МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ЛЬВІВСЬКА ПОЛІТЕХНІКА

Автоматизоване проектування комп’ютерних систем

Task 4. Create doxygen documentation

Виконав:

ст. гр КІ - 401

Сторожков О. В.

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():
<<<<<<< HEAD
```

```
    """!
    @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.
    """
=======
>>>>>>> feature/develop/task3
    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):
<<<<<<< HEAD
    """!
    @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.
    """
=======
>>>>>>> feature/develop/task3
    try:
        ser.write((message + '\n').encode())
    except serial.SerialException as e:
        print(f"Error sending message: {e}")


def receive_message(ser):
<<<<<<< HEAD
    """!
    @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.
    """
=======
```

```
>>>>>>> feature/develop/task3
    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):
<<<<<<< HEAD
    """!
    @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.
    """
=======
>>>>>>> feature/develop/task3
    messages = []
    for _ in range(count):
        message = receive_message(ser)
        if message:
            messages.append(message)
    return messages


def user_input_thread(ser):
<<<<<<< HEAD
    """!
    @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.
    """
=======
>>>>>>> feature/develop/task3
    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.
*
This structure contains the current game mode, player symbols,
and the symbol of the current player.
*/
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.
*
This function serializes the current configuration of the game into a
JSON format and prints it to the Serial monitor for later use.
*
@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 a string configuration from a JSON document.
*
This helper function attempts to load the configuration values
for a given key from the provided JSON document.
*
@param doc The JSON document containing the configuration.
@param key The key to look for in the document.
@param value The value of the key will be stored in this parameter if found.
@return true if the key exists and is a string, false otherwise.
*/
bool loadStringConfig(JsonDocument& doc, const char* key, String& value) {
if (doc.containsKey(key) && doc[key].is<String>()) {
value = doc[key].as<String>();
return true;
}
Serial.println(String(key) + " not found or invalid");
return false;
}

/**
@brief Loads the game configuration from a JSON string.
*
This function deserializes a JSON string into a configuration object
and applies the values to the current game settings.
*
@param jsonConfig The JSON string representing the 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.
*
This function sets all the cells of the 3x3 board to empty (' '),
preparing the board for the start of a new game.
*/
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.
*
This function displays the current game board in a readable format,
showing the position of 'X' and 'O' symbols and empty spaces as '.'.
*/
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.
*
This function checks all rows, columns, and diagonals for a win condition
by comparing the board positions with the given player's symbol.
*
@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).
*
This function checks all cells of the board to determine if there are any
remaining empty spaces. If all cells are filled, it returns true, otherwise
false.
*
@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.
The AI will attempt to block the opponent's move if it can, otherwise it will
choose a random empty space.
*
@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 Blocks the opponent's winning move.
*
The AI checks the rows, columns, and diagonals to find an opportunity to block
the opponent from winning.
If a winning move is detected for the opponent, the AI places its symbol to
block the move.
*
@param opponent The symbol of the opponent ('X' or 'O').
@return true if the AI blocked an opponent's winning move, false otherwise.
*/
bool blockOpponentMove(char opponent) {
int row[3] = {0, 1, 2};
int col[3] = {0, 1, 2};

// Check rows and columns
for (int i = 0; i < 3; i++) {
int rowCoords[3][2] = {{row[i], 0}, {row[i], 1}, {row[i], 2}};
int colCoords[3][2] = {{0, col[i]}, {1, col[i]}, {2, col[i]}};

if (canBlock(rowCoords[0], rowCoords[1], rowCoords[2], opponent)) {
return true;
}
if (canBlock(colCoords[0], colCoords[1], colCoords[2], opponent)) {
return true;
}
}

// Check diagonals
int diag1[3][2] = {{0, 0}, {1, 1}, {2, 2}};
int diag2[3][2] = {{0, 2}, {1, 1}, {2, 0}};

if (canBlock(diag1[0], diag1[1], diag1[2], opponent)) {
return true;
}
if (canBlock(diag2[0], diag2[1], diag2[2], opponent)) {
return true;
}
```

```
return false;
}

/**
@brief Checks if a set of coordinates forms a winning move for the opponent.
*
This function checks if the opponent has two of their symbols in a row,
column, or diagonal
and can place their symbol in the remaining empty space to win.
*
@param coords1 The first coordinate in the set.
@param coords2 The second coordinate in the set.
@param coords3 The third coordinate in the set.
@param opponent The symbol of the opponent ('X' or 'O').
@return true if the opponent can win by placing a symbol at one of the
coordinates, false otherwise.
*/
bool isWinningMove(int coords1[2], int coords2[2], int coords3[2], char
opponent) {
return (board[coords1[0]][coords1[1]] == opponent &&
board[coords2[0]][coords2[1]] == opponent && board[coords3[0]][coords3[1]] ==
' ') ||
(board[coords1[0]][coords1[1]] == opponent && board[coords2[0]][coords2[1]] ==
' ' && board[coords3[0]][coords3[1]] == opponent) ||
(board[coords1[0]][coords1[1]] == ' ' && board[coords2[0]][coords2[1]] ==
opponent && board[coords3[0]][coords3[1]] == opponent);
}

/**
@brief Places a move on the game board to block the opponent.
*
This function places the AI's symbol on the game board to block the opponent's
winning move.
It also prints the coordinates where the AI blocked the opponent.
*
@param coords The coordinates where the AI will place its symbol.
@param symbol The symbol of the AI ('X' or 'O').
*/
void placeMove(int coords[2], char symbol) {
board[coords[0]][coords[1]] = symbol;
Serial.println("AI blocked opponent's winning move at: " + String(coords[0] +
1) + " " + String(coords[1] + 1));
}

/**
@brief Checks if a move can block an opponent's winning move.
*
```

```
This function checks if the opponent is one move away from winning in the
specified row, column, or diagonal.
If a blocking move is possible, it places the AI's symbol in the empty spot to
block the opponent.
*
@param coords1 The first coordinate in the set.
@param coords2 The second coordinate in the set.
@param coords3 The third coordinate in the set.
@param opponent The symbol of the opponent ('X' or 'O').
@return true if the move can block the opponent's winning move, false
otherwise.
*/
bool canBlock(int coords1[2], int coords2[2], int coords3[2], char opponent) {
if (isWinningMove(coords1, coords2, coords3, opponent)) {
if (board[coords1[0]][coords1[1]] == ' ') {
placeMove(coords1, 'O');
} else if (board[coords2[0]][coords2[1]] == ' ') {
placeMove(coords2, 'O');
} else {
placeMove(coords3, 'O');
}
return true;
}

return false;
}

/**
@brief Handles the player's move on the game board.
*
This function updates the board with the current player's symbol, prints the
board,
and checks for a win or draw condition.
*
@param row The row where the player is placing their symbol.
@param col The column where the player is placing their symbol.
*/
void handlePlayerMove(int row, int col) {
board[row][col] = (currentPlayer == "X") ? 'X' : 'O';
printBoard();

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

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

/**
@brief Handles the AI's move in the game.
*
This function controls the AI's move, where it can either block the opponent's
move or play randomly.
After the AI plays, it checks for a win condition.
*/
void handleAIMove() {
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;
}
}
}

/**
@brief Switches the turn to the next player.
*
This function switches the current player between "X" and "O".
*/
void switchPlayer() {
currentPlayer = (currentPlayer == "X") ? 'O' : 'X';
}

/**
@brief Checks if a move is valid (within bounds and not already occupied).
*
This function checks if the selected row and column are within the game
board's boundaries
and if the spot is not already occupied by a symbol.
*
@param row The row of the move.
```

```
@param col The column of the move.
@return true if the move is valid, false otherwise.
*/
bool isValidMove(int row, int col) {
return (row >= 0 && row < 3 && col >= 0 && col < 3 && board[row][col] == ' ');
}

/**
@brief Processes a player's move from the input string.
*
This function parses the input string, checks if the move is valid, and
updates the game state.
It handles both player and AI moves, switching players as needed.
*
@param input The input string representing the move in the form "row col"
(e.g., "1 1").
*/
void processMove(String input) {
int row = input[0] - '1';
int col = input[2] - '1';

if (isValidMove(row, col)) {
handlePlayerMove(row, col);

if (!gameActive) return;

handleAIMove();

if (gameActive) {
switchPlayer();
}
} 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 Initializes the game and sets up initial conditions.
*
This function initializes the game board, sets the game as active, and assigns
symbols
to the players depending on the game mode. For human vs. AI mode, player 1 is
asked to choose
their symbol, while in AI vs. AI mode, the game runs automatically.
*/
void initializeGame() {
initializeBoard();
gameActive = true;

if (gameMode == 1) {
Serial.println("Player 1, choose your symbol: X or O");
currentPlayer = (random(2) == 0) ? 'X' : 'O';
player1Symbol = currentPlayer;
player2Symbol = (currentPlayer == "X") ? 'O' : 'X';
Serial.println("Player 1 is " + String(player1Symbol));
Serial.println("Player 2 is " + String(player2Symbol));
} else {
currentPlayer = 'X';
}

Serial.println("New game started! " + String(currentPlayer) + " goes first.");
printBoard();
}

/**
@brief Checks if the player has won and prints the result.
*
This function checks if a given player symbol has won the game. If the player
has won,
it prints the winner's symbol and ends the game.
*
@param symbol The symbol of the player to check for a win ('X' or 'O').
@return true if the player has won, false otherwise.
*/
bool checkAndPrintWinner(char symbol) {
if (checkWin(symbol)) {
Serial.println(String(symbol) + " wins!");
gameActive = false;
return true;
}
return false;
}
```

```
/**
@brief Checks if the game is a draw and prints the result.
*
This function checks if the game is a draw by verifying if the board is full.
If the game is a draw, it prints a draw message and ends the game.
*
@return true if the game is a draw, false otherwise.
*/
bool checkAndPrintDraw() {
if (isBoardFull()) {
Serial.println("It's a draw!");
gameActive = false;
return true;
}
return false;
}

/**
@brief Handles the Human vs AI game mode.
*
This function manages the flow of a human player versus AI game. It prompts
the player
for their move and processes it. If the AI's move or the player's move results
in a win or a draw,
the game ends.
*/
void processHumanVsAI() {
while (gameActive) {
if (currentPlayer == "X") {
Serial.println("Your move, player (enter row and column):");
while (Serial.available() == 0) {}
String userMove = Serial.readStringUntil('\n');
processMove(userMove);
printBoard();

if (checkAndPrintWinner('X') || checkAndPrintDraw()) {
break;
}

currentPlayer = 'O';
} else {
aiMove('O');
printBoard();
if (checkAndPrintWinner('O') || checkAndPrintDraw()) {
break;
}
```

```
currentPlayer = 'X';
}
}
}

/**
@brief Handles the AI vs AI game mode.
*
This function runs an AI vs. AI game, where each AI takes turns making moves
automatically.
The game ends when one of the AIs wins or the game results in a draw.
*/
void processAIvsAI() {
while (gameActive) {
aiMove('X');
printBoard();
if (checkAndPrintWinner('X') || checkAndPrintDraw()) {
break;
}

aiMove('O');
printBoard();
if (checkAndPrintWinner('O') || checkAndPrintDraw()) {
break;
}
}
}

/**
@brief Processes the received message from the serial input.
*
This function processes various types of messages received from the user
through
the serial input. It handles game mode selection, starting new games, saving
game configuration,
and processing moves.
*
@param receivedMessage The message received from the serial input.
*/
void processReceivedMessage(String receivedMessage) {
if (receivedMessage == "new") {
initializeGame();

if (gameMode == 0) {
processHumanVsAI();
} else if (gameMode == 2) {
```

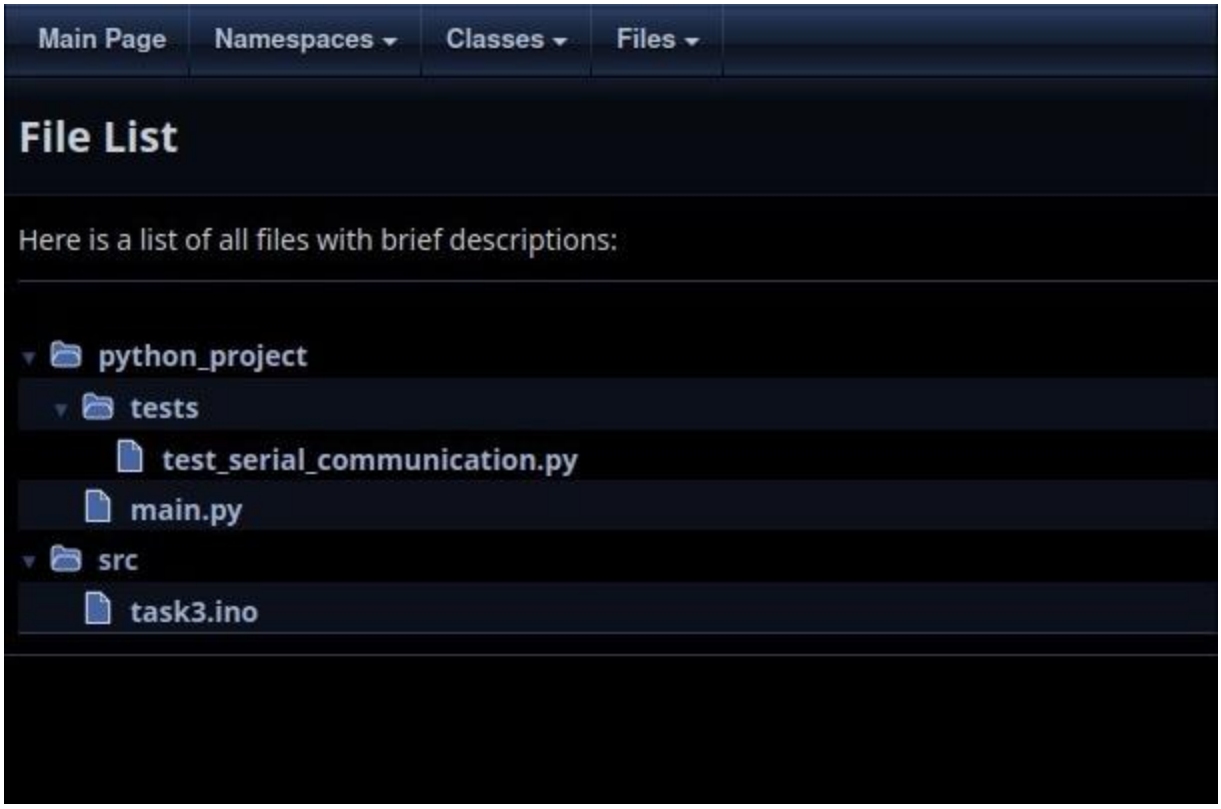
```
processAIvsAI();
}
} else if (receivedMessage.startsWith("save")) {
GameConfig config = { gameMode, player1Symbol, player2Symbol, currentPlayer };
saveConfig(config);
} else if (receivedMessage.startsWith("{")) {
if (receivedMessage.length() > 0) {
loadConfig(receivedMessage);
} else {
Serial.println("No message received");
}
} else if (receivedMessage.startsWith("modes")) {
handleGameMode(receivedMessage);
} else if (gameActive) {
processMove(receivedMessage);
} else {
Serial.println("No active game. Type 'new' to start.");
}
}

/**
@brief Handles the game mode selection based on the received message.
*
This function sets the game mode based on the received message. The game mode
determines
if the game is Human vs AI, Man vs Man, or AI vs AI.
*
@param receivedMessage The message containing the game mode selection.
*/
void handleGameMode(String receivedMessage) {
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");
}
}

/**
@brief Main loop that continuously checks for user input and processes it.
*
This function runs continuously in the loop. It listens for incoming serial
messages and
```

```
processes them accordingly, allowing for interaction with the game.
*/
void loop() {
if (Serial.available() > 0) {
String receivedMessage = Serial.readStringUntil('\n');
receivedMessage.trim();
processReceivedMessage(receivedMessage);
}
}
```

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.