Centre Name: Centre Number:

Task completed:

Date started	Date completed
2026-02-04	2026-02-04

Analysis

Try and create 3 or more key success criteria for your program.

Success Criteria:

1. User authentication – The program allows users to log in or register; credentials are stored (e.g. in users.csv) and checked before starting the game.

2. Correct chess rules – All pieces move according to standard rules (pawns, knights, bishops, rooks, queen, king), including castling, en passant, pawn promotion, and check/checkmate/stalemate detection.

3. Playable interface – The game displays an 8×8 board with piece images, file/rank labels, move highlights for the selected piece, and valid-move indicators (e.g. dots).

4. Time controls – Each side has a clock; time decreases on their turn, with optional increment per move. The game ends when one side runs out of time.

5. Game end and restart – The program shows a clear message for checkmate, stalemate, or time-out, and allows the user to restart (e.g. Play Again / New Game or key R/N).

Centre Name: Centre Number:

Design

✔ ***You may like to create a flow charts which will show broadly how your program will work. If so include your flow chart in this section.***

1. Show login window (Tkinter) → user enters username/password or registers.

2. If login OK → show settings window: choose side (White/Black), time control (e.g. 1+0, 3+2, 15+10).

3. Start Pygame window: load board and piece images, set initial game state and clocks.

4. Main loop: handle events (quit, mouse click, keys). On click: select square or make move if two squares selected; validate move using engine; update board and clocks; handle promotion pop-up if needed.

5. Each frame: decrement current player's time; redraw board, pieces, highlights, move history, clocks; if game over, show message and restart menu.

6. User can undo (e.g. U), restart (R/N), or quit.

Flow (broad behaviour):

✔ ***You must create pseudocode for a part of your program (minimum of 15 lines). If possible, try to create all of your program in pseudocode. Use the OCR guide in the specification to help you.***

```
SWITCH turn to other player

APPEND move to moveLog

UPDATE castle rights

UPDATE en passant possibility if pawn moved two squares

    SET promoted square to queen (or chosen piece later)

IF move is pawn promotion:

    MOVE rook to correct square

IF move is castle:

    UPDATE king location for that colour

IF piece moved is king:
```

Centre Name: Centre Number:

```
    SET board[move.endRow][move.endCol] to move.pieceMoved
        REMOVE captured pawn from board
    IF move is en passant:
    SET board[move.startRow][move.startCol] to empty
FUNCTION makeMove(move):


    RETURN legal
        CALL undoMove()
            ADD m to legal
        IF current player's king is NOT in check after move:
        CALL makeMove(m)
        save current enPassant state
            CONTINUE
        IF m would capture a king:
    FOR each move m in moves:
    legal = empty list
    CALL addCastleMoves(moves)
            CALL appropriate move function for piece type (pawn, rook, knight, etc.) with (r, c, moves)
                CONTINUE to next square
            IF piece is empty OR piece colour is not current player's turn:
            piece = board[r][c]
        FOR each column c from 0 to 7:
    FOR each row r from 0 to 7:
    moves = empty list
FUNCTION getValidMoves():
```

Centre Name: Centre Number:

Pseudocode (move validation and making a move):

Candidate Name:
Candidate Number:
Date:

Centre Name: Centre Number:

Test design

✔ *Think of tests that you can carry out to see if your system works*

✔ *Remember to try and use normal, boundary and erroneous tests.*

✔ *If you wish to, you may add more tests to the table.*

My tests:

Test	What am I testing?	What data will I use?	Normal/Boundary/Erroneous?	Expected Result
1	Login with valid user	Username: test, Password: test	Normal	Login succeeds and settings window appears
2	Login with wrong password	Username: test, Password: wrong	Erroneous	Error message "Invalid username or password"
3	Register new user	New username and password	Normal	Message "Registered. You can log in now." and can then log in
4	Making a legal move	Click e2 then e4 (White)	Normal	Pawn moves to e4; turn switches to Black
5	Move when time is zero	Let one player's clock reach 0:00	Boundary	Game shows "White/Black out of time - [other] wins" and restart option

Candidate Name:
Candidate Number:
Date:

Centre Name: Centre Number:

Development

✔ *Copy and paste your code into this section*

✔ *Remember to try and add comments to your code to make it more readable!*

My program code:

```
        return (self.startRow, self.startCol, self.endRow, self.endCol) == (other.startRow, other.startCol,
other.endRow, other.endCol)

            return False

        if not isinstance(other, Move):

    def __eq__(self, other):


        return self.colsToFiles[self.startCol] + self.rowsToRanks[self.startRow] +
self.colsToFiles[self.endCol] + self.rowsToRanks[self.endRow]

    def getChessNotation(self):


        self.enPassantPossibleBefore = None

        self.isPawnPromotion = (self.pieceMoved[1] == 'p' and (self.endRow == 0 or self.endRow == 7))

        self.isCastleMove = isCastle

            self.pieceCaptured = 'bp' if self.pieceMoved[0] == 'w' else 'wp'

        if self.isEnPassantMove:

        self.isEnPassantMove = isEnPassant

        self.pieceCaptured = board[self.endRow][self.endCol]

        self.pieceMoved = board[self.startRow][self.startCol]

        self.endRow, self.endCol = endSq[0], endSq[1]
```

Centre Name: Centre Number:

```
        self.startRow, self.startCol = startSq[0], startSq[1]

    def __init__(self, startSq, endSq, board, isEnPassant=False, isCastle=False):


    colsToFiles = {v: k for k, v in filesToCols.items()}

    filesToCols = {"a": 0, "b": 1, "c": 2, "d": 3, "e": 4, "f": 5, "g": 6, "h": 7}

    rowsToRanks = {v: k for k, v in ranksToRows.items()}

    ranksToRows = {"1": 7, "2": 6, "3": 5, "4": 4, "5": 3, "6": 2, "7": 1, "8": 0}

class Move():


        if move.pieceCaptured == 'bR' and move.endRow == 0 and move.endCol == 7:
self.castleRights['bks'] = False

        if move.pieceCaptured == 'bR' and move.endRow == 0 and move.endCol == 0:
self.castleRights['bqs'] = False

        if move.pieceCaptured == 'wR' and move.endRow == 7 and move.endCol == 7:
self.castleRights['wks'] = False

        if move.pieceCaptured == 'wR' and move.endRow == 7 and move.endCol == 0:
self.castleRights['wqs'] = False

            elif move.startRow == 0 and move.startCol == 7: self.castleRights['bks'] = False

            if move.startRow == 0 and move.startCol == 0: self.castleRights['bqs'] = False

        elif move.pieceMoved == 'bR':

            elif move.startRow == 7 and move.startCol == 7: self.castleRights['wks'] = False

            if move.startRow == 7 and move.startCol == 0: self.castleRights['wqs'] = False

        elif move.pieceMoved == 'wR':

            self.castleRights['bks'] = self.castleRights['bqs'] = False

        elif move.pieceMoved == 'bK':

            self.castleRights['wks'] = self.castleRights['wqs'] = False

        if move.pieceMoved == 'wK':
```

Centre Name: Centre Number:

```
    def updateCastleRights(self, move):



                    moves.append(Move((r,4),(r,2), self.board, isCastle=True))

                if not self.squareAttacked(r, 2, byWhite=True) and not self.squareAttacked(r, 3,
byWhite=True):

                if self.castleRights['bqs'] and self.board[r][1] == "--" and self.board[r][2] == "--" and
self.board[r][3] == "--":

                    moves.append(Move((r,4),(r,6), self.board, isCastle=True))

                if not self.squareAttacked(r, 5, byWhite=True) and not self.squareAttacked(r, 6,
byWhite=True):

                if self.castleRights['bks'] and self.board[r][5] == "--" and self.board[r][6] == "--":

            if not self.isInCheck(False):

            r = 0

        else:

                    moves.append(Move((r,4),(r,2), self.board, isCastle=True))

                if not self.squareAttacked(r, 2, byWhite=False) and not self.squareAttacked(r, 3,
byWhite=False):

                if self.castleRights['wqs'] and self.board[r][1] == "--" and self.board[r][2] == "--" and
self.board[r][3] == "--":

                    moves.append(Move((r,4),(r,6), self.board, isCastle=True))

                if not self.squareAttacked(r, 5, byWhite=False) and not self.squareAttacked(r, 6,
byWhite=False):

                if self.castleRights['wks'] and self.board[r][5] == "--" and self.board[r][6] == "--":

            if not self.isInCheck(True):

            r = 7

        if self.whiteToMove:

    def addCastleMoves(self, moves):
```

Centre Name: Centre Number:

```
        return False
            return True
        if self.squareInBounds(rr, cc) and self.board[rr][cc] == attackerColor + 'K':
        rr, cc = r + dr, c + dc
    for dr, dc in [(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1)]:
            cc += dc
            rr += dr
                break
                    return True
                if piece[0] == attackerColor and piece[1] in 'RQ':
            if piece != "--":
            piece = self.board[rr][cc]
        while self.squareInBounds(rr, cc):
        rr, cc = r + dr, c + dc
    for dr, dc in [(-1,0),(1,0),(0,-1),(0,1)]:
            cc += dc
            rr += dr
                break
                    return True
                if piece[0] == attackerColor and piece[1] in 'BQ':
            if piece != "--":
            piece = self.board[rr][cc]
        while self.squareInBounds(rr, cc):
        rr, cc = r + dr, c + dc
    for dr, dc in [(-1,-1),(-1,1),(1,-1),(1,1)]:
```

Centre Name: Centre Number:

```
            return True
        if self.squareInBounds(rr, cc) and self.board[rr][cc] == attackerColor + 'N':
        rr, cc = r + dr, c + dc
    for dr, dc in [(-2,-1),(-2,1),(-1,-2),(-1,2),(1,-2),(1,2),(2,-1),(2,1)]:
            return True
        if self.squareInBounds(rr, cc) and self.board[rr][cc] == attackerColor + 'p':
        rr, cc = r + dr, c + dc
    for dc in (-1, 1):
    dr = -1 if byWhite else 1
    attackerColor = 'w' if byWhite else 'b'
def squareAttacked(self, r, c, byWhite):


    return self.squareAttacked(king_r, king_c, byWhite=not forWhite)
    king_r, king_c = self.whiteKingLocation if forWhite else self.blackKingLocation
        forWhite = self.whiteToMove
    if forWhite is None:
def isInCheck(self, forWhite=None):


            moves.append(Move((r, c), (nr, nc), self.board))
        if target == "--" or target[0] != ownColor:
        target = self.board[nr][nc]
            continue
        if not self.squareInBounds(nr, nc):
        nr, nc = r + dr, c + dc
    for dr, dc in [(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1)]:
```

Centre Name: Centre Number:

```
        ownColor = self.board[r][c][0]
    def getKingMoves(self, r, c, moves):


                moves.append(Move((r, c), (nr, nc), self.board))
            if target == "--" or target[0] != ownColor:
            target = self.board[nr][nc]
                continue
            if not self.squareInBounds(nr, nc):
            nr, nc = r + dr, c + dc
        for dr, dc in [(-2,-1),(-2,1),(-1,-2),(-1,2),(1,-2),(1,2),(2,-1),(2,1)]:
        ownColor = self.board[r][c][0]
    def getKnightMoves(self, r, c, moves):


                nc += dc
                nr += dr
                    break
                        moves.append(Move((r, c), (nr, nc), self.board))
                    if target[0] != ownColor:
                else:
                    moves.append(Move((r, c), (nr, nc), self.board))
                if target == "--":
                target = self.board[nr][nc]
            while self.squareInBounds(nr, nc):
            nr, nc = r + dr, c + dc
        for dr, dc in directions:
```

Centre Name: Centre Number:

```
        ownColor = self.board[r][c][0]
    def _getSlidingMoves(self, r, c, moves, directions):


        self._getSlidingMoves(r, c, moves, [(-1,0),(1,0),(0,-1),(0,1),(-1,-1),(-1,1),(1,-1),(1,1)])
    def getQueenMoves(self, r, c, moves):
        self._getSlidingMoves(r, c, moves, [(-1,-1),(-1,1),(1,-1),(1,1)])
    def getBishopMoves(self, r, c, moves):
        self._getSlidingMoves(r, c, moves, [(-1,0),(1,0),(0,-1),(0,1)])
    def getRookMoves(self, r, c, moves):


                moves.append(Move((r, c), (ep_r, ep_c), self.board, isEnPassant=True))
            if (r + direction, c + 1) == (ep_r, ep_c) and self.board[r][c+1] != "--" and self.board[r][c+1][0] !=
piece[0]:
                moves.append(Move((r, c), (ep_r, ep_c), self.board, isEnPassant=True))
            if (r + direction, c - 1) == (ep_r, ep_c) and self.board[r][c-1] != "--" and self.board[r][c-1][0] !=
piece[0]:
            ep_r, ep_c = self.enPassantPossible
        if self.enPassantPossible:
                moves.append(Move((r, c), (nr, nc), self.board))
            if target != "--" and target[0] != piece[0]:
            target = self.board[nr][nc]
                continue
            if not self.squareInBounds(nr, nc):
            nr, nc = r + direction, c + dc
        for dc in (-1, 1):
                moves.append(Move((r, c), (r + 2*direction, c), self.board))
```

Centre Name: Centre Number:

```
            if r == startRow and self.board[r + 2*direction][c] == "--":
            moves.append(Move((r, c), (r + direction, c), self.board))
        if self.squareInBounds(r + direction, c) and self.board[r + direction][c] == "--":
        startRow = 6 if piece[0] == 'w' else 1
        direction = -1 if piece[0] == 'w' else 1
        piece = self.board[r][c]
    def getPawnMoves(self, r, c, moves):


        return 0 <= r < 8 and 0 <= c < 8
    def squareInBounds(self, r, c):


        return legal
            self.undoMove()
                legal.append(m)
            if not self.isInCheck(not self.whiteToMove):
            self.makeMove(m)
            m.enPassantPossibleBefore = self.enPassantPossible
                continue
            if m.pieceCaptured and (m.pieceCaptured == 'wK' or m.pieceCaptured == 'bK'):
        for m in moves:
        legal = []
        self.addCastleMoves(moves)
                self.moveFunctions[piece[1]](r, c, moves)
                    continue
                if piece == "--" or (piece[0] == 'w') != self.whiteToMove:
```

Centre Name: Centre Number:

```
                piece = self.board[r][c]
            for c in range(8):
        for r in range(8):
        moves = []
    def getValidMoves(self):


        self.whiteToMove = not self.whiteToMove
        self.enPassantPossible = move.enPassantPossibleBefore
        self.castleRights = self.castleRightsLog[-1].copy()
        self.castleRightsLog.pop()
                self.board[move.endRow][3] = "--"
                self.board[move.endRow][0] = self.board[move.endRow][3]
            else:
                self.board[move.endRow][5] = "--"
                self.board[move.endRow][7] = self.board[move.endRow][5]
            if move.endCol == 6:
        if move.isCastleMove:
            self.blackKingLocation = (move.startRow, move.startCol)
        elif moved == 'bK':
            self.whiteKingLocation = (move.startRow, move.startCol)
        if moved == 'wK':
            self.board[move.endRow][move.endCol] = move.pieceCaptured
        else:
            self.board[move.startRow][move.endCol] = move.pieceCaptured
            self.board[move.endRow][move.endCol] = "--"
```

Candidate Name:
Candidate Number:
Date:

```
        if move.isEnPassantMove:
            self.board[move.startRow][move.startCol] = moved
            moved = moved[0] + 'p'
        if move.isPawnPromotion:
        moved = move.pieceMoved
        self.board[move.startRow][move.startCol] = move.pieceMoved
        move = self.moveLog.pop()
            return
        if len(self.moveLog) == 0:
    def undoMove(self):


        self.whiteToMove = not self.whiteToMove
        self.moveLog.append(move)
        self.castleRightsLog.append(self.castleRights.copy())
        self.updateCastleRights(move)
            self.enPassantPossible = None
        else:
            self.enPassantPossible = ((move.startRow + move.endRow)//2, move.startCol)
        if move.pieceMoved[1] == 'p' and abs(move.startRow - move.endRow) == 2:
            self.board[move.endRow][move.endCol] = move.pieceMoved[0] + 'Q'
        if move.isPawnPromotion:
                self.board[move.endRow][0] = "--"
                self.board[move.endRow][3] = self.board[move.endRow][0]
            else:
                self.board[move.endRow][7] = "--"
```

Centre Name: Centre Number:

```
            self.board[move.endRow][5] = self.board[move.endRow][7]
        if move.endCol == 6:
    if move.isCastleMove:
        self.blackKingLocation = (move.endRow, move.endCol)
    elif move.pieceMoved == 'bK':
        self.whiteKingLocation = (move.endRow, move.endCol)
    if move.pieceMoved == 'wK':
    self.board[move.endRow][move.endCol] = move.pieceMoved
        self.board[move.startRow][move.endCol] = "--"
    if move.isEnPassantMove:
    self.board[move.startRow][move.startCol] = "--"
def makeMove(self, move):


    self.castleRightsLog = [self.castleRights.copy()]
    self.castleRights = {'wks': True, 'wqs': True, 'bks': True, 'bqs': True}
    self.enPassantPossible = None
    self.blackKingLocation = (0, 4)
    self.whiteKingLocation = (7, 4)
    }
        'B': self.getBishopMoves, 'Q': self.getQueenMoves, 'K': self.getKingMoves
        'p': self.getPawnMoves, 'R': self.getRookMoves, 'N': self.getKnightMoves,
    self.moveFunctions = {
    self.moveLog = []
    self.whiteToMove = True
        ["wR", "wN", "wB", "wQ", "wK", "wB", "wN", "wR"]]
```

Candidate Name:
Candidate Number:
Date:

Centre Name: Centre Number:

```
            ["wp", "wp", "wp", "wp", "wp", "wp", "wp", "wp"],
            ["--", "--", "--", "--", "--", "--", "--", "--"],
            ["--", "--", "--", "--", "--", "--", "--", "--"],
            ["--", "--", "--", "--", "--", "--", "--", "--"],
            ["--", "--", "--", "--", "--", "--", "--", "--"],
            ["bp", "bp", "bp", "bp", "bp", "bp", "bp", "bp"],
            ["bR", "bN", "bB", "bQ", "bK", "bB", "bN", "bR"],
        self.board = [
    def __init__(self):
class GameState():


# Chess game state and move logic (GameState, Move, piece movement, check, castling, en passant)


chessEngine.py - Game state, move generation, check/checkmate, castling, en passant.




    main()
if __name__ == "__main__":


    return (result["base"], result["inc"], result["side"])
        return None
    if not result["done"]:
    root.mainloop()
    tk.Button(root, text='Cancel', command=root.destroy).pack(fill='x')
    tk.Button(root, text='Start', command=start).pack(fill='x', pady=(8,0))
```

Centre Name: Centre Number:

```
        root.destroy()
        result["side"] = side_var.get()
        result["inc"] = inc
        result["base"] = base
        result["done"] = True
                break
                base, inc = b, i
            if r.cget('value') == key:
        for r, b, i in radios:
        base, inc = 180, 0
        key = time_var.get()
    def start():
        radios.append((r, base, inc))
        r.pack(anchor='w')
        r = tk.Radiobutton(root, text=label, variable=time_var, value=key)
    for label, key, base, inc in options:
    radios = []
    ]
        ('Rapid 30+0', 'rapid30+0', 1800, 0), ('Classical 60+0', 'class60+0', 3600, 0),
        ('Blitz 5+0', 'blitz5+0', 300, 0), ('Rapid 15+10', 'rapid15+10', 900, 10),
        ('Blitz 3+0', 'blitz3+0', 180, 0), ('Blitz 3+2', 'blitz3+2', 180, 2),
        ('Bullet 1+0', 'bullet1+0', 60, 0), ('Bullet 2+1', 'bullet2+1', 120, 1),
    options = [
    time_var = tk.StringVar(value='blitz3+0')
    tk.Label(root, text="Time Control").pack(anchor='w', pady=(8,0))
```

Centre Name: Centre Number:

```
    tk.Radiobutton(root, text='Black', variable=side_var, value='black').pack(anchor='w')
    tk.Radiobutton(root, text='White', variable=side_var, value='white').pack(anchor='w')
    side_var = tk.StringVar(value='white')
    tk.Label(root, text="Choose Side").pack(anchor='w')
    root.title(f"{CLIENT_NAME} - Game Settings")
    root = tk.Tk()
    result = {"done": False, "base": 180, "inc": 0, "side": 'white'}
def settings_window():


    return result["user"]
    root.mainloop()
    tk.Button(root, text="Register", command=do_register).grid(row=2, column=1, sticky='we')
    tk.Button(root, text="Login", command=do_login).grid(row=2, column=0, sticky='we')
        messagebox.showinfo("OK", "Registered. You can log in now.")
            f.write(f"{name}:{pw}\n")
        with open(path, 'a', encoding='utf-8') as f:
        users[name] = pw
            return
            messagebox.showerror("Error", "User exists")
        if name in users:
            return
            messagebox.showerror("Error", "Please enter username and password")
        if not name or not pw:
        name, pw = u.get().strip(), pwd.get().strip()
    def do_register():
```

Centre Name: Centre Number:

```
            messagebox.showerror("Error", "Invalid username or password")
        else:
            root.destroy()
            result["user"] = name
        if name in users and users[name] == pw:
        name, pw = u.get().strip(), pwd.get().strip()
    def do_login():
    pwd.grid(row=1, column=1)
    u.grid(row=0, column=1)
    pwd = tk.Entry(root, show='*')
    u = tk.Entry(root)
    tk.Label(root, text="Password").grid(row=1, column=0)
    tk.Label(root, text="Username").grid(row=0, column=0)
    root.title(f"{CLIENT_NAME} - Login")
    root = tk.Tk()
    result = {"user": None}
        pass
    except FileNotFoundError:
                users[parts[0].strip()] = parts[1].strip()
            if len(parts) == 2:
            parts = line.split(':', 1) if ':' in line else line.split(',', 1)
                continue
            if not line or line.startswith(""):
            line = line.strip()
        for line in f:
```

Centre Name: Centre Number:

```
        with open(path, 'r', encoding='utf-8') as f:
    try:
    users = {}
    path = os.path.join(base_dir, "users.csv")
    base_dir = os.path.dirname(os.path.abspath(__file__))
def login_window():


    screen.blit(inst_text, inst_text.get_rect(center=(center_x, center_y + 80)))
    inst_text = font.render("Press R or N for Restart", True, p.Color('black'))
    screen.blit(new_text, new_text.get_rect(center=new_game_rect.center))
    new_text = font.render("New Game", True, p.Color('black'))
    p.draw.rect(screen, p.Color('black'), new_game_rect, 2)
    p.draw.rect(screen, p.Color('lightblue'), new_game_rect)
    new_game_rect = p.Rect(center_x - button_width//2, center_y + 10, button_width, button_height)
    screen.blit(play_text, play_text.get_rect(center=play_again_rect.center))
    play_text = font.render("Play Again", True, p.Color('black'))
    p.draw.rect(screen, p.Color('black'), play_again_rect, 2)
    p.draw.rect(screen, p.Color('lightgreen'), play_again_rect)
    play_again_rect = p.Rect(center_x - button_width//2, center_y - 60, button_width, button_height)
    button_width, button_height = 200, 50
    center_x, center_y = Width//2, Height//2
    screen.blit(s, (0, 0))
    s.fill(p.Color('white'))
    s.set_alpha(200)
    s = p.Surface((Width, Height))
```

Centre Name: Centre Number:

```
def drawRestartMenu(screen, font):


    screen.blit(inst_text, inst_rect)
    inst_rect = inst_text.get_rect(center=(center_x, center_y - piece_size))
    inst_text = font.render("Choose promotion piece:", True, p.Color('black'))
        screen.blit(label, label_rect)
        label_rect = label.get_rect(center=(x + piece_size//2, y + piece_size + 20))
        label = font.render(piece, True, p.Color('black'))
            screen.blit(piece_img, (x+5, y+5))
            piece_img = p.transform.scale(IMAGES[piece_name], (piece_size-10, piece_size-10))
        if piece_name in IMAGES:
        piece_name = piece_color + piece
        piece_color = 'w' if gs.whiteToMove else 'b'
        p.draw.rect(screen, p.Color('black'), p.Rect(x, y, piece_size, piece_size), 2)
        p.draw.rect(screen, p.Color('lightgray'), p.Rect(x, y, piece_size, piece_size))
        y = start_y
        x = start_x + i*piece_size
    for i, piece in enumerate(pieces):
    pieces = ['Q', 'R', 'B', 'N']
    start_y = center_y - piece_size//2
    start_x = center_x - 2*piece_size
    piece_size = SQ_SIZE
    center_x, center_y = Width//2, Height//2
    screen.blit(s, (0, 0))
    s.fill(p.Color('white'))
```

Centre Name: Centre Number:

```
    s.set_alpha(200)
    s = p.Surface((Width, Height))
def drawPromotionMenu(screen, font, whiteBottom, gs):


        y += 20
        screen.blit(text, (x, y))
        text = font.render(f"{i+1}. {move}", True, p.Color('black'))
    for i, move in enumerate(moves[-8:]):
    y += 22
    screen.blit(font.render("Moves:", True, p.Color('black')), (x, y))
    x, y = 10, 50
def drawMoveHistory(screen, font, moves):


    screen.blit(t, rect)
    rect = t.get_rect(center=(Width//2, Height//2))
    t = font.render(text, True, p.Color('red'))
    screen.blit(s, (0, 0))
    s.fill(p.Color('white'))
    s.set_alpha(140)
    s = p.Surface((Width, Height))
def drawGameOver(screen, font, text):


    screen.blit(white_surf, (10, Height - 10 - white_surf.get_height()))
    screen.blit(black_surf, (10, 10))
    white_surf = font.render(f"White: {fmt(white_ms)}", True, p.Color('black'))
```

Centre Name: Centre Number:

```
    black_surf = font.render(f"Black: {fmt(black_ms)}", True, p.Color('black'))
        return f"{m:02d}:{s:02d}"
        s = total%60
        m = total//60
        total = max(0, ms//1000)
    def fmt(ms):
def drawClocks(screen, font, white_ms, black_ms):


        screen.blit(text, (4, r*SQ_SIZE + 4))
        text = small.render(rankChar, True, p.Color('black'))
        rankChar = ranks[7-r] if whiteBottom else ranks[r]
    for r in range(8):
        screen.blit(text, (c*SQ_SIZE + 4, Height - text.get_height() - 4))
        text = small.render(fileChar, True, p.Color('black'))
        fileChar = files[c] if whiteBottom else files[7-c]
    for c in range(8):
    ranks = ['1','2','3','4','5','6','7','8']
    files = ['a','b','c','d','e','f','g','h']
    small = p.font.SysFont(None, 24)
def drawLabels(screen, whiteBottom):


            p.draw.circle(screen, p.Color('red'), center, SQ_SIZE//8)
            center = (dc2*SQ_SIZE + SQ_SIZE//2, dr2*SQ_SIZE + SQ_SIZE//2)
            dc2 = m.endCol if whiteBottom else 7 - m.endCol
            dr2 = m.endRow if whiteBottom else 7 - m.endRow
```

Centre Name: Centre Number:

```
        if (m.startRow, m.startCol) == (r, c):
    for m in validMoves:
    screen.blit(s, (dc*SQ_SIZE, dr*SQ_SIZE))
    dc = c if whiteBottom else 7 - c
    dr = r if whiteBottom else 7 - r
    s.fill(p.Color('yellow'))
    s.set_alpha(100)
    s = p.Surface((SQ_SIZE, SQ_SIZE))
    r, c = sqSelected
        return
    if sqSelected == ():
def drawHighlights(screen, sqSelected, validMoves, whiteBottom):


                screen.blit(IMAGES[piece], (x, y))
                y = dr*SQ_SIZE + (SQ_SIZE - PIECE_SIZE)//2
                x = dc*SQ_SIZE + (SQ_SIZE - PIECE_SIZE)//2
                dc = c if whiteBottom else 7 - c
                dr = r if whiteBottom else 7 - r
            if piece != "--":
            piece = board[r][c]
        for c in range(Dimension):
    for r in range(Dimension):
def drawPieces(screen, board, whiteBottom):


            p.draw.rect(screen, color, p.Rect(dc*SQ_SIZE, dr*SQ_SIZE, SQ_SIZE, SQ_SIZE))
```

Centre Name: Centre Number:

```
            dc = c if whiteBottom else 7 - c
            dr = r if whiteBottom else 7 - r
            color = colors[(r+c) % 2]
        for c in range(Dimension):
    for r in range(Dimension):
    colors = [p.Color("white"), p.Color("gray")]
def drawBoard(screen, whiteBottom):


    drawLabels(screen, whiteBottom)
    drawPieces(screen, gs.board, whiteBottom)
    drawHighlights(screen, sqSelected, validMoves, whiteBottom)
    drawBoard(screen, whiteBottom)
def drawGameState(screen, gs, sqSelected, validMoves, whiteBottom):


        p.display.flip()
            drawRestartMenu(screen, font)
        if show_restart_menu:
            drawGameOver(screen, font, game_over_text)
        if game_over_text:
            drawPromotionMenu(screen, font, orientationWhiteBottom, gs)
        if promotion_pending:
        drawClocks(screen, font, white_time_ms, black_time_ms)
        drawMoveHistory(screen, small_font, move_history)
        drawGameState(screen, gs, sqSelected, validMoves, orientationWhiteBottom)
                show_restart_menu = True
```

```
                game_over_text = "Black out of time - White wins"
                black_time_ms = 0
            if black_time_ms <= 0:
            black_time_ms -= dt
        else:
                show_restart_menu = True
                game_over_text = "White out of time - Black wins"
                white_time_ms = 0
            if white_time_ms <= 0:
            white_time_ms -= dt
        if gs.whiteToMove:
    if not game_over_text:
                playerClicks = []
                sqSelected = ()
                show_restart_menu = False
                game_over_text = ""
                promotion_pending = None
                move_history = []
                black_time_ms = int(base_seconds * 1000)
                white_time_ms = int(base_seconds * 1000)
                validMoves = gs.getValidMoves()
                gs = GameState()
            elif e.key == p.K_r or e.key == p.K_n:
                    show_restart_menu = False
                    game_over_text = ""
```

Centre Name: Centre Number:

```
                promotion_pending = None
                    move_history.pop()
                if move_history:
                validMoves = gs.getValidMoves()
                gs.undoMove()
            if len(gs.moveLog) > 0:
        if e.key == p.K_u:
    elif e.type == p.KEYDOWN:
                    show_restart_menu = True
                        game_over_text = "Stalemate"
                    else:
                        game_over_text = "Checkmate - " + ("White" if not gs.whiteToMove else "Black")
+ " wins"
                    if gs.isInCheck(gs.whiteToMove):
                if len(validMoves) == 0:
                validMoves = gs.getValidMoves()
                promotion_pending = None
                gs.board[promotion_pending.endRow][promotion_pending.endCol] =
promotion_pending.pieceMoved[0] + chosen_piece
                chosen_piece = piece_choices[choice_idx]
                piece_choices = ['Q', 'R', 'B', 'N']
            if 0 <= choice_idx < 4:
            choice_idx = click_x // piece_size
            click_x = location[0] - start_x
            start_y <= location[1] <= start_y + piece_size):
        if (start_x <= location[0] <= start_x + 4*piece_size and
```

Centre Name: Centre Number:

```
            start_y = center_y - piece_size//2
            start_x = center_x - 2*piece_size
            piece_size = SQ_SIZE
            center_x, center_y = Width//2, Height//2
            location = p.mouse.get_pos()
        elif e.type == p.MOUSEBUTTONDOWN and promotion_pending:
            playerClicks = []
            sqSelected = ()
                    show_restart_menu = True
                    game_over_text = "Stalemate"
                    else:
                    game_over_text = "Checkmate - " + ("White" if not gs.whiteToMove else
"Black") + " wins"
                    if gs.isInCheck(gs.whiteToMove):
                if len(validMoves) == 0:
                else:
                    promotion_pending = move
                if move.isPawnPromotion:
                move_history.append(move.getChessNotation())
                validMoves = gs.getValidMoves()
                    black_time_ms += increment_ms
                else:
                    white_time_ms += increment_ms
                if moved_white:
                gs.makeMove(move)
                moved_white = gs.whiteToMove
```

```
                        break
                        move = vm
                    if vm == move:
                for vm in validMoves:
            if move in validMoves:
            move = Move(playerClicks[0], playerClicks[1], gs.board)
        if len(playerClicks) == 2:
            playerClicks.append(sqSelected)
            sqSelected = (row, col)
        else:
            playerClicks = []
            sqSelected = ()
        if sqSelected == (row, col):
            row = 7 - row
            col = 7 - col
        if not orientationWhiteBottom:
        row = location[1]//SQ_SIZE
        col = location[0]//SQ_SIZE
        location = p.mouse.get_pos()
    elif e.type == p.MOUSEBUTTONDOWN and not game_over_text and not promotion_pending:
        running = False
    if e.type == p.QUIT:
for e in p.event.get():
dt = clock.tick(MAX_FPS)
while running:
```

Centre Name: Centre Number:

```
    show_restart_menu = False
    promotion_pending = None
    move_history = []
    game_over_text = ""
    increment_ms = int(increment_seconds * 1000)
    black_time_ms = int(base_seconds * 1000)
    white_time_ms = int(base_seconds * 1000)
    validMoves = gs.getValidMoves()
    playerClicks = []
    sqSelected = ()
    running = True
    loadImages()
    gs = GameState()
    small_font = p.font.SysFont(None, 24)
    font = p.font.SysFont(None, 42)
    screen.fill(p.Color("white"))
    clock = p.time.Clock()
    p.display.set_caption(CLIENT_NAME)
    screen = p.display.set_mode((Width, Height))
    PIECE_SIZE = max(60, int(SQ_SIZE * (100/135)))
    SQ_SIZE = Height // Dimension
    Width = Height = board_size
    global Width, Height, SQ_SIZE, PIECE_SIZE
    board_size = min(1080, max_fit)
    max_fit = max(400, min(info.current_w, info.current_h) - 80)
```

Centre Name: Centre Number:

```
    info = p.display.Info()
    p.init()
    orientationWhiteBottom = (player_side == 'white')
    base_seconds, increment_seconds, player_side = settings
        return
    if settings is None:
    settings = settings_window()
        return
    if not user:
    user = login_window()
def main():


        IMAGES[piece] = p.transform.scale(p.image.load(path), (PIECE_SIZE, PIECE_SIZE))
        path = os.path.join(img_dir, piece + ".png")
    for piece in pieces:
    img_dir = os.path.join(base_dir, "images")
    base_dir = os.path.dirname(os.path.abspath(__file__))
    pieces = ["wp", "wR", "wN", "wB", "wK", "wQ", "bp", "bR", "bN", "bB", "bK", "bQ"]
def loadImages():


PIECE_SIZE = 100
IMAGES = {}
MAX_FPS = 15
SQ_SIZE = Height // Dimension
Dimension = 8
```

Centre Name: Centre Number:

```
Width = Height = 1080

CLIENT_NAME = "Foot Master"


from chessEngine import GameState, Move

import os

from tkinter import messagebox

import tkinter as tk

import pygame as p
```

chessMain.py - Main game: login, settings, Pygame board, move input, clocks, promotion, game over.

Centre Name: Centre Number:

Testing

✔ *Show you have completed the tests you thought of*

✔ *Identify if you needed to make changes to your program*

✔ *Include the screenshots of the tests*

My tests:

Test	What am I testing?	Expected result	Pass/Fail	Do I need to change my program? If so, how?
1	Login with valid user	Login succeeds, settings appear	Pass	No change
2	Login with wrong password	Error message shown	Pass	No change
3	Register new user	Registration and login work	Pass	No change
4	Legal move (e2–e4)	Pawn moves, turn changes	Pass	No change
5	Clock reaches zero	Time-out message and	Pass	No change

Candidate Name:
Candidate Number:
Date:

Centre Name: Centre Number:

		winner shown		

My test screenshots:

Figure: Login screen

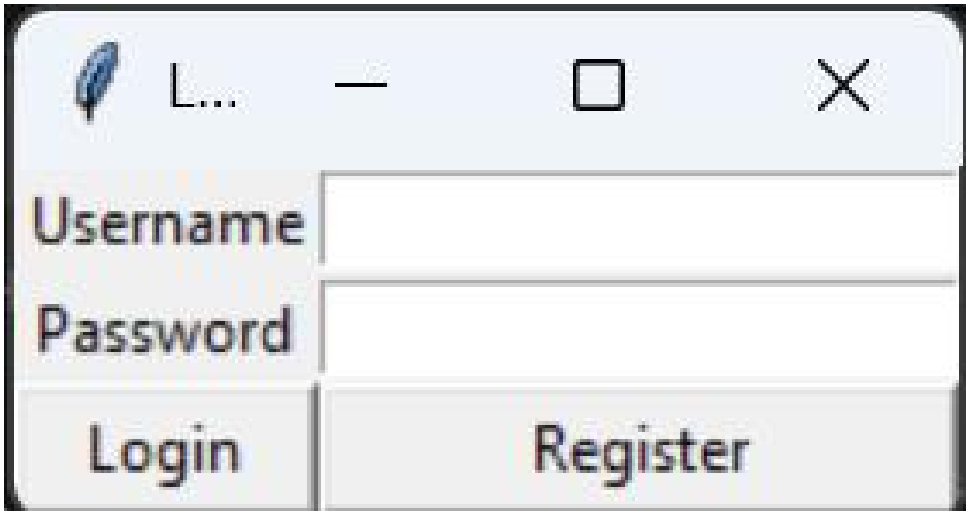

Figure: Board at start

Candidate Name:
Candidate Number:
Date:

Centre Name: Centre Number:

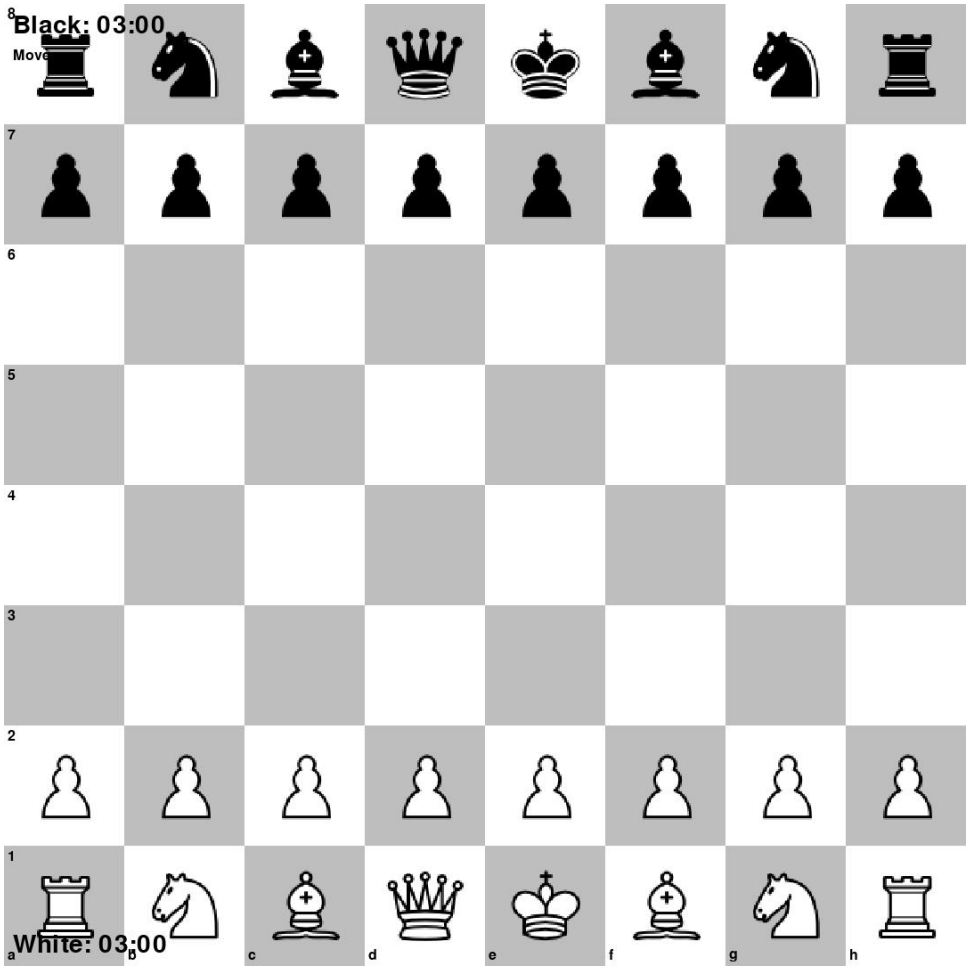

Figure: Valid moves highlighted

Candidate Name:
Candidate Number:
Date:

Centre Name: Centre Number:

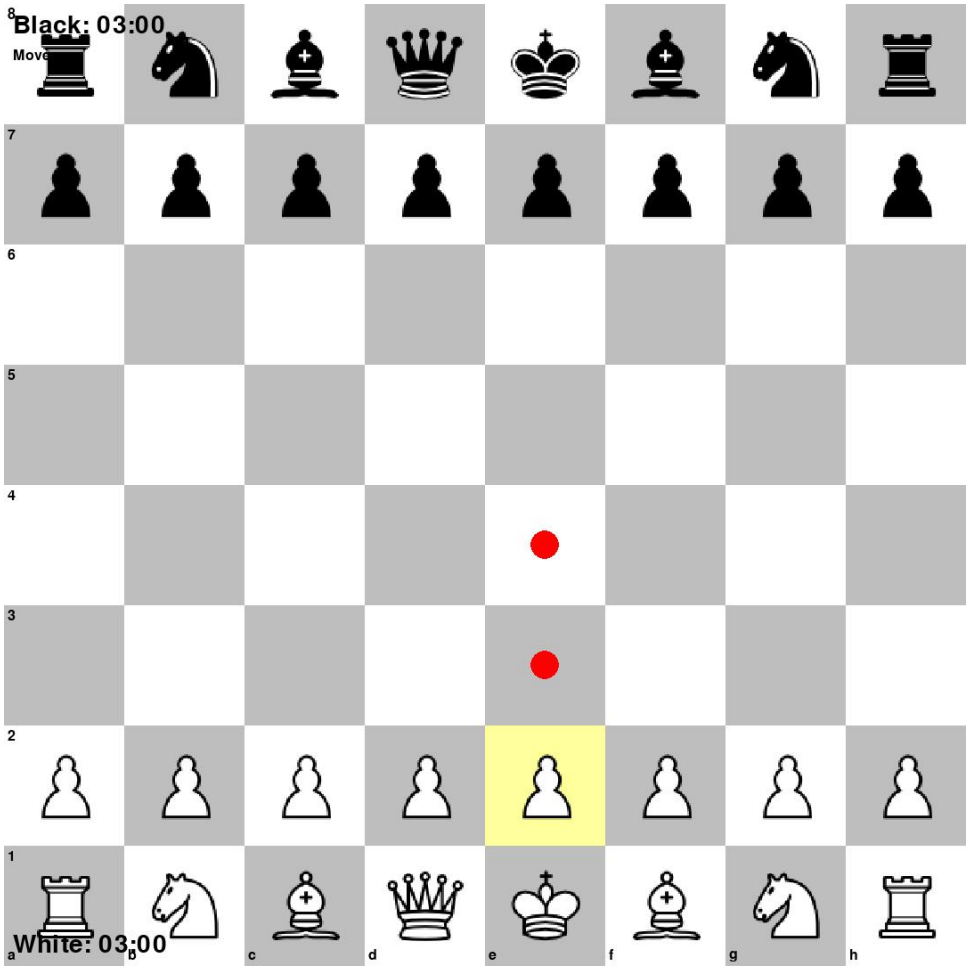

Figure: Clocks and move history

Candidate Name:
Candidate Number:
Date:

Centre Name: Centre Number:

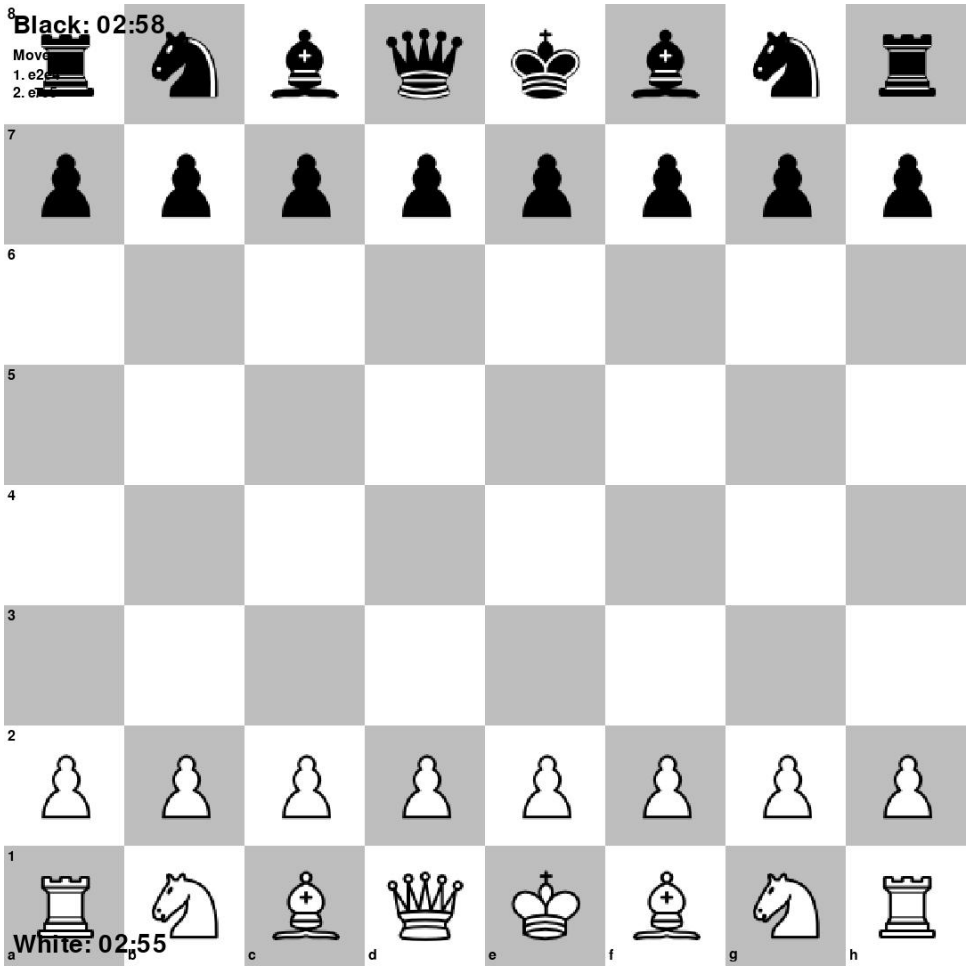

Figure: Checkmate

Candidate Name:
Candidate Number:
Date:

Centre Name: Centre Number:

Figure: Promotion menu

Candidate Name:
Candidate Number:
Date:

Centre Name: Centre Number:

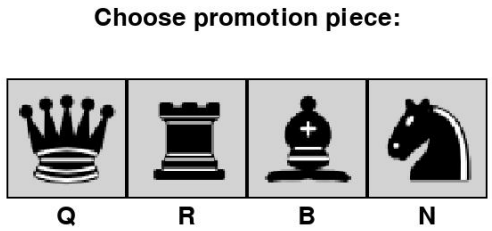

Figure: Restart menu

Candidate Name:
Candidate Number:
Date:

Centre Name: Centre Number:

Candidate Name:
Candidate Number:
Date:

Centre Name: Centre Number:

Evaluation

✔ Evaluate how successful your program was. You should like your evaluation to your testing results.

✔ You should reflect on any new skills you have developed

This section should be approximately 200-500 words.

How successful was my program?

The program meets the success criteria: users can log in and register, choose side and time control, and play a full game of chess on an 8×8 board with correct rules. All piece types move correctly, including castling and en passant, and check, checkmate and stalemate are detected. The interface shows move highlights and the last few moves, and both players have clocks that count down with optional increment. When the game ends (checkmate, stalemate or time-out), a clear message is shown and the user can restart. Testing showed that valid and invalid logins, registration, normal moves, and time-out behave as expected. One limitation is that the game is two-player on one machine (no network or AI opponent). Another is that promotion defaults to queen in the engine, but the GUI allows choosing queen, rook, bishop or knight. Overall, the program is successful for its intended purpose as a local two-player chess game with timing and standard rules.

What new skills have I developed?

I developed skills in structuring a larger program into two modules (engine vs. GUI). I used Pygame for the game loop, drawing, and mouse input, and Tkinter for dialogs, which required combining two libraries in one application. I implemented standard chess rules (move generation, check detection, castling, en passant, promotion) and data structures such as the board representation and move log. I practised event-driven design (clicks, key presses, timer updates) and simple file I/O for user credentials. I also improved code readability by adding comments and keeping the engine separate from the display logic, which made debugging and testing easier.