МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ЛЬВІВСЬКА ПОЛІТЕХНІКА

Автоматизоване проектування комп’ютерних систем

Task 4. Create doxygen documentation

Виконав:
ст. гр КІ - 401
Демчук Д. П.

Прийняв: Федак П. Р.

2024

Опис теми

Для виконання завдання №4 потрібно виконати наступні задачі:

1. Додати doxygen коментарі для всіх публічних функцій, класів, властивостей, полів...
2. Створити документації на основі коментарів doxygen

Теоретичні відомості

Doxygen — це інструмент для автоматичної генерації документації з вихідного коду програмного забезпечення. Він підтримує різні мови програмування, такі як C++, C, Java, Python, та інші. Doxygen аналізує коментарі в коді та генерує структуровану документацію у вигляді HTML, PDF, чи інших форматів, що спрощує розуміння і підтримку проєкту.

Doxyfile — це конфігураційний файл, який використовується Doxygen для визначення параметрів генерації документації. Він містить налаштування, такі як формат вихідного документа, директорії для сканування, фільтри, опції форматування та інші параметри, які керують процесом створення документації.

Виконання завдання

1. Додав doxygen коментарі для файлів серверної та клієнтської частин, тестів.

main.py

```
import serial
import time
import threading
import json
import os
```

```
CONFIG_FILE = 'config/game_config.json'


def setup_serial_port():
    """!
    @brief Sets up the serial port for communication.
    @details Prompts the user to enter the serial port (e.g., /dev/ttyUSB0 or
COM3) and
             returns a serial connection object.
    @return Serial connection object.
    @throws serial.SerialException if the serial port cannot be opened.
    """
    try:
        port = input("Enter the serial port (e.g., /dev/ttyUSB0 or COM3): ")
        return serial.Serial(port, 9600, timeout=1)
    except serial.SerialException as e:
        print(f"Error: {e}")
        exit(1)


def send_message(message, ser):
    """!
    @brief Sends a message over the serial connection.
    @details Encodes the message and sends it via the given serial connection.
    @param message The message to send.
    @param ser The serial connection object.
    @throws serial.SerialException if sending the message fails.
    """
    try:
        ser.write((message + '\n').encode())
    except serial.SerialException as e:
        print(f"Error sending message: {e}")


def receive_message(ser):
    """!
    @brief Receives a message from the serial connection.
    @details Reads a line from the serial connection, decodes it, and strips it
of any
             unnecessary whitespace or errors.
    @param ser The serial connection object.
    @return The received message or None if an error occurs.
    @throws serial.SerialException if receiving the message fails.
    """
    try:
        received = ser.readline().decode('utf-8', errors='ignore').strip()
```

```
        if received:
            print(received)
        return received
    except serial.SerialException as e:
        print(f"Error receiving message: {e}")
        return None


def receive_multiple_messages(ser, count):
    """!
    @brief Receives multiple messages from the serial connection.
    @details Calls the receive_message function multiple times to collect a list
of received messages.
    @param ser The serial connection object.
    @param count The number of messages to receive.
    @return A list of received messages.
    """
    messages = []
    for _ in range(count):
        message = receive_message(ser)
        if message:
            messages.append(message)
    return messages


def user_input_thread(ser):
    """!
    @brief Handles user input in a separate thread.
    @details Continuously listens for user input. Depending on the input, the
user can send messages
             or save/load game configurations. The thread will exit if the user
types 'exit'.
    @param ser The serial connection object.
    """
    global can_input
    while True:
        if can_input:
            user_message = input()
            if user_message.lower() == 'exit':
                print("Exiting...")
                global exit_program
                exit_program = True
                break
            elif user_message.lower().startswith('save'):
                save_game_config(user_message)
            elif user_message.lower().startswith('load'):
                file_path = input("Enter the path to the configuration file: ")
```

```
                load_game_config(file_path, ser)
            send_message(user_message, ser)
            can_input = False


def monitor_incoming_messages(ser):
    """!
    @brief Monitors incoming messages on the serial connection in a separate
thread.
    @details Continuously checks for messages from the serial connection and
updates the can_input
             flag when new data is received.
    @param ser The serial connection object.
    """
    global can_input
    global last_received_time
    while not exit_program:
        received = receive_message(ser)
        if received:
            last_received_time = time.time()
            if not can_input:
                can_input = True


def save_game_config(message):
    """!
    @brief Saves the game configuration to a JSON file.
    @details Saves game mode, player symbols, and other configurations to the
`game_config.json` file.
    @param message The message containing the configuration details.
    @throws Exception if saving the configuration fails.
    """
    config = {
        "gameMode": 0,
        "player1Symbol": 'X',
        "player2Symbol": 'O'
    }

    try:
        params = message.split()
        if len(params) == 2 and params[1] in ['0', '1', '2']:
            config["gameMode"] = int(params[1])

        with open(CONFIG_FILE, 'w') as f:
            json.dump(config, f)
        print(f"Configuration saved to {CONFIG_FILE}")
    except Exception as e:
```

```
        print(f"Error saving configuration: {e}")


def load_game_config(file_path, ser):
    """!
    @brief Loads the game configuration from a JSON file.
    @details Reads the configuration from a file and sends it to the serial
device. If the file is not
             found, prompts the user to provide a valid path.
    @param file_path The path to the configuration file.
    @param ser The serial connection object.
    @throws Exception if loading the configuration fails.
    """
    try:
        if os.path.exists(file_path):
            with open(file_path, 'r') as f:
                config = json.load(f)
                game_mode = config.get("gameMode", 0)
                player1_symbol = config.get("player1Symbol", 'X')
                player2_symbol = config.get("player2Symbol", 'O')

                print(f"Game Mode: {game_mode}")
                print(f"Player 1 Symbol: {player1_symbol}")
                print(f"Player 2 Symbol: {player2_symbol}")

                json_message = {
                    "gameMode": game_mode,
                    "player1Symbol": player1_symbol,
                    "player2Symbol": player2_symbol
                }

                json_str = json.dumps(json_message)
                print(json_str)

                send_message(json_str, ser)
        else:
            print("Configuration file not found. Please provide a valid path.")
    except Exception as e:
        print(f"Error loading configuration: {e}")


if __name__ == "__main__":
    """!
    @brief Main entry point of the program.
    @details Sets up the serial port and starts two threads: one for monitoring
incoming messages
             and one for handling user input. The program will keep running
```

```
until the exit flag is set.
    """
    ser = setup_serial_port()
    can_input = True
    exit_program = False
    last_received_time = time.time()

    threading.Thread(target=monitor_incoming_messages, args=(ser,),
daemon=True).start()
    threading.Thread(target=user_input_thread, args=(ser,), daemon=True).start()

    try:
        while not exit_program:
            if time.time() - last_received_time ⩾ 1 and can_input:
                pass
            else:
                time.sleep(0.1)
    except KeyboardInterrupt:
        print("Exit!")
    finally:
        if ser.is_open:
            print("Closing serial port...")
            ser.close()
```

test_serial_communication.py

```
import pytest
from unittest.mock import patch, MagicMock
import serial
import sys
import os
sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__),
'..')))
from main import send_message, receive_message, save_game_config,
load_game_config


def test_send_message():
    """!
    @brief Tests the send_message function.
    @details This test verifies that the send_message function correctly calls
the serial
             port's write method with the expected message in the correct format
(encoded as bytes).
    """
```

```
    mock_serial = MagicMock(spec=serial.Serial)
    send_message("Hello", mock_serial)
    mock_serial.write.assert_called_with(b"Hello\n")


def test_receive_message():
    """!
    @brief Tests the receive_message function.
    @details This test simulates receiving a message from the serial connection
and checks
             that the function returns the correct decoded string.
    """
    mock_serial = MagicMock(spec=serial.Serial)
    mock_serial.readline.return_value = b"Test Message\n"
    result = receive_message(mock_serial)
    assert result == "Test Message"


def test_receive_empty_message():
    """!
    @brief Tests the receive_message function with an empty message.
    @details This test simulates receiving an empty message (just a newline) and
ensures that
             the function returns an empty string.
    """
    mock_serial = MagicMock(spec=serial.Serial)
    mock_serial.readline.return_value = b"\n"
    result = receive_message(mock_serial)
    assert result == ""


@patch('builtins.input', return_value='COM3')
def test_serial_port(mock_input):
    """!
    @brief Tests serial port setup.
    @details This test simulates user input for selecting the serial port and
verifies that
             the serial port configuration is correctly set to the mocked input
value.
    """
    mock_serial = MagicMock(spec=serial.Serial)
    mock_serial.portstr = 'COM3'
    port = 'COM3'
    ser = mock_serial
    assert ser.portstr == port
```

task3.ino

```
#include <Arduino.h>
#include <ArduinoJson.h>

char board[3][3];
bool gameActive = false;
String player1Symbol = "X";
String player2Symbol = "O";
String currentPlayer = "X";
int gameMode = 0;

/**
 * @brief Structure to hold the game configuration.
 */
struct GameConfig {
  int gameMode;           ///< The game mode (e.g., single-player or
multiplayer)
  String player1Symbol;   ///< Symbol for Player 1 (e.g., "X")
  String player2Symbol;   ///< Symbol for Player 2 (e.g., "O")
  String currentPlayer;   ///< Current player's symbol (e.g., "X" or "O")
};

/**
 * @brief Saves the current game configuration to Serial as a JSON string.
 *
 * @param config The GameConfig struct holding the current configuration.
 */
void saveConfig(const GameConfig &config) {
  StaticJsonDocument<200> doc;
  doc["gameMode"] = config.gameMode;
  doc["player1Symbol"] = config.player1Symbol;
  doc["player2Symbol"] = config.player2Symbol;
  doc["currentPlayer"] = config.currentPlayer;

  String output;
  serializeJson(doc, output);
  Serial.println(output);
}

/**
 * @brief Loads the game configuration from a JSON string.
 *
 * @param jsonConfig The JSON string representing the saved game configuration.
 */
```

```
void loadConfig(String jsonConfig) {
  StaticJsonDocument<200> doc;
  DeserializationError error = deserializeJson(doc, jsonConfig);

  if (error) {
    Serial.println("Failed to load configuration");
    return;
  }

  if (doc.containsKey("gameMode")) {
    gameMode = doc["gameMode"].as<int>();
  } else {
    Serial.println("gameMode not found");
    return;
  }

  if (doc.containsKey("player1Symbol") && doc["player1Symbol"].is<String>()) {
    player1Symbol = doc["player1Symbol"].as<String>();
  } else {
    Serial.println("player1Symbol not found or invalid");
    return;
  }

  if (doc.containsKey("player2Symbol") && doc["player2Symbol"].is<String>()) {
    player2Symbol = doc["player2Symbol"].as<String>();
  } else {
    Serial.println("player2Symbol not found or invalid");
    return;
  }

  Serial.println("Configuration loaded!");
}

/**
 * @brief Initializes the game board with empty spaces.
 */
void initializeBoard() {
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
      board[i][j] = ' ';
    }
  }
}

/**
 * @brief Prints the current state of the game board.
 */
```

```
void printBoard() {
  String boardState = "Board state:\n";

  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
      if (board[i][j] == 'X' || board[i][j] == 'O') {
        boardState += board[i][j];
      } else {
        boardState += '.';
      }
      if (j < 2) boardState += "|";
    }
    if (i < 2) boardState += "\n-+-+-\n";
    else boardState += "\n";
  }

  Serial.println(boardState);
}

/**
 * @brief Checks if a given player has won the game.
 *
 * @param player The symbol of the player ('X' or 'O').
 * @return true if the player has won, false otherwise.
 */
bool checkWin(char player) {
  for (int i = 0; i < 3; i++) {
    if ((board[i][0] == player && board[i][1] == player && board[i][2] ==
player) ||
        (board[0][i] == player && board[1][i] == player && board[2][i] ==
player)) {
      return true;
    }
  }

  if ((board[0][0] == player && board[1][1] == player && board[2][2] == player)
||
      (board[0][2] == player && board[1][1] == player && board[2][0] == player))
{
    return true;
  }

  return false;
}

/**
 * @brief Checks if the game board is full (no empty spaces).
```

```
 *
 * @return true if the board is full, false otherwise.
 */
bool isBoardFull() {
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
      if (board[i][j] == ' ') {
        return false;
      }
    }
  }
  return true;
}

/**
 * @brief AI makes a move on the game board.
 *
 * The AI either blocks the opponent's winning move or plays randomly.
 *
 * @param aiSymbol The symbol representing the AI ('X' or 'O').
 */
void aiMove(char aiSymbol) {
  if (blockOpponentMove(aiSymbol == 'X' ? 'O' : 'X')) {
    return;
  }

  int startX = random(3);
  int startY = random(3);

  if (random(2) == 0) {
    startX = 0;
    startY = 0;
  }

  for (int i = startX; i < 3; i++) {
    for (int j = startY; j < 3; j++) {
      if (board[i][j] == ' ') {
        board[i][j] = aiSymbol;
        Serial.println("AI played at: " + String(i + 1) + " " + String(j + 1));
        return;
      }
    }
  }

  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
      if (board[i][j] == ' ') {
```

```
        board[i][j] = aiSymbol;
        Serial.println("AI played randomly at: " + String(i + 1) + " " +
String(j + 1));
        return;
      }
    }
  }
}

/**
 * @brief Attempts to block the opponent from making a winning move.
 *
 * @param opponent The symbol of the opponent ('X' or 'O').
 * @return true if the opponent's winning move is blocked, false otherwise.
 */
bool blockOpponentMove(char opponent) {
  for (int i = 0; i < 3; i++) {
    // Horizontal and vertical lines
    if (canBlock(i, 0, i, 1, i, 2, opponent)) {
      return true;
    }
    if (canBlock(0, i, 1, i, 2, i, opponent)) {
      return true;
    }
  }

  if (canBlock(0, 0, 1, 1, 2, 2, opponent)) {
    return true;
  }
  if (canBlock(0, 2, 1, 1, 2, 0, opponent)) {
    return true;
  }

  return false;
}

/**
 * @brief Checks if the AI can block the opponent's winning move.
 *
 * This function checks all three possible positions in a row, column, or
diagonal
 * where the opponent has two symbols and the third position is empty. If such a
 * position exists, the AI will block the opponent's winning move by placing its
 * symbol ('O') in that position.
 *
 * @param x1 The x-coordinate of the first position in the line.
 * @param y1 The y-coordinate of the first position in the line.
```

```
 * @param x2 The x-coordinate of the second position in the line.
 * @param y2 The y-coordinate of the second position in the line.
 * @param x3 The x-coordinate of the third position in the line.
 * @param y3 The y-coordinate of the third position in the line.
 * @param opponent The symbol of the opponent ('X' or 'O').
 * @return true if the move was blocked; false otherwise.
 */
bool canBlock(int x1, int y1, int x2, int y2, int x3, int y3, char opponent) {
  if (board[x1][y1] == opponent && board[x2][y2] == opponent && board[x3][y3] ==
' ') {
    board[x3][y3] = 'O';
    Serial.println("AI blocked opponent's winning move at: " + String(x3 + 1) +
" " + String(y3 + 1));
    return true;
  }
  if (board[x1][y1] == opponent && board[x2][y2] == ' ' && board[x3][y3] ==
opponent) {
    board[x2][y2] = 'O';
    Serial.println("AI blocked opponent's winning move at: " + String(x2 + 1) +
" " + String(y2 + 1));
    return true;
  }
  if (board[x1][y1] == ' ' && board[x2][y2] == opponent && board[x3][y3] ==
opponent) {
    board[x1][y1] = 'O';
    Serial.println("AI blocked opponent's winning move at: " + String(x1 + 1) +
" " + String(y1 + 1));
    return true;
  }

  return false;
}

/**
 * @brief Processes a move made by the player or AI.
 *
 * This function processes a move by either a player (in player vs player mode)
or AI
 * (in player vs AI mode). It updates the board, checks for a win, a draw, or
proceeds
 * to the next turn.
 *
 * In player vs player mode, the current player can choose any empty spot on the
board.
 * In player vs AI mode, the AI makes a move after the player.
 *
 * @param input A string representing the move, formatted as "row,col", e.g.,
```

```
"1,1" for the top-left position.
 *
 * @note The game will stop if there is a winner or the board is full (draw).
 */
void processMove(String input) {
  int row = input[0] - '1';
  int col = input[2] - '1';

  if (row ≥ 0 && row < 3 && col ≥ 0 && col < 3 && board[row][col] == ' ') {
    if (gameMode == 1) {
      board[row][col] = (currentPlayer == "X") ? 'X' : 'O';
    } else {
      board[row][col] = 'X';
    }
    printBoard();

    if (checkWin('X')) {
      Serial.println("Player X wins!");
      gameActive = false;
      return;
    }

    if (checkWin('O')) {
      Serial.println("Player O wins!");
      gameActive = false;
      return;
    }

    if (isBoardFull()) {
      Serial.println("It's a draw!");
      gameActive = false;
      return;
    }

    if (gameMode == 2) {
      aiMove(player1Symbol[0]);
      if (checkWin(player1Symbol[0])) {
        Serial.println("Player 1 (AI) wins!");
        gameActive = false;
        return;
      }
      aiMove(player2Symbol[0]);
      if (checkWin(player2Symbol[0])) {
        Serial.println("Player 2 (AI) wins!");
        gameActive = false;
        return;
      }
```

```
    }

    currentPlayer = (currentPlayer == "X") ? 'O' : 'X';
  } else {
    Serial.println("Invalid move, try again.");
  }
}

/**
 * @brief Initializes the serial communication.
 *
 * This function sets up the serial communication with a baud rate of 9600 to
allow
 * communication between the Arduino and the user through the serial monitor.
 * It is called once when the program starts.
 */
void setup() {
  Serial.begin(9600);
}

/**
 * @brief Main game loop that handles user input and game flow.
 *
 * This function continuously runs the game logic and handles player moves, game
state,
 * and communication via the serial interface. It listens for specific commands
to
 * start a new game, save or load the game configuration, change the game mode,
 * and process player or AI moves.
 *
 * - If a new game is started (via the "new" command), the board is initialized,
 *   and the players are asked to choose their symbols. The game proceeds in one
of
 *   the three game modes: Player vs Player, Player vs AI, or AI vs AI.
 * - If the "save" command is received, the current game configuration is saved.
 * - If the "modes" command is received, the game mode is set to the specified
value.
 * - If no game is active, the user is prompted to type 'new' to start a new
game.
 *
 * @note The loop runs continuously, processing user input and updating the game
state.
 */
void loop() {
  if (Serial.available() > 0) {
    String receivedMessage = Serial.readStringUntil('\n');  ///< Read the
incoming serial message
```

```
    receivedMessage.trim();  ///< Remove any trailing whitespace or newline
characters

    if (receivedMessage == "new") {
      initializeBoard();  ///< Initialize a new game board
      gameActive = true;  ///< Set the game state to active

      if (gameMode == 1) {
        Serial.println("Player 1, choose your symbol: X or O");
        currentPlayer = (random(2) == 0) ? 'X' : 'O';  ///< Randomly choose
Player 1's symbol
        player1Symbol = currentPlayer;
        player2Symbol = (currentPlayer == "X") ? 'O' : 'X';
        Serial.println("Player 1 is " + String(player1Symbol));
        Serial.println("Player 2 is " + String(player2Symbol));
      } else {
        currentPlayer = 'X';  ///< Set Player 1 to be 'X' in AI modes
      }

      Serial.println("New game started! " + String(currentPlayer) + " goes
first.");
      printBoard();  ///< Print the initial game board

      if (gameMode == 0) {  ///< Man vs AI mode
        while (gameActive) {
          if (currentPlayer == "X") {
            Serial.println("Your move, player (enter row and column):");
            while (Serial.available() == 0) { }
            String userMove = Serial.readStringUntil('\n');  ///< Read the
user's move
            processMove(userMove);  ///< Process the user's move
            printBoard();  ///< Print the updated board

            if (checkWin('X')) {
              Serial.println("Player X wins!");
              gameActive = false;
              break;
            }
            if (isBoardFull()) {
              Serial.println("It's a draw!");
              gameActive = false;
              break;
            }

            currentPlayer = 'O';  ///< Switch to Player 2 (AI)
          } else {
            aiMove('O');  ///< AI makes its move
```

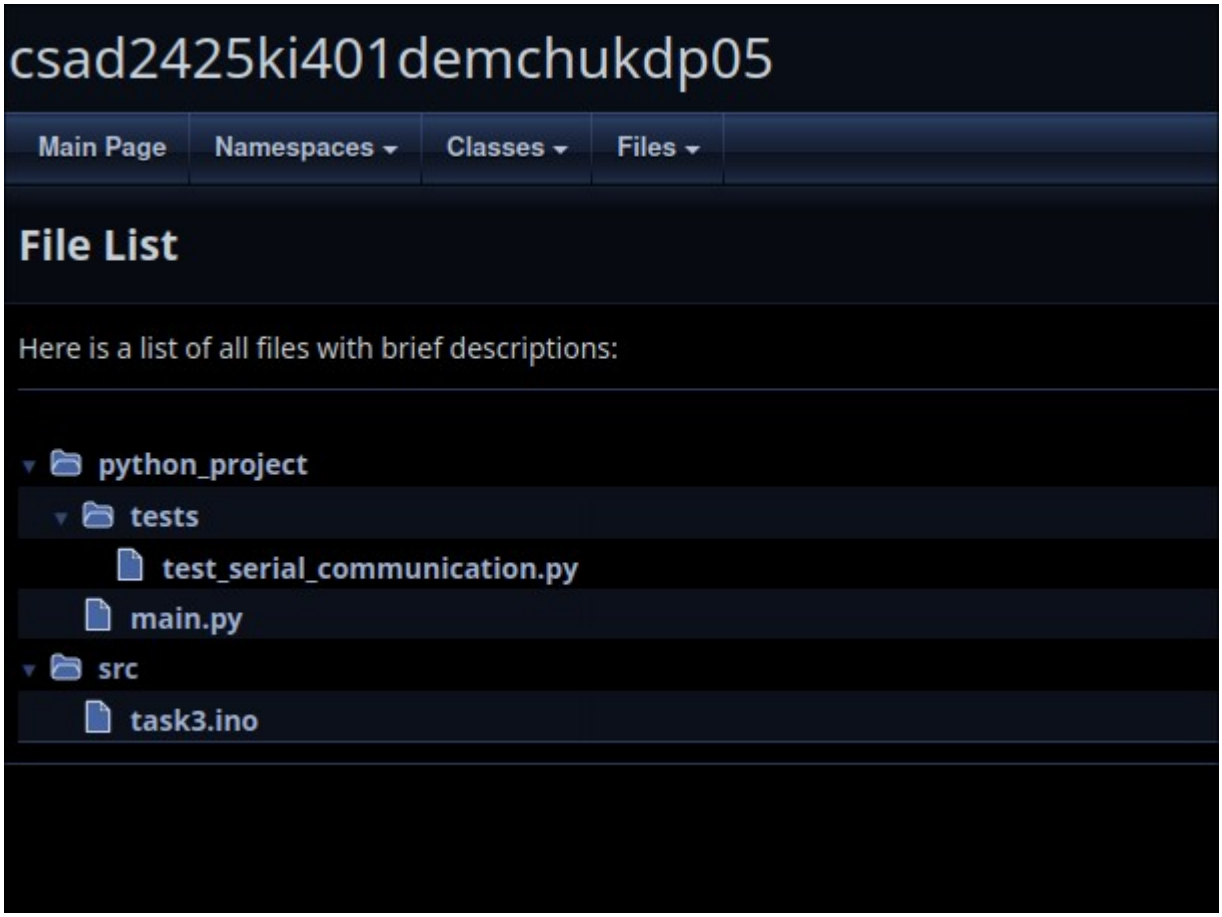
```
          printBoard();  ///< Print the updated board
          if (checkWin('O')) {
            Serial.println("AI O wins!");
            gameActive = false;
            break;
          }
          if (isBoardFull()) {
            Serial.println("It's a draw!");
            gameActive = false;
            break;
          }

          currentPlayer = 'X';  ///< Switch to Player 1
        }
      }
    }
    else if (gameMode == 2) {  ///< AI vs AI mode
      while (gameActive) {
        aiMove('X');  ///< AI X makes its move
        printBoard();  ///< Print the updated board
        if (checkWin('X')) {
          Serial.println("AI X wins!");
          gameActive = false;
          break;
        }
        if (isBoardFull()) {
          Serial.println("It's a draw!");
          gameActive = false;
          break;
        }

        aiMove('O');  ///< AI O makes its move
        printBoard();  ///< Print the updated board
        if (checkWin('O')) {
          Serial.println("AI O wins!");
          gameActive = false;
          break;
        }
        if (isBoardFull()) {
          Serial.println("It's a draw!");
          gameActive = false;
          break;
        }
      }
    }
  } else if (receivedMessage.startsWith("save")) {
    GameConfig config = { gameMode, player1Symbol, player2Symbol,
```

```
currentPlayer };
      saveConfig(config);  ///< Save the current game configuration
    } else if (receivedMessage.startsWith("{")) {
      if (receivedMessage.length() > 0) {
          loadConfig(receivedMessage);  ///< Load the game configuration
      } else {
          Serial.println("No message received");
      }
    } else if (receivedMessage.startsWith("modes")) {
      if (receivedMessage == "modes 0") {
        gameMode = 0;
        Serial.println("Game mode: Man vs AI");
      } else if (receivedMessage == "modes 1") {
        gameMode = 1;
        Serial.println("Game mode: Man vs Man");
      } else if (receivedMessage == "modes 2") {
        gameMode = 2;
        Serial.println("Game mode: AI vs AI");
      }
    } else if (gameActive) {
      processMove(receivedMessage);  ///< Process the move if the game is active
    } else {
      Serial.println("No active game. Type 'new' to start.");
    }
  }
}
```

2. Згенерувати конфігураційни файл **Doxyfile** та вніс необхідні параметри.
3. Відкрив **index.html**:



Висновок

Під час виконання завдання №4 було згенеровано doxygen документацію.

Список використаних джерел

1. Doxygen Manual. "Introduction to Doxygen". https://doxygen.nl/manual/index.html.
2. Doxygen Manual. "Configuration (Doxyfile)". https://doxygen.nl/manual/config.html.