**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).

**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:

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():

Pseudocode (move validation and making a move):

**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 |

**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]

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':

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):

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)]:

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)]:

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:

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))

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:

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] = "--"

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] = "--"

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"]]

["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))

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))

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():

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:

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))

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'))

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'))

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

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))

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 = ""

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

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:

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)

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

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.

**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 winner shown | Pass | No change |

**My test screenshots:**

Figure: Login screen

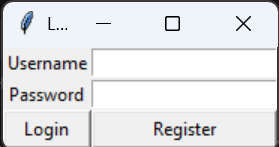


Figure: Board at start

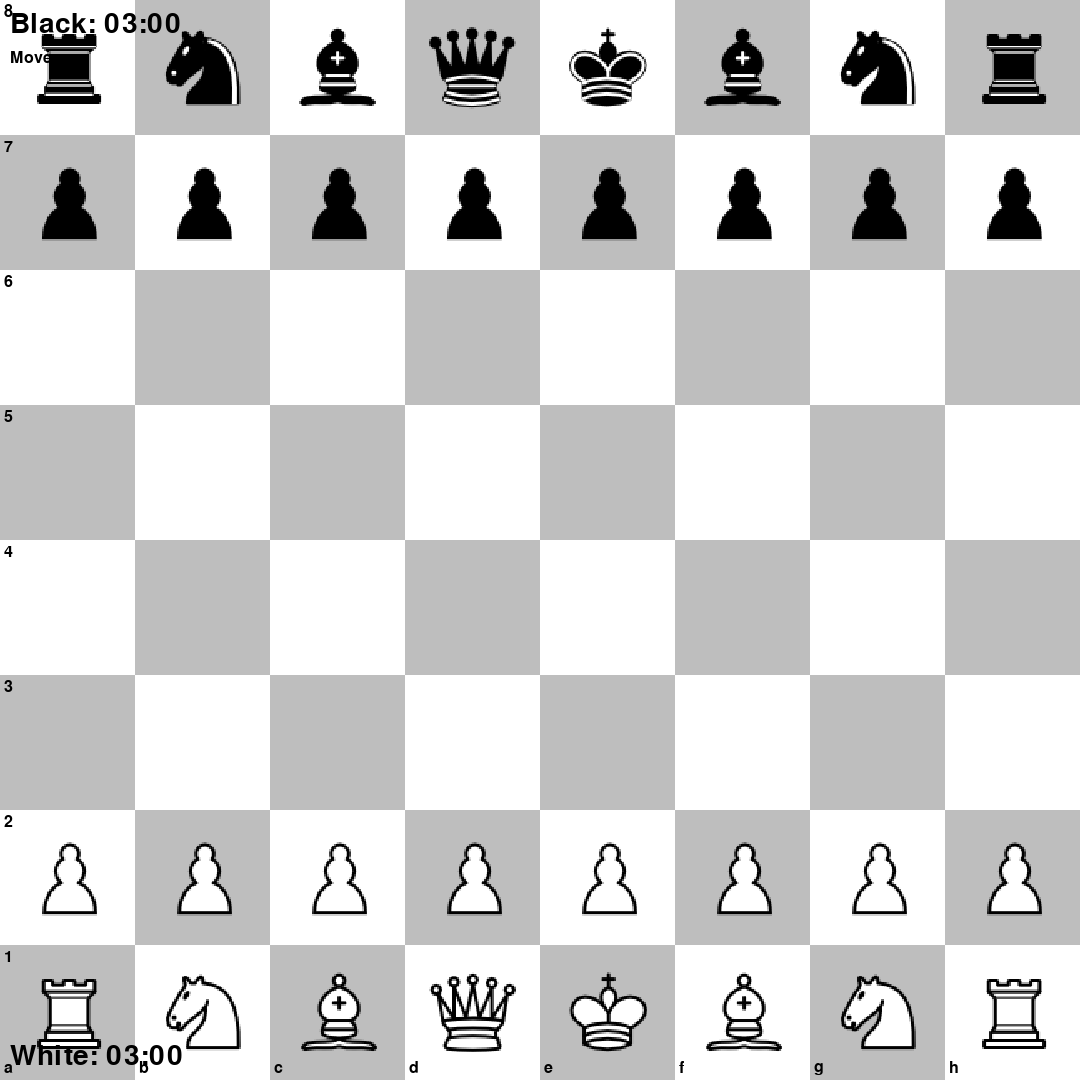


Figure: Valid moves highlighted

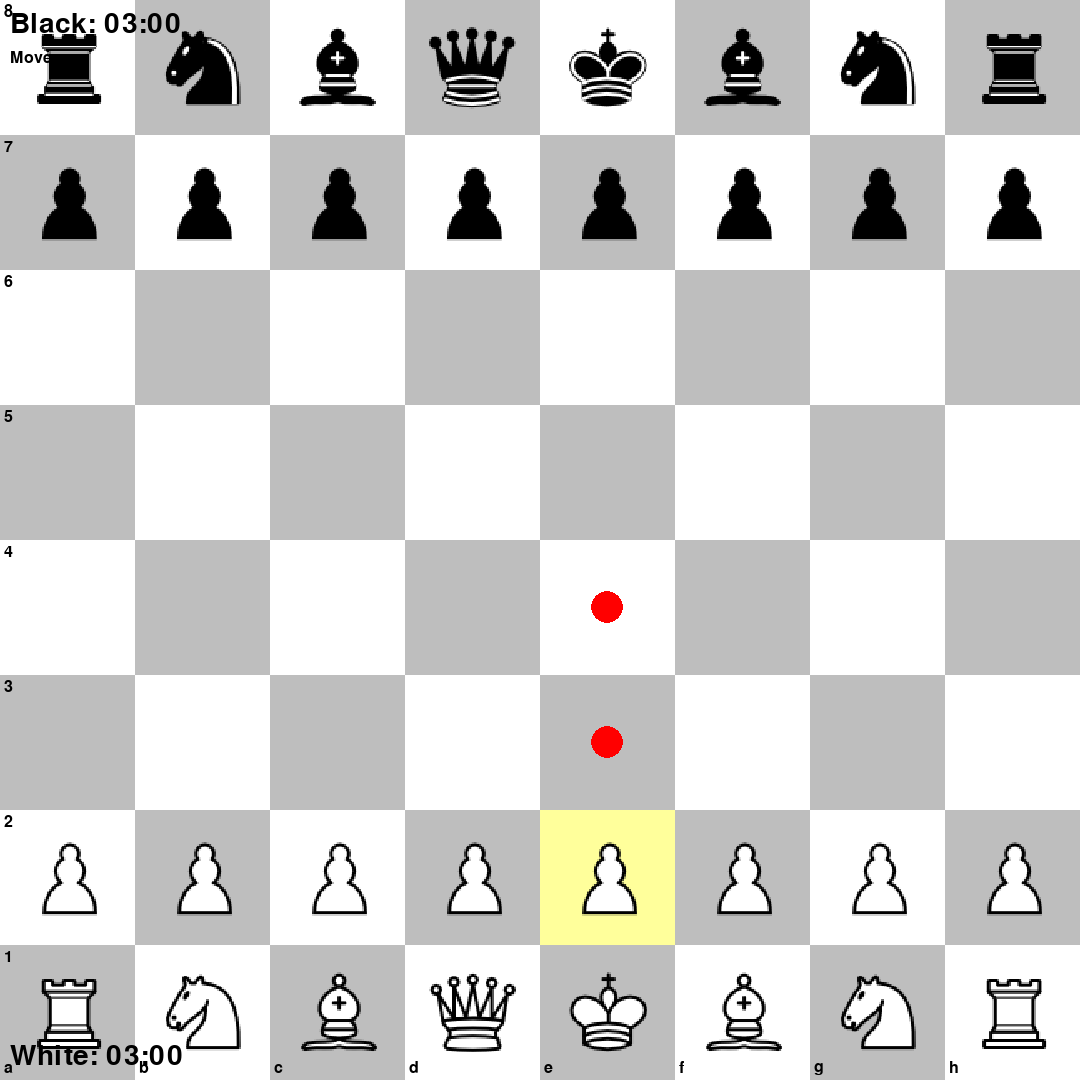


Figure: Clocks and move history

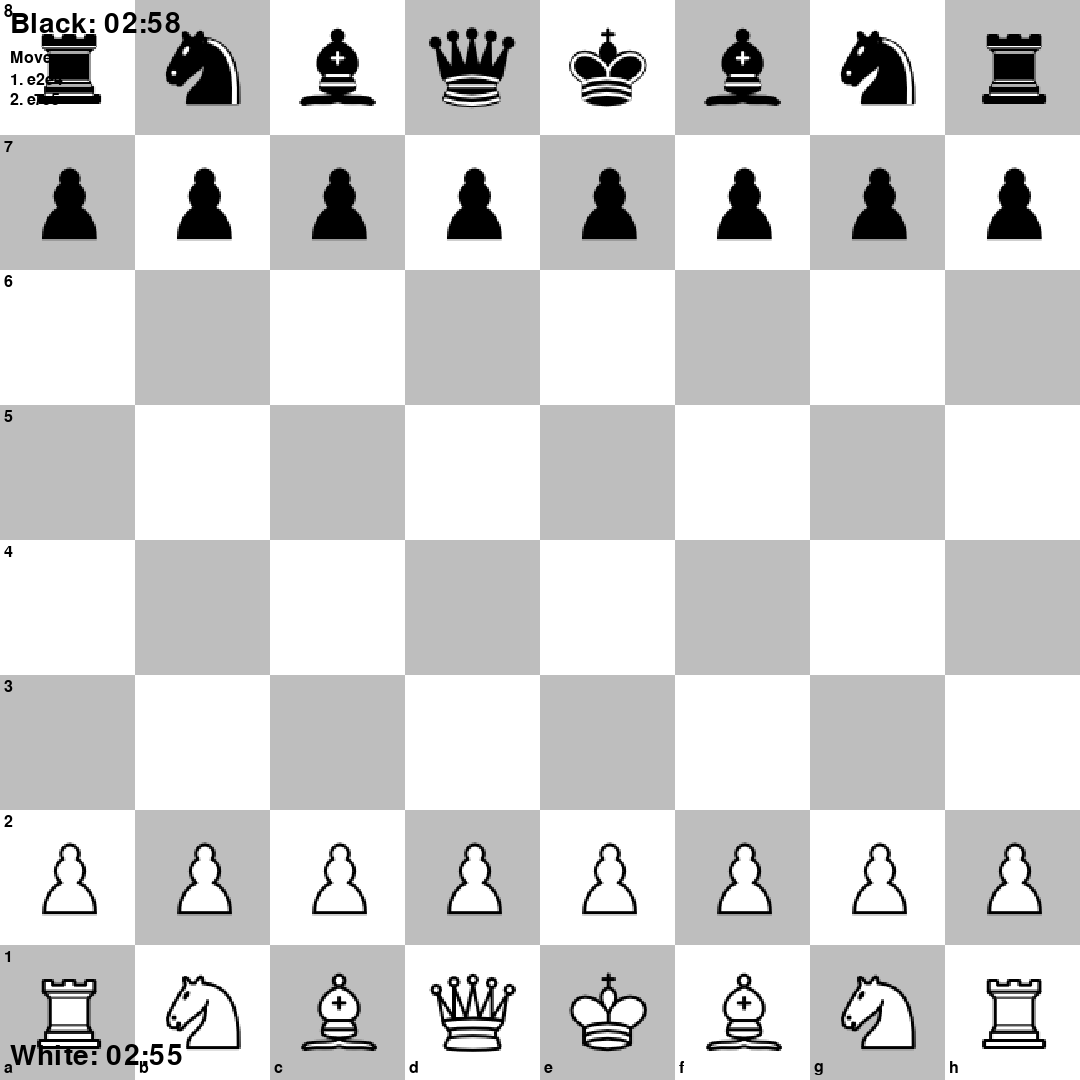


Figure: Checkmate



Figure: Promotion menu

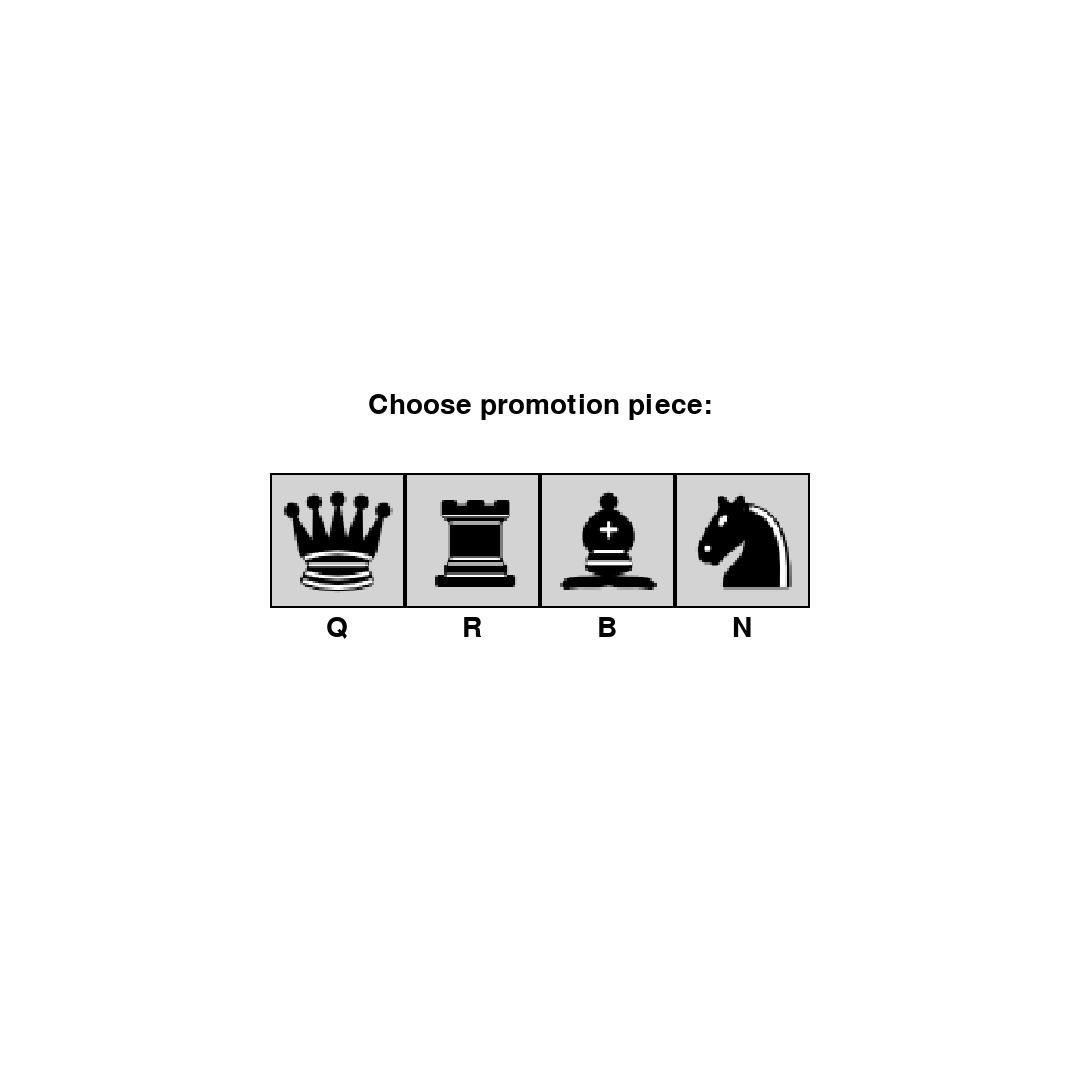


Figure: Restart menu



**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.