SP-14 Blue Chess AI

Final Report

CS 4850 Fall Semester 2025

Conner Handley, Lowell Lucky, Ryan Nguyen, and Ulises Villagomez

Sharon Perry

December 8, 2025

https://14blue-sp.github.io/website/

https://github.com/14Blue-SP/Fall25-Project-part2

STATS AND STATUS		
LOC	1513	
Components/Tools	N/A	
Hours Estimate	368	
Hours Actual	400	
Status	Project is 100% complete and working as designed	

Introduction

This report describes the requirements, development, design, test plan, test report, and version control of the 2025 Fall semester Senior project, Blue Chess AI project. Throughout the requirements section, the functional and non-functional requirements are described, such as implementing the minimax algorithm and mobile friendly interface, respectively. The overarching design philosophy of the chess engine follows conventional designs and constructs akin to other online Chess engines. The design portion delves into system architecture and methodology elaborating on how features such as classes and the data flow are handled. The development portion highlights the technical portions of our chess engine and provides instructions on how to set up the program from the github repository. Moving on to the test plan, this section describes the test cases of the project, for example, testing the piece movement in player vs. player. In the test report section, the team shows the testing done by the test cases described in the test plan. Finally in the version control, the team describes how they managed the versions of the project using github and multiple branches to separate the versions, before pushing them into the main branch.

Challenges and Assumptions

There were a few challenges during the development of the Chess Engine, one of the major ones were the development of the save feature. This challenge was tough and eventually resulted with the feature being scrapped due to time constraints. Another challenge was the development of the minimax algorithm. This challenge came from the time the AI took to move a piece. When the limit of the maximum depth searched was high, the AI would take a long time to look through the best possible moves, thus disrupting the smoothness of gameplay. With each new development of the minimax algorithm that was implemented, the slower that AI became, thus it was decided to lower the limit of the maximum depth.

During the planning phase of the project, there were a few assumptions in regard to the Chess engine. Those were that users would be able to access the application through modern browsers, Internet connectivity is available, and the AI engine would run on the client side.

Requirements

Overview

The Chess AI Engine is an interactive platform that allows users to play against another local user, play against computer controlled AI, or to initiate an AI vs AI game. The engine will run entirely on a web browser using HTML, CSS, and JavaScript. The AI algorithm's functionality implements the minimax decision theory. It will provide features such as, save state, move validation, and game mode selection.

Project Scope

The goal of this project is to design and implement a web based chess platform accessible to desktop and mobile devices. The system will aim to be responsive, intuitive, and scalable. The Chess AI Engine will provide the user with the ability to play chess with 3 options of game mode selection, offering save state during play time. User account creation, user authentication, mobile application development, and game history storage is out of scope.

Constraints

There are two types of constraints, environmental and systemic. The environmental constraints involve where the engine will run in, and that is all modern web browsers, these include Safari, Chrome, and FireFox. The layout must be responsive to both desktop and mobile devices. Systemic constraints can include client hardware, potentially limiting the processing power for AI depth. The storage is capped by browser local storage for offline play. The minimum system requirements are 4 core CPU, 8GB of RAM, and 3GB storage.

Functional Requirements

Game Setup

- New Game: Player vs Player (local), Player vs AI, AI vs AI
- Board orientation selection for Player vs Player

Chess Board and Pieces

- Display 8x8 checkered board with proper coloring (light and dark squares)
- Render each side with 16 pieces appropriately colored
- Each side starts with 8 pawns, 2 rooks, 2 bishops, 2 knights, 1 queen, and 1 king
- Highlight selected piece and available legal moves

Chess Gameplay

- Validate moves according to chess rules
- Support for check, checkmate, and stalemate detection
- Undo/redo moves (optional)
- Pawn: Move forward 1 square, or 2 squares from starting position, capture diagonally
- Rook: Move horizontally or vertically any number of unobstructed squares
- Knight: Move in L shape (2 squares in one direction, 1 square perpendicular)
- Bishop: Move diagonally any number of unobstructed squares
- Queen: Move horizontally, vertically, or diagonally any number of unobstructed squares
- King: Move one square in any direction
- Castling: King moves 2 squares toward rook (kingside or queenside) when conditions met
- En Passant: Pawn captures opponent pawn that just moved 2 squares forward

- Pawn Promotion: When pawn reaches opposite end, promote to Queen, Rook, Bishop, or Knight

AI Opponent Functionality

- Implement minimax algorithm with alpha-beta pruning

Non-Functional Requirements

Performance

- AI moves calculated within 3 seconds on standard hardware

Usability

- Intuitive click and place/drag and drop piece movement
- Mobile friendly touch interface

Maintainability and Portability

- Modular JavaScript code for AI and UI separation
- Easily deployable to any web hosting environment

External Interface Requirements

User Interface Requirements

- Interactive chess board with click to move
- Real time move validation and feedback

Hardware Interface Requirements

- Client device with keyboard and mouse or touchscreen

Software Interface Requirements

- Runs in browser with HTML, CSS, and JavaScript

Analysis

Use Cases

- Start Game: User selects "New Game," chooses game mode and begins play
- Load Game: User selects “Load Game” and resumes play of previous game
- Make Move: User moves a piece, system validates move, updates board, and triggers AI move.
- Save Game: User exits current game and automatically saves and resumes upon load game

Data Flow

- User input (piece movement) > move validator > update board state
- AI Engine > calculate best move > update board state

Design

Design Considerations

The chess engine will be written in Java, following suit, Javascript will be used to host the chess engine on a website. Thus, users will require internet access to operate. The user must have a minimum of 8 Giga Bytes of RAM, a 4 core CPU, 3 Giga Bytes HDD. A mobile website version of the chess engine is currently not being planned, however it would be an interesting thing to tackle if there is time. Algorithms such as minimax will be used as well as the alpha beta pruning optimization.

The current constraint is hosting the website. While it may be launched virtually under the IP address attached to the Personal Computer that runs the executable, there needs to be a host to launch it to the internet. A possible solution to this would be by means of obtaining a powerful enough computer that would host the website 24/7.

Another constraint is the lack of a clear way to represent the chess pieces when outputting them to the user. There are several methods, however there are none that are considered “the best”. Each has their own drawbacks or complexity that might or might not boggle the system down.

The last constraint would be writing to a user's file system from within a website. One possible way to handle this is to make the user download a text file if they want to save the game in its state.

The best suited development strategy for our given time frame is Rapid Application Development (RAD). This methodology prioritizes the rapid creation of prototypes and iterative development over extensive, upfront planning. RAD focuses on a series of development cycles where the team quickly builds and refines the application based on user feedback. However, since we do not have any users to test out the application, the documentation people will test the application and give feedback. Waterfall and Agile were considered, though ultimately our team landed on RAD, as it can be considered slightly faster than agile. Waterfall is linear, where planning takes central stage, often slowing down the production cycle. Waterfall focuses highly on completing each phase one by one, slowing down the process entirely.

Architectural Strategies

The core of the engine will be built in Javascript allowing the project to be hosted on a website. Javascript was chosen due to the developers familiarity and proficiency. Furthermore, it allows the project to be hosted on a website, evading the need to create a separate website to host the information of the project as per the requirements.

As the chess engine is being written with HTML in mind, the inputs are going to be from a left mouse click. There are no current plans on implementing anything with a right mouse click. Outputs would include showing the chess board, and updating the chess board whenever a piece has been moved. Future plans of outputting a noise when a piece is moved have been considered.

The web-app will require permissions to access the user’s files. The chess engine will save a text file to the user’s file system as a way to save the current game. This will be done by writing a string containing the position of the current chess pieces on the board. The chess engine will then load up the previous game upon request by the user.

The minimax algorithm will be used to determine all the possible positions in chess. However, the possible positions in chess are estimated to be about 10^43 to 10^50. Thus, alpha-beta pruning will be used in tangent with the minimax algorithm to optimize the selection of available positions.

The current plan of the chess engine is to implement a player vs player, which will really only have the engine create outputs whenever any of the two players moves a chess piece. The chess engine will only allow legal and pseudo-legal moves as well as any special moves such as castling. Player vs computer is also currently planned to be developed. This mode will allow the player to face the chess engine. The computer will use the minimax algorithm to determine the best possible move against the player. Future plans of adding computer vs computer have been discussed, and might be implemented if time allows.

System Architecture

The system is partitioned into two packages. The first package is the Piece package, which contains the class Piece. The class piece is an abstract class containing methods that allow the pieces to function. The Piece class has 6 children that override the methods to whatever they require. The second package is the Main package. This is where all the logic behind the game of chess happens. There is the Main class, where the engine will be housed. The check scanner, checks for checks/checkmates, Board class, and Scorer.

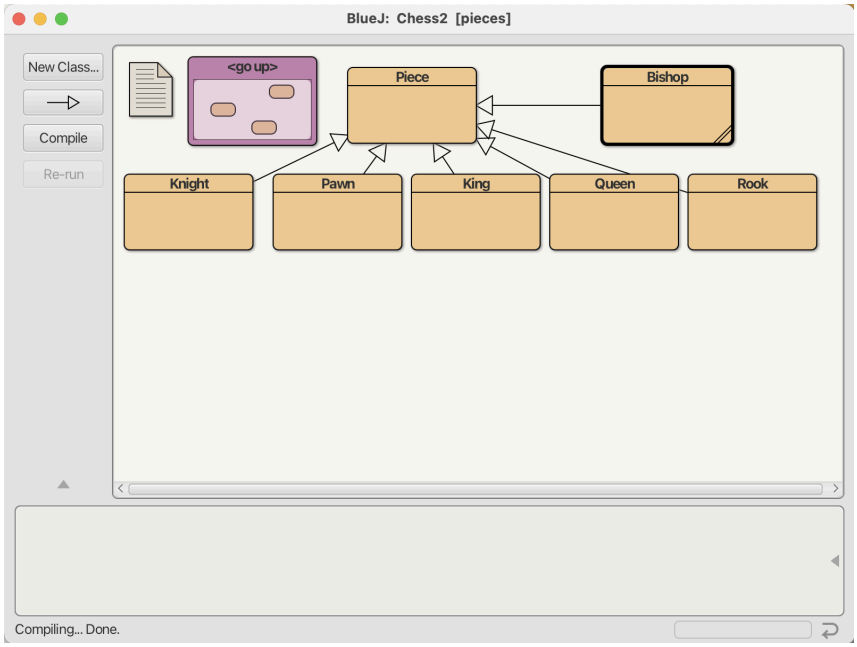


Diagram of architecture of class structure for chess pieces

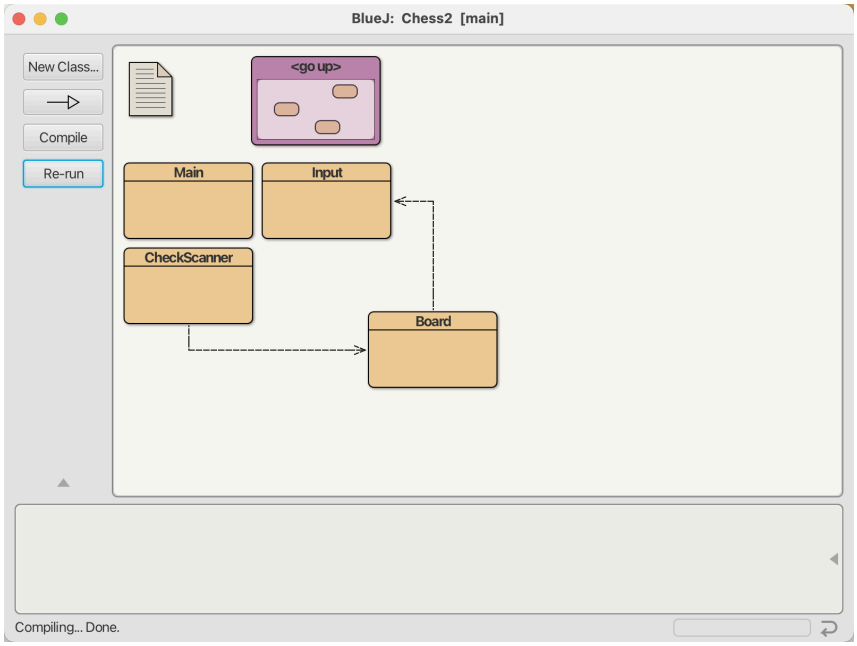


Diagram of overarching architecture and dataflow

Detailed System Design

Classifications

1. Piece Package
    1.1. Piece Class
        1.1.1. King Piece subclass
        1.1.2. Queen Piece subclass
        1.1.3. Bishop Piece subclass
        1.1.4. Rook Piece subclass
        1.1.5. Knight Piece subclass
        1.1.6. Pawn Piece subclass
2. Main Package
    2.1 Model
        2.1.1. Board Model
        2.1.2. CheckScanner
        2.1.2. GameModel
        2.1.3. Scorer

Constraints

The first of constraints include storage, due to the use of the minimax algorithm, it involves deep processing and exponential growth resulting in excessive memory consumption. An additional constraint is the general processing speed, as chess has innate complexities with every move decision. This causes a blatant limitation of how deep our model can look into moves, while still maintaining a fluid program run time. We are planning on going 3 layers deep for each move throughout the game to ensure a stable machine state. Another limitation is the

ability to only save one game at a time. Any game saved after an initial save will overwrite the prior save. We will also have the limitation of chess piece movement as follows:

1. Board / Main rules
    a. Boundaries (no piece can make a move off the board)
        i. Rows called ranks
        ii. Columns called files
    b. All pieces maintain respective movesets.
        i. All pieces except for King can give a check on opponent King
        ii. All pieces have the ability to perform the capture of a piece if legally allowed.
    c. No piece may take its own color, or pass through it
        i. Except for Knight see point 5.a for move details
    d. White moves first
2. King
    a. Move 1 legal space in any direction, that does not move into check
    b. Must move out of check if in check, or block the check with another piece
3. Queen
    a. Can move any legal file, rank, or diagonal, by any amount
4. Bishop
    a. Can move any legal diagonal any number of spaces.
        i. Can't move off it's diagonal starting color (i.e. bishop who starts on light square can't move onto a dark square)
5. Knight
    a. Can moves in a legal "L" shape
        i. Two squares either horizontal or vertical, then one square perpendicular to that direction
6. Rook
    a. Can move in a legal file or rank any number of spaces
7. Pawn
    a. Can move one space forwards
        i. Can move two spaces forwards if Pawn hasn't moved
        ii. Can take diagonally
            1. Can perform En Passant (see glossary)
    b. Must be promoted to a Knight, Bishop, Rook or Queen, upon reaching opponent back rank

Resources

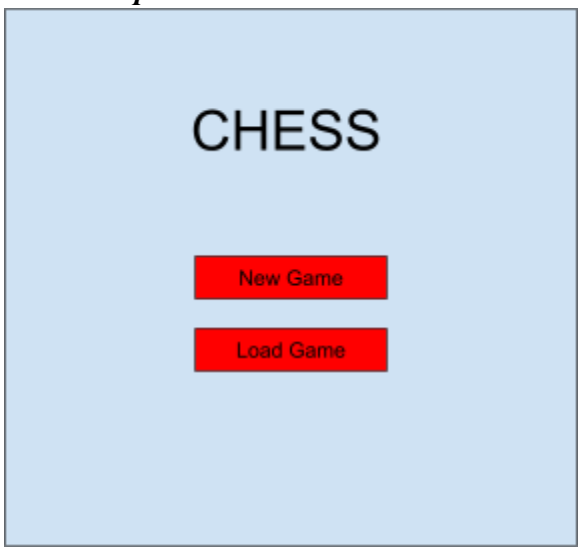
The current software libraries the program is using are javafx, java.utils, javax.awt, and java.arrays. The software library javafx is used to create interactable interfaces that take in an input, usually from the mouse, or keyboard input, and display it on a computer screen as an output. The next software library used, java.utils, is an extensive library that adds multiple necessary classes that expand the usability of Java. Javax.awt is a platform dependent API that helps the programmer create GUI's or windows. The last currently used software library is java.arrays, and as the name implies, it gives access to the classes that make up arrays in java. There are currently no known race conditions that could occur with any of the functions or classes in the program.

Interface/Experts

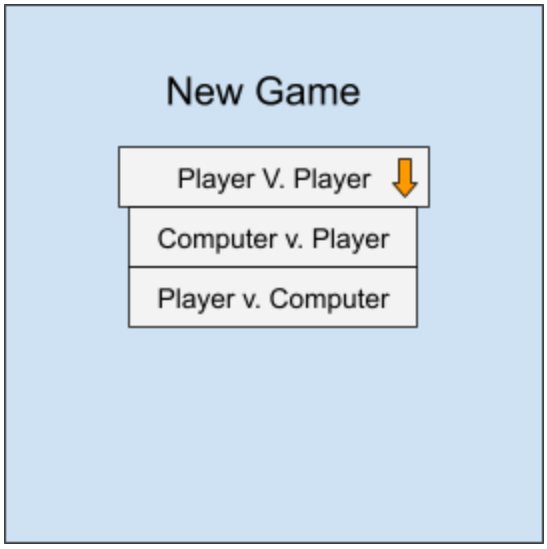
The Piece class has 4 different methods, a constructor, a method to check if a move collides with another piece, a method to display the piece at the proper location, and one to move the piece if the method is legal. Each subclass inherits the 4 methods from the piece class, and overrides them to suit the needs of each subclass.

The first method is the constructor, this allows any object to be created, the original method is public Piece(Board board). The constructor takes in a Board object from the Board class. The next method checks to see if the current move is valid, public boolean isValidMove(int col, int row). This method takes in two integers, one representing the columns and the other representing the rows. Since each piece has different moves that are considered legal, this method is overridden in each subclass, each one defining their own legal move. The third method is public void paint(Graphics gfx), which determines the proper location of the piece. The method displays the piece at the determined location on the board. The final method in the pieces class is public void move(int newCol, int newRow), this method moves the piece after the move has been determined legal.

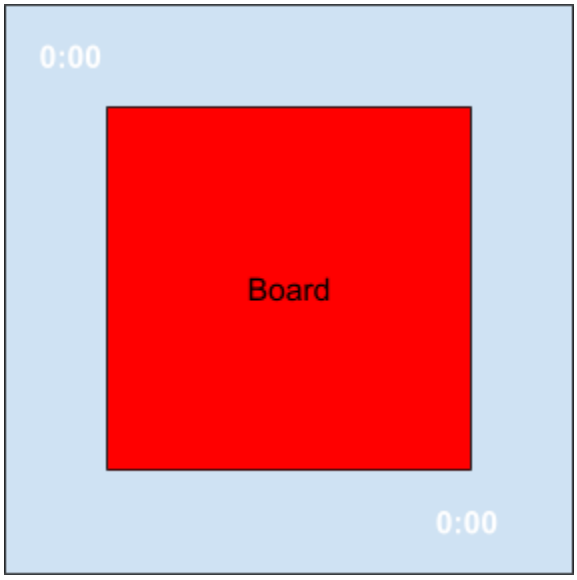
UI Mockup



Initial Start up Screen



Game mode selection screen



Rendered Board

Development

Technical Summary

The core game engine serves as the foundational logic layer, maintaining comprehensive game state management and enforcing all standard chess rules according to FIDE regulations. The engine tracks board position, piece locations, turn sequence, and special conditions including castling eligibility, en passant opportunities, pawn promotions, etc. Move validation occurs in real time, ensuring all piece movements adhere to their respective constraints. The system detects terminal game states including check, checkmate, stalemate, and draw conditions

The engine supports multiple gameplay modes to accommodate different use cases. Player versus player mode allows two human players to compete locally, while player versus AI mode enables single-player experiences against the computer. The system also includes an AI versus AI mode for testing, demonstration, or analysis purposes.

The artificial intelligence component employs classical game tree search algorithms optimized for chess. The decision making process utilizes the minimax algorithm with alpha-beta pruning, a well established approach in adversarial game theory. The minimax algorithm recursively explores the game tree by alternating between maximizing the AI's position and minimizing the opponent's advantage, evaluating terminal nodes through a heuristic evaluation function that considers factors such as positional control, king safety, and overall tactical opportunities. Alpha-beta pruning enhances computational efficiency by eliminating branches that probably cannot affect the final move selection, reducing the effective branching factor and enabling deeper search depths within equivalent processing time. This optimization typically achieves search depth reductions of 30-40% while guaranteeing identical move selection to the non-pruned minimax implementation. The AI would as a result operate within strict time constraints (< 3 seconds per move) to maintain responsive gameplay.

Project Setup

Initial Setup

- Clone or download the project repository to your local machine
- Ensure you have the required programming environment installed

Dependency Installation

- Install all required dependencies listed in the project requirements file
- This typically includes libraries for game state management, UI rendering, etc
- A working browser (to run JavaScript programs)

Configuration

- Review and configure game settings i.e. AI difficulty levels, Player color

Running the Application

- Launch the main application file or entry point
- Select your desired game mode i.e. Player vs Player, Player vs AI

Testing and Verification

- Verify AI functionality by testing against known chess positions
- Confirm that the alpha-beta pruning optimization is functioning correctly by checking performance metrics

Test Plan

Test Objectives

The primary objectives of this testing plan are to:

- Verify that the chess engine correctly implements standard chess rules and move validation
- Validate that the AI opponent makes legal and strategically sound moves
- Ensure the user interface responds correctly to user interactions (piece selection, movement, etc.)
- Confirm that game state is maintained accurately throughout play
- Verify cross-browser compatibility and responsive design functionality
- Validate that the game can detect check, checkmate, and stalemate conditions

Scope of Testing

In-Scope

- Chess board rendering and display
- Piece movement validation
- AI engine move generation
- Game state management (turn handling, piece capture, etc.)
- Special moves (castling, en passant, pawn promotion, etc.)
- Win/loss/draw condition detection
- User interface interactions (click, drag and drop)
- New game and reset functionality
- Browser compatibility (Chrome, Firefox, Safari, Edge)

Out-of-Scope

- Game saving and loading
- Mobile accessibility

Test Cases

Test Case ID	**Description**	**Expected Result**
TC-1	Board initialization	8x8 chess board displays with pieces in correct starting positions

TC-2	Pawn movement - single square	Pawn moves one square forward to empty square
TC-3	Pawn movement - double square	Pawn moves two squares forward from starting position
TC-4	Pawn diagonal capture	Pawn captures opponent piece diagonally
TC-5	Knight movement	Knight moves in valid L-shape pattern
TC-6	Bishop movement	Bishop moves diagonally any number of squares
TC-7	Rook movement	Rook moves horizontally or vertically any number of squares
TC-8	Queen movement	Queen moves diagonally, horizontally, or vertically
TC-9	King movement	King moves one square in any direction
TC-10	Castling - king side	King and rook perform kingside castling when conditions met
TC-11	Castling - queen side	King and rook perform queenside castling when conditions met
TC-12	Castling prevention - king moved	Castling blocked after king has moved
TC-13	Castling prevention - through check	Castling blocked when king passes through check
TC-14	En passant capture	Pawn captures opponent pawn via en passant
TC-15	Pawn promotion	Pawn reaching opposite end promotes to queen/rook/bishop/knight
TC-16	Check detection	Game correctly identifies when king is in check
TC-17	Checkmate detection	Game correctly identifies checkmate and ends game
TC-18	Stalemate detection	Game correctly identifies stalemate and declares draw

TC-19	Illegal move prevention	System prevents illegal moves from being executed
TC-20	AI move generation	AI makes legal moves within reasonable time (< 3 seconds)
TC-21	Turn alternation	Turns properly alternate between player and AI
TC-22	Drag and drop functionality	Pieces can be moved via drag and drop
TC-23	Click to move functionality	Pieces can be moved by clicking source and destination
TC-24	Captured piece display	Section on the board that shows pieces captured
TC-25	New game button	New game button resets board to starting position
TC-26	Browser compatibility - Chrome	Game functions correctly in Chrome browser
TC-27	Browser compatibility - Firefox	Game functions correctly in Firefox browser
TC-28	Browser compatibility - Safari	Game functions correctly in Safari browser
TC-29	Browser compatibility - Edge	Game functions correctly in Edge browser

Test Procedures

Procedure for TC-1: Board Initialization

1. Open index.html in web browser
2. Select color choice if choosing game Player vs Ai mode
3. Observe initial board rendering
4. Verify 8x8 grid is displayed
5. Verify white pieces are on rows 1-2
6. Verify black pieces are on rows 7-8
7. Verify piece positions match standard chess starting position
8. Expected: Board displays correctly with all pieces in proper starting positions

Procedure for TC-2: Pawn Movement - Single Square

1. Load game in browser
2. Click on white pawn at e2
3. Verify legal moves are highlighted (e3 and e4)
4. Click on e3 square
5. Verify pawn moves from e2 to e3
6. Expected: Pawn successfully moves one square forward

Procedure for TC-3: Pawn Movement - Double Square

1. Start new game
2. Click on white pawn at d2
3. Verify both d3 and d4 are highlighted as legal moves
4. Click on d4 square
5. Verify pawn moves from d2 to d4 (two squares)
6. Verify turn passes to opponent
7. Expected: Pawn successfully moves two squares forward from starting position

Procedure for TC-4: Pawn Diagonal Capture

1. Start new game and set up position where opponent piece is diagonally adjacent to pawn
2. Make moves: e4, d5
3. Click on white pawn at e4
4. Verify d5 is highlighted as legal capture move
5. Click on d5
6. Verify white pawn captures black pawn at d5
7. Verify captured piece is removed and shown in captured pieces display
8. Expected: Pawn successfully captures diagonally

Procedure for TC-5: Knight Movement

1. Start new game
2. Click on white knight at b1
3. Verify valid L-shaped moves are highlighted (a3, c3)
4. Click on c3
5. Verify knight moves to c3
6. Click on knight at c3
7. Verify new valid moves include a2, a4, b5, d5, e4, e2, d1, b1
8. Expected: Knight moves in valid L-shape pattern only

Procedure for TC-6: Bishop Movement

1. Start new game
2. Move pawns to clear diagonal: e4, e5
3. Click on white bishop at f1
4. Verify diagonal moves are highlighted (e2, d3, c4, b5, a6)
5. Click on a6

6. Verify bishop moves diagonally from f1 to a6
7. Expected: Bishop moves diagonally any number of unobstructed squares

Procedure for TC-7: Rook Movement

1. Start new game
2. Move pawn at a2 to a4, then a5
3. Click on white rook at a1
4. Verify vertical moves are highlighted (a2, a3, a4)
5. Move rook to a3
6. Click on rook at a3
7. Verify horizontal and vertical moves are highlighted
8. Expected: Rook moves horizontally or vertically any number of squares

Procedure for TC-8: Queen Movement

1. Start new game
2. Move pawns to free queen: d4, d5
3. Click on white queen at d1
4. Verify diagonal, horizontal, and vertical moves are all highlighted
5. Move queen to h5
6. Click on queen at h5
7. Verify all direction moves are available
8. Expected: Queen moves in all eight directions (diagonal, horizontal, vertical)

Procedure for TC-9: King Single Square Movement

1. Start new game
2. Move pawns to allow king movement: e4, e5
3. Click on white king at e1
4. Verify only adjacent squares are highlighted (e2)
5. Move king to e2
6. Click on king at e2
7. Verify only one-square moves in all directions are shown
8. Expected: King moves exactly one square in any direction

Procedure for TC-10: Castling - Kingside

1. Start new game
2. Clear path between king and kingside rook: Move Nf3, Ng8-f6, then Bg1-e2
3. Click on white king at e1
4. Verify castling option is highlighted at g1
5. Click on g1
6. Verify king moves to g1 and rook automatically moves to f1
7. Expected: Kingside castling executes with both pieces moving correctly

Procedure for TC-11: Castling - Queenside

1. Start new game
2. Clear path between king and queenside rook: Move d4, d5, then Nc3, Be2, Qd2
3. Click on white king at e1
4. Verify queenside castling option is highlighted at c1
5. Click on c1
6. Verify king moves to c1 and rook moves to d1
7. Expected: Queenside castling executes with both pieces moving correctly

Procedure for TC-12: Castling Prevention - King Moved

1. Start new game
2. Clear path for kingside castling
3. Move white king from e1 to e2, then back to e1
4. Click on king at e1
5. Verify castling is NOT highlighted as an option
6. Expected: Castling is blocked after king has moved previously

Procedure for TC-13: Castling Prevention - Through Check

1. Set up position where king would pass through attacked square during castling
2. Arrange black queen or bishop to attack f1 square
3. Click on white king at e1
4. Verify kingside castling is NOT available
5. Expected: Castling is prevented when king passes through check

Procedure for TC-14: En Passant Capture

1. Start new game
2. Move white pawn from e2 to e4 (then AI or manual move)
3. Move white pawn from e4 to e5
4. Move black pawn from d7 to d5 (two squares, landing beside white pawn)
5. Click on white pawn at e5
6. Verify d6 is highlighted as legal en passant capture
7. Click on d6
8. Verify white pawn moves to d6 and black pawn at d5 is captured
9. Expected: En passant capture executes correctly

Procedure for TC-15: Pawn Promotion

1. Set up position where white pawn is at row 7 (e.g., e7)
2. Click on white pawn at e7
3. Move pawn to e8
4. Verify promotion dialog appears with options (Queen, Rook, Bishop, Knight)
5. Select Queen
6. Verify pawn is replaced with Queen at e8

7. Expected: Pawn promotes to selected piece upon reaching opposite end

Procedure for TC-16: Check Detection

1. Start new game
2. Play moves to put opponent king in check (e.g., Scholar's Mate setup)
3. Move piece that places king in check
4. Verify "Check" indicator appears
5. Verify king in check is highlighted or indicated visually
6. Expected: Game correctly identifies and displays check condition

Procedure for TC-17: Checkmate Detection

1. Execute Scholar's Mate sequence: e4, e5; Bc4, Nc6; Qh5, Nf6; Qxf7#
2. Observe game response after final move
3. Verify "Checkmate" message appears
4. Verify game prevents further moves
5. Verify winner is declared
6. Expected: Game correctly identifies checkmate and ends game

Procedure for TC-18: Stalemate Detection

1. Set up stalemate position (king with no legal moves but not in check)
2. Execute move that creates stalemate
3. Verify "Stalemate" or "Draw" message appears
4. Verify game ends and declares draw
5. Expected: Game correctly identifies stalemate and declares draw

Procedure for TC-19: Illegal Move Prevention

1. Start new game
2. Click on white pawn at e2
3. Attempt to click on illegal square (e.g., e5, f3, d3)
4. Verify pawn does NOT move to illegal square
5. Try moving bishop before clearing pawn path
6. Verify bishop cannot move through pieces
7. Expected: System prevents all illegal moves from being executed

Procedure for TC-20: AI Move Generation

1. Start new game with AI as black
2. Make any legal opening move as white (e.g., e4)
3. Start timer/stopwatch
4. Wait for AI to respond
5. Stop timer when AI completes move
6. Verify move is legal
7. Verify response time is under 3 seconds

8. Expected: AI makes legal move within reasonable time

Procedure for TC-21: Turn Alternation

1. Start new game
2. Make white's first move
3. Verify only black pieces can be moved
4. Wait for or make black's move
5. Verify only white pieces can be moved
6. Expected: Turns properly alternate between players

Procedure for TC-22: Drag and Drop Functionality

1. Start new game
2. Click and hold white pawn at e2
3. Drag pawn to e4 square
4. Release mouse button
5. Verify pawn moves to e4
6. Test with multiple piece types
7. Expected: All pieces can be moved via drag and drop

Procedure for TC-23: Click to Move Functionality

1. Start new game
2. Click on white pawn at d2 (first click)
3. Verify piece is selected and moves are highlighted
4. Click on d4 square (second click)
5. Verify pawn moves to d4
6. Test with multiple piece types
7. Expected: Pieces move with two click method (select then destination)

Procedure for TC-24: New Game Button

1. Start game and make several moves
2. Locate "New Game" or "Reset" button
3. Click the button
4. Verify confirmation dialog appears
5. Confirm new game
6. Verify board resets to starting position
7. Expected: New game button completely resets game state

Procedure for TC-25: Captured Pieces Display

1. Start new game
2. Make moves that result in captures
3. Locate captured pieces display area
4. Verify captured white pieces appear in white's captured section

5. Verify captured black pieces appear in black's captured section
6. Verify pieces display with correct icons/images
7. Expected: Captured pieces are shown correctly in designated area

Procedure for TC-26: Browser Compatibility - Chrome

1. Open game in Google Chrome (latest version)
2. Execute test cases TC-001, TC-002, TC-010, TC-017, TC-020, TC-026
3. Note any rendering issues, console errors, or functionality failures
4. Test all interactive elements (buttons, piece movement, etc)
5. Verify CSS styling displays correctly
6. Expected: All features function correctly without errors in Chrome

Procedure for TC-27: Browser Compatibility - Firefox

1. Open game in Mozilla Firefox (latest version)
2. Execute test cases TC-001, TC-002, TC-010, TC-017, TC-020, TC-026
3. Note any rendering issues, console errors, or functionality failures
4. Test all interactive elements (buttons, piece movement, etc.)
5. Verify CSS styling displays correctly
6. Expected: All features function correctly without errors in Firefox

Procedure for TC-028: Browser Compatibility - Safari

1. Open game in Safari (latest version, macOS or iOS)
2. Execute test cases TC-001, TC-002, TC-010, TC-017, TC-020, TC-026
3. Note any rendering issues, console errors, or functionality failures
4. Test all interactive elements (buttons, piece movement, etc.)
5. Verify CSS styling displays correctly
6. Expected: All features function correctly without errors in Safari

Procedure for TC-029: Browser Compatibility - Edge

1. Open game in Microsoft Edge (latest version)
2. Execute test cases TC-001, TC-002, TC-010, TC-017, TC-020, TC-026
3. Note any rendering issues, console errors, or functionality failures
4. Test all interactive elements (buttons, piece movement, etc.)
5. Verify CSS styling displays correctly
6. Expected: All features function correctly without errors in Edge

Test Environment

Hardware Requirements

- Desktop/Laptop computer
- Minimum 4GB RAM
- 4 Core CPU

- 3 Giga Bytes HDD
- Standard keyboard and mouse
- Display resolution: 1366x768 minimum

Software Requirements

- Operating Systems: Windows 10/11, macOS 10.15+, Linux (Ubuntu 20.04+)
- Web Browsers:
    - Google Chrome (version 100+)
    - Mozilla Firefox (version 100+)
    - Safari (version 15+)
    - Microsoft Edge (version 100+)
- Text editor/IDE for code inspection (optional)
- Browser Developer Tools (for debugging)

Network Requirements

- Internet connection required to initially access and download files from GitHub
- No network connection required for functionalities

Test Data

Sample Game Positions

- Standard starting position
- Mid-game position with castling available
- Endgame position (king and pawn vs king)
- Checkmate positions (scholar's mate, back rank mate, smothered mate)
- Stalemate position
- En passant setup position
- Pawn promotion position

Test Scenarios

- Opening moves: e4, d4, Nf3, c4
- Common tactical patterns: pins, forks, skewers
- Special move scenarios: all castling variations, en passant, all promotion options

AI Test Cases

- AI playing as white and black
- Time constrained move generation scenarios

Test Report

Test Execution Summary

Test Case ID	Description	Pass	Fail	Severity	Notes
TC-1	Board initialization	Yes			No issues
TC-2	Pawn movement - single square	Yes			No issues
TC-3	Pawn movement - double square	Yes			No issues
TC-4	Pawn diagonal capture	Yes			No issues
TC-5	Knight movement	Yes			No issues
TC-6	Bishop movement	Yes			No issues
TC-7	Rook movement	Yes			No issues
TC-8	Queen movement	Yes			No issues
TC-9	King movement	Yes			No issues
TC-10	Castling - kingside	Yes			No issues
TC-11	Castling - queenside	Yes			No issues
TC-12	Castling prevention - king moved	Yes			No issues
TC-13	Castling prevention - through check	Yes			No issues
TC-14	En passant capture	Yes			No issues
TC-15	Pawn promotion	Yes			Works as intended
TC-16	Check detection	Yes			Works as intended
TC-17	Checkmate detection	Yes			Works as intended
TC-18	Stalemate detection	Yes			Works as intended
TC-19	Illegal move prevention	Yes			No issues

TC-20	AI move generation	Yes		Moderate	Functional but slow
TC-21	Turn alternation	Yes			No issues
TC-22	Drag and drop functionality	Yes		Minor	Works, however vs. AI will take a few seconds to show results
TC-23	Click to move functionality		Yes	Moderate	Not operational
TC-24	Captured piece display		Yes	Minor	Not operational
TC-25	New game button	Yes			No issues
TC-26	Browser compatibility - Chrome	Yes			No issues
TC-27	Browser compatibility - Firefox	Yes			No issues
TC-28	Browser compatibility - Safari	Yes			No issues
TC-29	Browser compatibility - Edge	Yes			No issues

Severity Levels

- **Critical:** Core functionality broken, game unplayable
- **Moderate:** Feature not working as intended, workaround available
- **Minor:** Cosmetic issue or minor inconvenience, does not affect gameplay

Test Results Summary

Total Test Cases: 29
Passed: 27
Failed: 2
Pass Rate: 93.1%

Version Control

The development team utilizes GitHub as the primary version control system for this project. All source code, including HTML, CSS, and JavaScript files, is maintained in a centralized GitHub repository with a main/master approach where the main branch is a stable product always ready for deployment. Temporary branches are created for bug fixes, new features, and testing that are isolated from the main branch maintaining stability. Each team member commits changes with descriptive commit messages that reference specific features or test cases being addressed. Pull requests are reviewed by at least one other team member before merging to ensure code quality and prevent integration issues. Backup and recovery is handled by mirror cloning at every version update and any additional subsequent mirror clones as needed.

Summary

This report provides an examination of the complete software development lifecycle for the web-based Chess Engine AI project, documenting the journey from initial conceptualization through systematic testing and validation. The project successfully delivered a fully functional chess application capable of enforcing all standard chess rules, detecting complex game states including check, checkmate, and stalemate, and providing intelligent opposition through an AI opponent utilizing minimax algorithm with alpha-beta pruning optimization.

The development process began with careful consideration of project scope, technical constraints, and resource limitations inherent to a four person team structure. Initial assumptions regarding implementation complexity, particularly concerning special chess moves such as castling, en passant, and pawn promotion, proved both accurate in some areas and underestimated in others. The team navigated significant technical challenges including but not limited to move validation logic that required accounting for edge cases and AI performance optimization to maintain responsive gameplay within acceptable time constraints. The testing phase encompassed distinct test cases covering functional requirements, user interface interactions, AI behavior across multiple difficulty levels, and browser compatibility validation.

Additionally other considerations arose for the development team aside from technical challenges. Time management and resource allocation became priorities highlighting the need for continuous communication between every role to establish proper efficiency and prevent roadblocks throughout development.

Looking forward, this project establishes a solid foundation for potential enhancements including networked multiplayer functionality, persistent game saving and loading capabilities, and expanded game analysis features. Ultimately, this Chess Engine AI project successfully achieved its technical objectives while providing the development team with invaluable hands-on experience in collaborative software development.

Appendix A: Glossary

Agile: Software development methodology, made to be faster than the methods that came before.

Alpha-Beta Pruning: An optimization used in conjunction with the minimax algorithm.
Castling: Special chess move where if the King moves two spaces towards the Rook, they switch places. Can only happen if neither two pieces have moved from their original spot.
En Passant: A capture by a pawn of an enemy pawn on the same rank and an adjacent file that has just made a two square move.
Minimax Algorithm: Simple Artificial Intelligence Algorithm used to decide the best possible move in the game. The algorithm considers all possible moves that can be made in the game it is implemented for.
Rapid Application Development (RAD): Software development methodology that prioritizes the creation of prototypes and iterative development over extensive, upfront planning.
Waterfall: Linear software development methodology where planning takes a central stage. Each phase must be completed before moving on.

Appendix B: References

Chess. (2025). Retrieved from https://en.wikipedia.org/wiki/Chess

Official FIDE Chess Rules (https://www.fide.com/)