tick(60)

        winner_text = self.font2.render(f"Winner: {self.winner}", True, (0, 0, 0))
        screen.blit(winner_text, (40, 10))
        pygame.display.update()
        pygame.time.delay(5000)
        pygame.quit()
```

Explanation:

The **Game** class will handle the game's overall logic, such as checking for a win or a draw and alternating turns between the players.

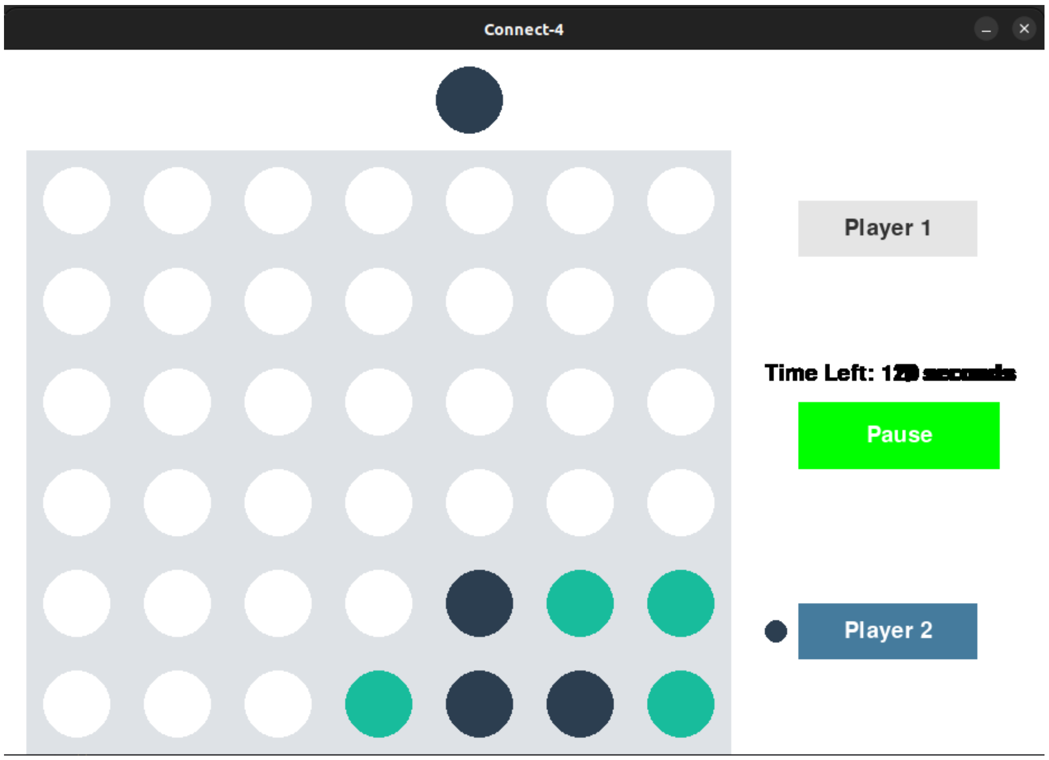
1. **__init__(self, player1, player2)**: This method is the constructor of the **Game** class. It initializes the game with a board of size 6x7 and two players with their respective names. It also sets the initial player as **player1**, the piece value as 1, and sets the game as not over. It creates a clock object to manage the game speed, creates two font objects to display the text on the screen, sets up a timer event and initializes the time left for the game. Finally, it sets up the width and height of the game screen.
2. **switch_player(self)**: This method switches the current player after each turn. If the current player is player 1, it will switch to player 2, and vice versa. It also sets the time left for the game as 120 seconds (about 2 minutes).
3. **check_winner(self, screen)**: This method checks if there is a winner in the game by calling the **get_winner()** method of the board object. If there is a winner, it sets the **game_over** variable as **True**, displays the winner's name on the screen and returns **True**. If the board is full and there is no winner, it sets the **game_over** variable as **True** and returns **True**. Otherwise, it returns **False**.
4. **draw_timer_button(self, screen)**: This method draws the timer button on the screen. It creates a rectangle with two different colors to represent whether the timer is paused or running. It also displays the text "Pause" or "Resume" on the button, depending on the state of the timer.
5. **draw_players(self, screen)**: This method draws the names of the two players on the screen. It also highlights the active player's name in a different color than the inactive player's name. It also displays a small circle next to the active player's name.
6. **run(self)**: This is the main method of the **Game** class. It creates a Pygame window and sets up the game environment. It also runs the main loop of the game, where it listens for events and takes action based on the events. It displays the game board, the player names, and the timer button on the screen. It also updates the timer and the screen based on the events. Finally, it quits the Pygame window when the game is over.

Game(“Player 1”, “Player 2”).run()

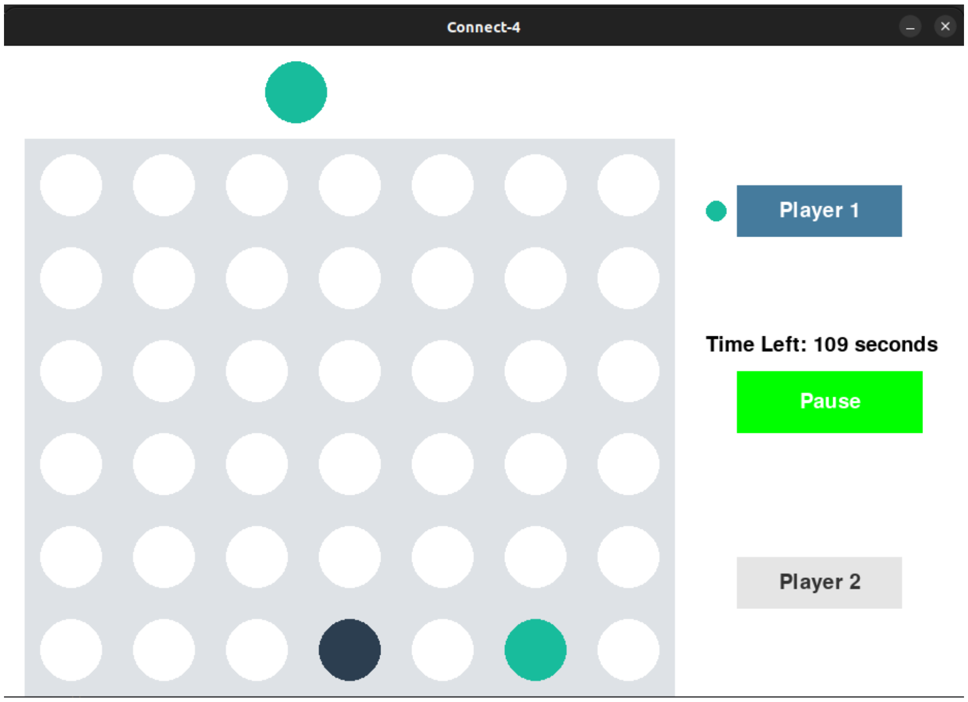
Output:

# Screenshot - 1

Testing



#screenshot 2



Testing

#	Test	Result	Evidence
1.	Load the external game files b (oard.py, player.py)	Pass – The Player Class and Board Class from player.py and board.py are loaded and played	Screenshot 1
2.	Display the game timer	Fail-(timer can't remove previous drawn time) To solve this issue we just draw a rectangle same color as the background color then draw the latest time to display	Screenshot 1 and screenshot 2