

Chess Game Simulation

Object-Oriented Analysis and Design

The goal of this system is to simulate a classical chess game between two players.

The application is responsible for creating and managing a chessboard, placing pieces in their initial positions, validating moves according to the rules of chess, tracking the current game state, and determining when the game ends (checkmate, stalemate, resignation). The system is designed with object-oriented principles in mind so that the game logic is easy to extend and maintain (for example, adding AI or a graphical interface in the future).

1. System Requirements

- Functional Requirements

1. The system must allow starting a new chess game
2. When a new game starts, all chess pieces must be placed in their standard initial positions
3. The system must support a game between two players
4. The system must display the current state of the chessboard
5. A player must be able to select a piece and make a move

6. The system must ensure that only the player whose turn it is can make a move
7. The system must validate moves according to the movement rules of each piece
8. The system must prevent illegal moves that leave the player's king in check
9. The system must support capturing opponent pieces
10. The system must detect when a king is in check
11. The system must detect checkmate and stalemate situations
12. The system must allow a player to resign, which immediately ends the game
13. The system must keep a history of all moves made during the game
14. The system must support pawn promotion when a pawn reaches the last rank
15. The system must support castling according to standard chess rules

- **Non-Functional Requirements**

1. The system should correctly follow official chess rules
2. Move validation should be performed quickly so that the game feels responsive

3. The code should be easy to understand and maintain
4. The system should be designed in a way that allows adding new features (such as AI or a timer) later
5. Game state and move history should remain consistent even if an invalid move is attempted
6. The system should be testable using unit tests for move validation and game logic

2. Use cases

Use Case 1: Start new game

Actor: Player

Description: A player starts a new chess game. The system creates a new board, places all pieces in their initial positions, sets the turn to White, and marks the game as active.

Use Case 2: Make a move

Actor: Current Player

Precondition: The game is in progress

Main flow:

1. The player selects a piece on the board
2. The player selects a target square
3. The system checks whether the move is legal
4. If the move is valid, the system applies the move to the board
5. If an opponent's piece is on the target square, it is captured
6. The move is added to the move history

7. The system checks for check, checkmate, or stalemate
8. The turn switches to the other player

Alternative flow: If the move is illegal, the system rejects the move and the board remains unchanged.

Use Case 3: Resign game

Actor: Player

Description: A player resigns the game. The system ends the game and declares the opponent as the winner.

Use Case 4: Pawn promotion

Actor: Player

Precondition: A pawn reaches the last rank

Description: The system asks the player to choose a piece (Queen, Rook, Bishop, or Knight). The pawn is replaced by the selected piece, and the game continues.

Use Case 5: Detect game end

Actor: System

Description: After each move, the system checks whether the game has ended due to checkmate or stalemate and updates the game status accordingly.

3. Objects, Classes and Relationships

The system is built around several core objects:

- **ChessGame** – controls the overall game flow, current turn, and game status.
- **Board** – represents the chessboard and stores the positions of all pieces
- **Square** – represents a single square on the board using file and rank
- **Piece** – an abstract class representing a chess piece
- **King, Queen, Rook, Bishop, Knight, Pawn** – concrete implementations of chess pieces
- **Player** – represents a player and stores their color
- **Move** – represents a single move, including source and destination squares, captured pieces, and special move information
- **MoveValidator** – contains the logic for validating moves and checking for check conditions

Relationships:

- **ChessGame** owns a **Board** and a list of **Move** objects
- **Board** contains **chess pieces** positioned on squares
- Each **Move** is associated with a **piece** and two **squares** (from and to).

- **Piece** is a base class, and specific **piece** types inherit from it.
- **MoveValidator** uses the current game and board state to verify move legality.

4. Class Diagram

The class diagram shows the main classes of the system, their attributes, and relationships, including inheritance between chess piece types and associations between game components.

(The class diagram is provided separately as a UML diagram)

5. Java Code

Added as a link on github.