**INTERNATIONAL UNIVERSITY**

**VIETNAM NATIONAL UNIVERSITY, HCM CITY**

**School of Computer Science & Engineering**



**DATA STRUCTURES AND ALGORITHMS REPORT**

**Topic: Chess Game**

| **Full name** | **Student ID** |
| --- | --- |
| Nguyễn Hoàng Phúc | ITITWE21082 |
| Phạm Thái Quốc | ITITDK21050 |
| Võ Nguyên Thanh Liêm | ITITIU21069 |
| Nguyễn Ngọc Bảo Hân | ITITWE21069 |



**Table of Contents**

[**I. INTRODUCTION**](#_heading=h.30j0zll) **2**

[**II. GAME RULES**](#_heading=h.tyjcwt) **3**

[**III. CLASS DIAGRAMS**](#_heading=h.44sinio) **9**

[**IV.**](#_heading=h.44sinio) **GIT EXPLANATION 16**

**V. LECTURE APPLICATIONS 16**

[**VI. LIMITATION**](#_heading=h.3j2qqm3) **19**

**VII. CONCLUSION 19**

**VIII. TASKING 20**

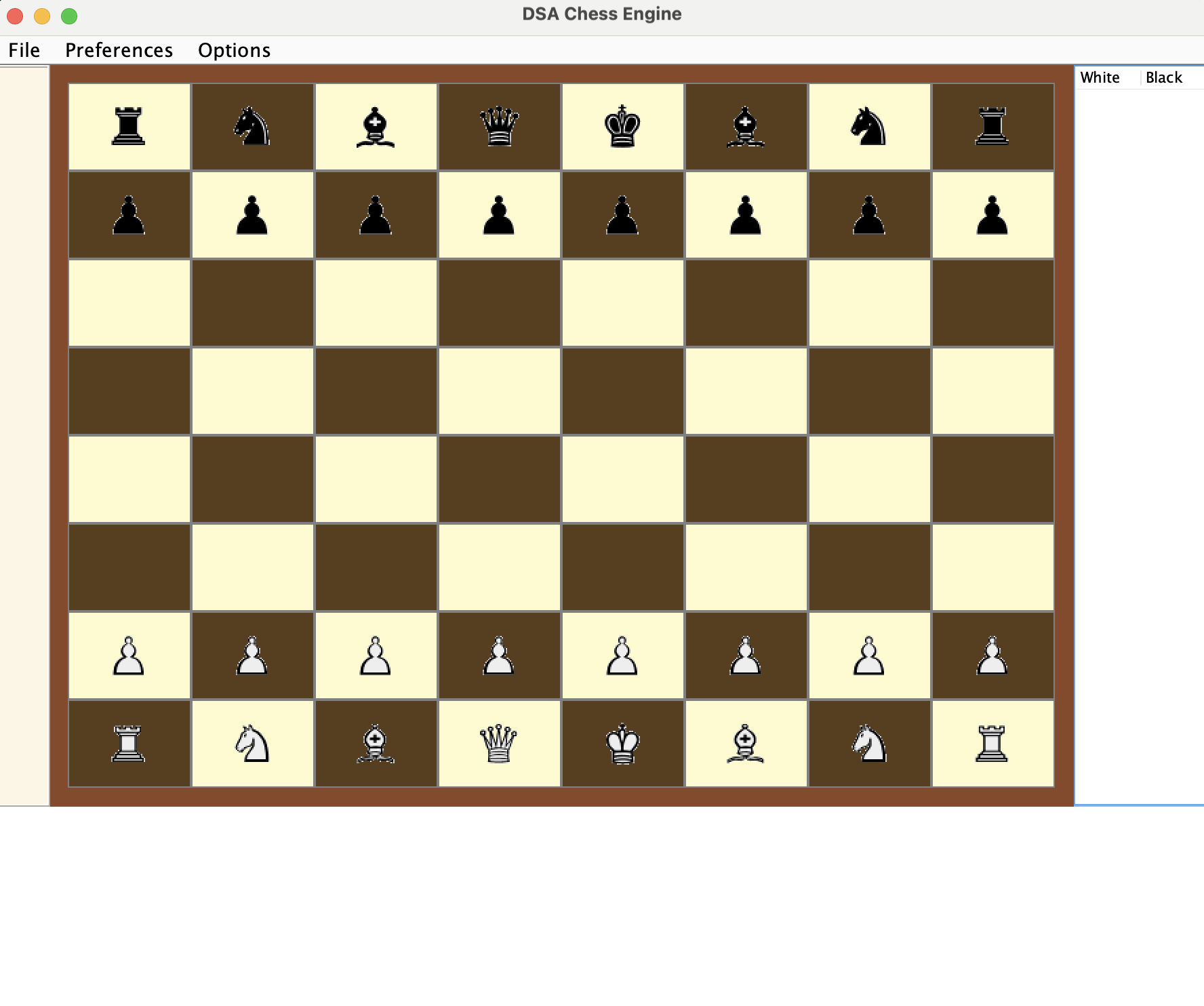
**IX**[**. REFERENCES**](#_heading=h.4i7ojhp) **20**

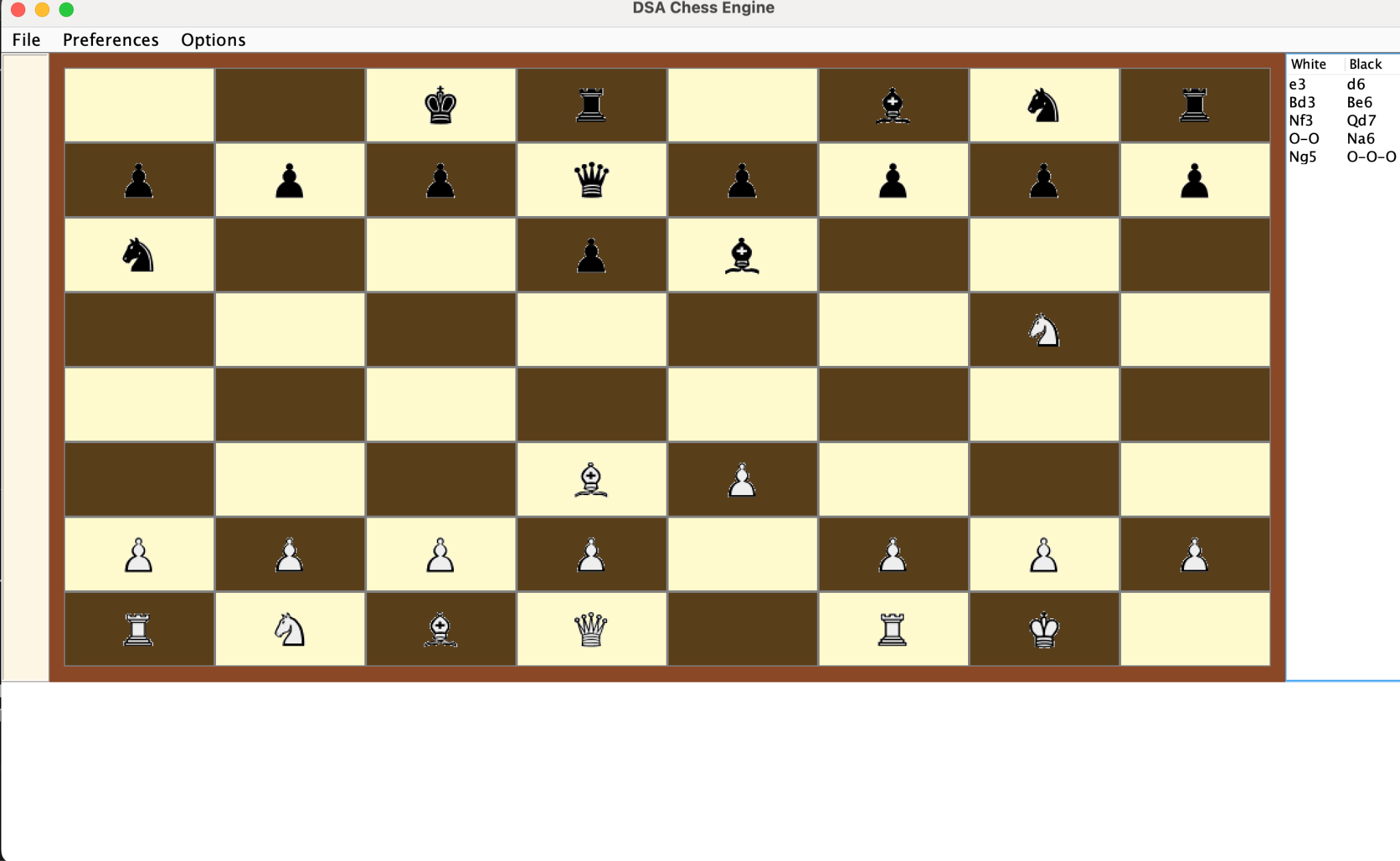
# I. INTRODUCTION

This report presents an overview of the chess game project focused on Data Structures and Algorithms (DSA). It details the project's objectives, significance, and methodologies applied during development. The primary goal is to develop a chess game that leverages advanced data structures and algorithms to manage game logic, optimize performance, and enhance the overall gaming experience. This project aims to deepen the understanding of data structures and algorithms by applying them in a practical and engaging context, while also developing skills in game development and strategic thinking.

**Semester:** 2 (2023-2024)

# II. GAME RULES











# *2.1 How to play*

### *2.1.1 Start the game*

* *White pieces always move first.*
* *Players alternate turns.*

### *2.1.2 Movements*

* **Pawn**: Moves forward one square, captures diagonally.
* **Rook**: Moves horizontally or vertically any number of squares.
* **Knight**: Moves in an "L" shape, can jump over pieces.
* **Bishop**: Moves diagonally any number of squares.
* **Queen**: Moves horizontally, vertically, or diagonally any number of squares.
* **King**: Moves one square in any direction.

### *2.1.3 Understanding the different types of character*

Each piece has distinct movement rules and roles in the game.

### *2.1.4 Questions on the road*

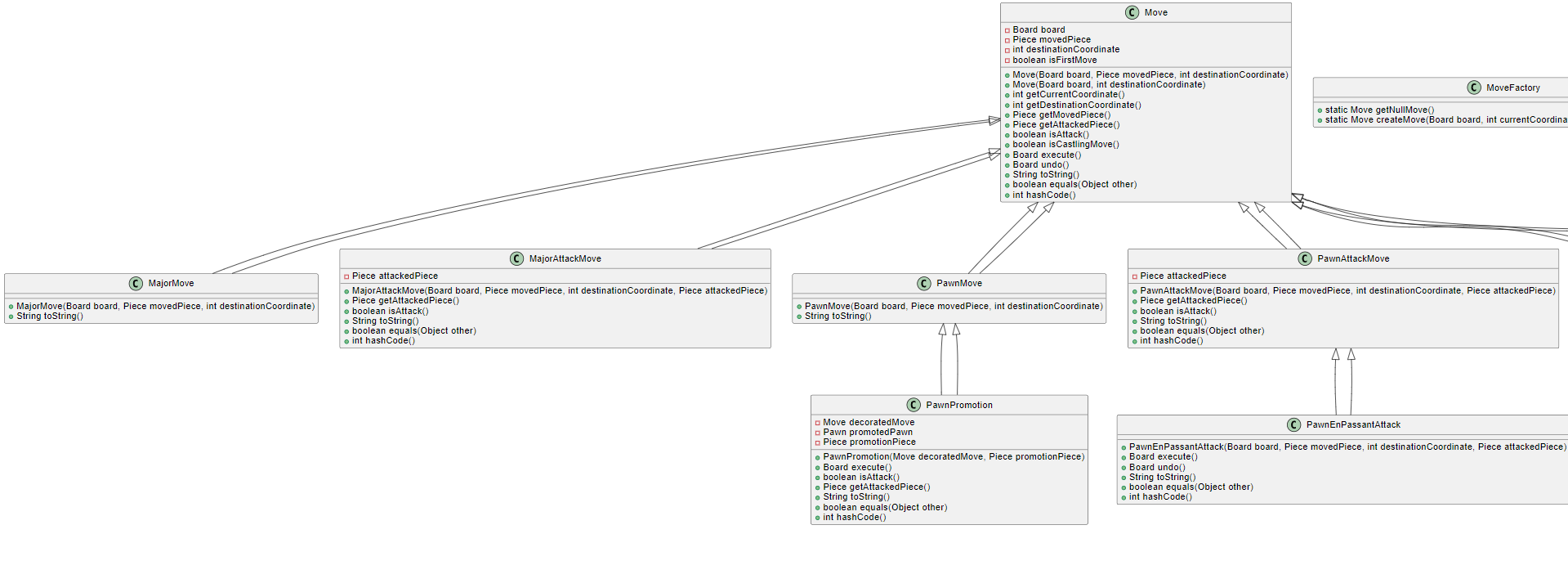
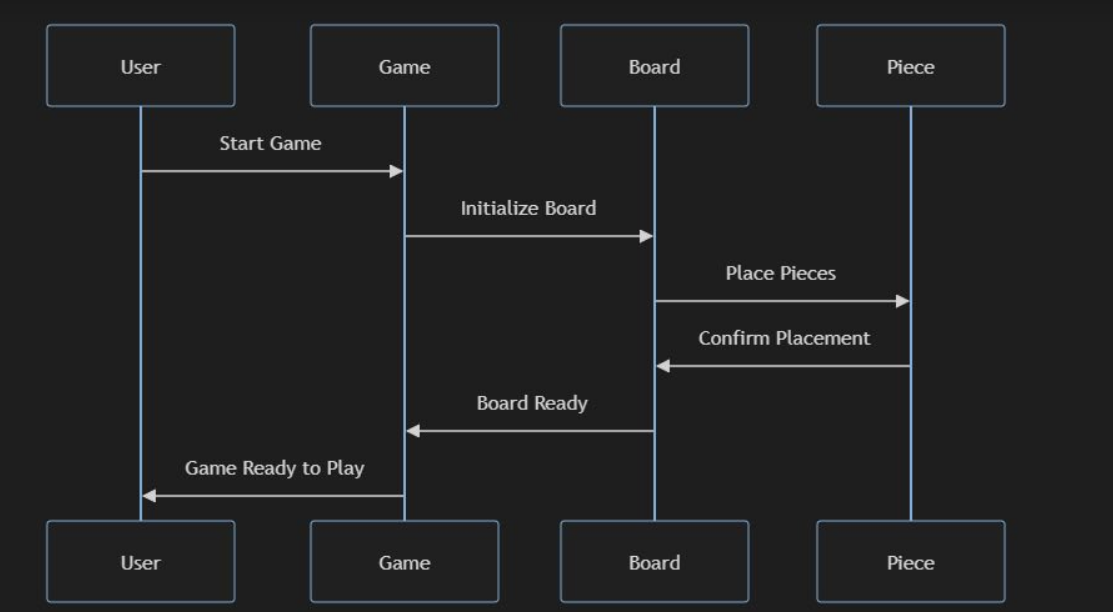
Frequently asked questions about game mechanics and rules are addressed.

## *2.2 Mechanics*

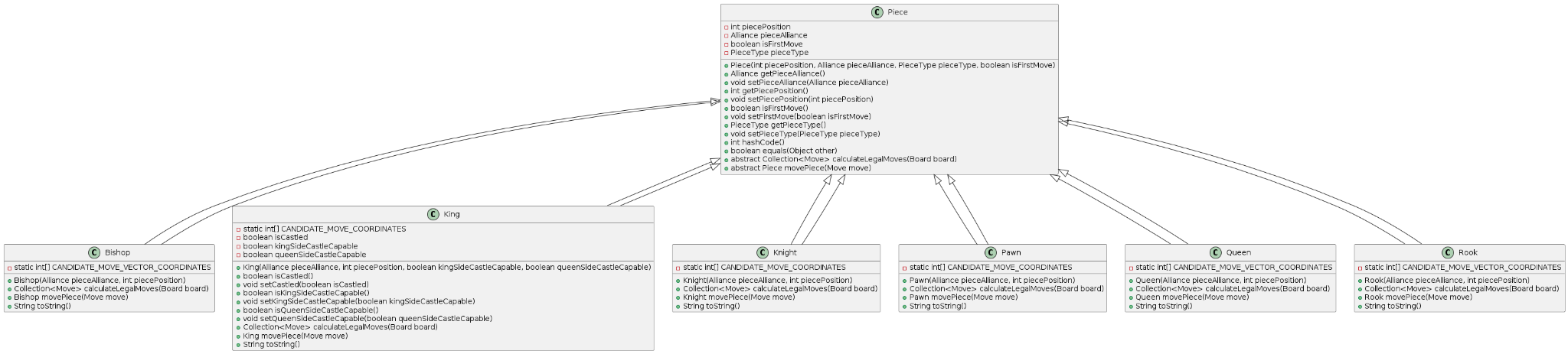
* **Check and Checkmate**: Algorithms for handling checks, capturing pieces, and blocking moves.
* **Stalemate and Draw**: Algorithms for detecting stalemate, threefold repetition, and the 50-move rule.

## *2.3 Score pattern*

Points and rankings based on game performance and outcomes.

**III. CLASS DIAGRAMS**  






Piece Class ( Super Class):

**Attributes:**

* int piecePosition: Represents the position of the piece on the board.
* Alliance pieceAlliance: Denotes the alliance (color) of the piece, either white or black.
* boolean isFirstMove: Indicates if it's the first move of the piece.
* PieceType pieceType: Enum representing the type of the piece (King, Queen, Rook, etc.).

**Methods:**

* Alliance getPieceAlliance(): Returns the alliance of the piece.
* int getPiecePosition(): Returns the position of the piece.
* boolean isFirstMove(): Checks if it's the first move of the piece.
* PieceType getPieceType(): Returns the type of the piece.
* void setPiecePosition(int piecePosition): Sets the position of the piece.
* void setFirstMove(boolean isFirstMove): Sets the first move status.
* abstract Collection<Move> calculateLegalMoves(Board board): Abstract method to calculate legal moves.
* abstract Piece movePiece(Move move): Abstract method to execute a move and return the new piece state.
* String toString(): Returns a string representation of the piece based on its type and alliance.

King Class (Sub Class):

* boolean isCastled(): Checks if the king has castled.
* void setCastled(boolean isCastled): Sets the castling status.
* boolean isKingSideCastleCapable(): Checks if the king can castle on the king's side.
* void setKingSideCastleCapable(boolean isKingSideCastleCapable): Sets the king side castle capability.
* boolean isQueenSideCastleCapable(): Checks if the king can castle on the queen's side.
* void setQueenSideCastleCapable(boolean isQueenSideCastleCapable): Sets the queen side castle capability.
* Collection<Move> calculateLegalMoves(Board board): Calculates legal moves for the king.
* King movePiece(Move move): Executes a move and returns the new king state.
* String toString(): Returns "K" for white king and "k" for black king.

Pawns, Knights, Rooks, Queens, Bishops:

**Attributes:**

* static int[] CANDIDATE\_MOVE\_COORDINATES / static int[] CANDIDATE\_MOVE\_VECTOR\_COORDINATES: Arrays representing possible move coordinates or vectors for the piece.

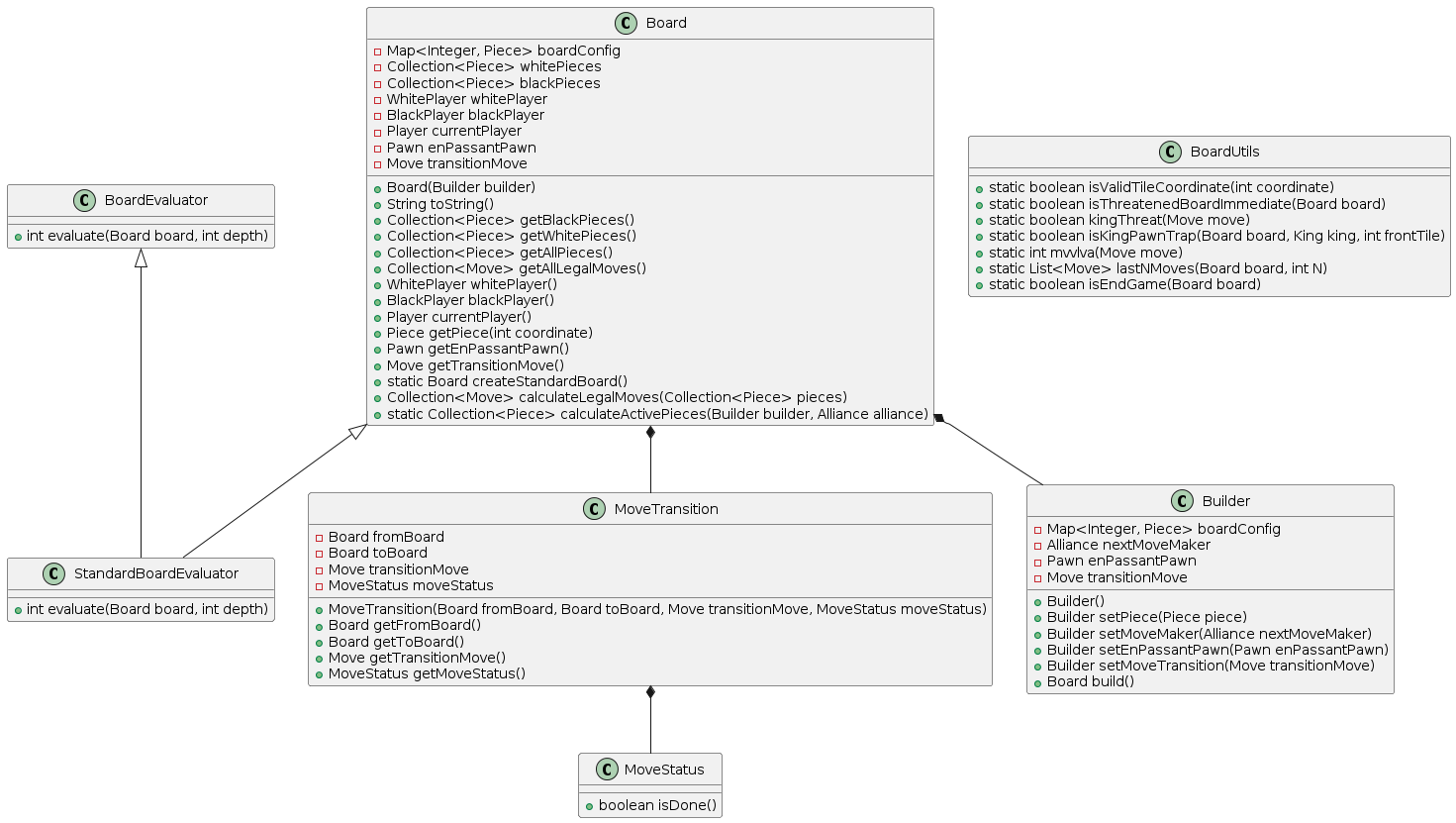
**Methods:**

* Collection<Move> calculateLegalMoves(Board board): Calculates legal moves for the piece.
* Piece movePiece(Move move): Executes a move and returns the new piece state.
* String toString(): Returns a string representation of the piece based on its type and alliance.

-> All subclasses of Piece inherit common attributes and methods from the Piece class.

Each subclass implements specific move calculation and representation methods.

Common methods like calculateLegalMoves(Board board), movePiece(Move move), and toString() are defined in each subclass to handle specific piece behaviors.



Board Diagrams:

#### Board Class

**Attributes:**

* Map<Integer, Piece> boardConfig: A map representing the configuration of pieces on the board.
* Collection<Piece> whitePieces: Collection of white pieces currently on the board.
* Collection<Piece> blackPieces: Collection of black pieces currently on the board.
* WhitePlayer whitePlayer: The player controlling the white pieces.
* BlackPlayer blackPlayer: The player controlling the black pieces.
* Player currentPlayer: The player who is currently making a move.
* Pawn enPassantPawn: The pawn eligible for en passant capture.
* Move transitionMove: The last move made on the board.

**Methods:**

* Board(Builder builder): Constructor to create a board using the Builder pattern.
* String toString(): Converts the board to a string representation.
* Collection<Piece> getBlackPieces(): Returns the collection of black pieces.
* Collection<Piece> getWhitePieces(): Returns the collection of white pieces.
* Collection<Piece> getAllPieces(): Returns the collection of all pieces on the board.
* Collection<Move> getAllLegalMoves(): Returns the collection of all legal moves.
* WhitePlayer whitePlayer(): Returns the white player.
* BlackPlayer blackPlayer(): Returns the black player.
* Player currentPlayer(): Returns the current player.
* Piece getPiece(int coordinate): Returns the piece at the specified coordinate.
* Pawn getEnPassantPawn(): Returns the en passant pawn.
* Move getTransitionMove(): Returns the last transition move.
* static Board createStandardBoard(): Creates a standard board configuration.
* Collection<Move> calculateLegalMoves(Collection<Piece> pieces): Calculates the legal moves for the given pieces.
* static Collection<Piece> calculateActivePieces(Builder builder, Alliance alliance): Calculates the active pieces for the given alliance.

#### BoardUtils Class

**Methods:**

* static boolean isValidTileCoordinate(int coordinate): Checks if a tile coordinate is valid.
* static boolean isThreatenedBoardImmediate(Board board): Checks if the board is immediately threatened.
* static boolean kingThreat(Move move): Checks if a move threatens the king.
* static boolean isKingPawnTrap(Board board, King king, int frontTile): Checks if the king is trapped by a pawn.
* static int mvvlva(Move move): Returns the most valuable victim - least valuable attacker score for a move.
* static List<Move> lastNMoves(Board board, int N): Returns the last N moves made on the board.
* static boolean isEndGame(Board board): Checks if the game is in an endgame state.

#### BoardEvaluator Interface

**Methods:**

* int evaluate(Board board, int depth): Evaluates the board position.

#### StandardBoardEvaluator Class

**Methods:**

* int evaluate(Board board, int depth): Evaluates the board position.

#### MoveTransition Class

**Attributes:**

* Board fromBoard: The board before the move.
* Board toBoard: The board after the move.
* Move transitionMove: The move that caused the transition.
* MoveStatus moveStatus: The status of the move.

**Methods:**

* MoveTransition(Board fromBoard, Board toBoard, Move transitionMove, MoveStatus moveStatus): Constructor for the move transition.
* Board getFromBoard(): Returns the board before the move.
* Board getToBoard(): Returns the board after the move.
* Move getTransitionMove(): Returns the transition move.
* MoveStatus getMoveStatus(): Returns the move status.

#### MoveStatus Class

**Attributes:**

* boolean isDone(): Checks if the move is done.

#### Builder Class

**Attributes:**

* Map<Integer, Piece> boardConfig: The board configuration.
* Alliance nextMoveMaker: The alliance of the next move maker.
* Pawn enPassantPawn: The en passant pawn.
* Move transitionMove: The transition move.

**Methods:**

* Builder(): Constructor for the builder.
* Builder setPiece(Piece piece): Sets a piece on the board.
* Builder setMoveMaker(Alliance nextMoveMaker): Sets the next move maker.
* Builder setEnPassantPawn(Pawn enPassantPawn): Sets the en passant pawn.
* Builder setMoveTransition(Move transitionMove): Sets the move transition.
* Board build(): Builds the board.

Similarities:

**calculateLegalMoves(Board board)**, **movePiece(Move move)**, and **toString()** methods are common across all Piece subclasses (Pawn, Queen, Rook, Knight, Bishop).

**CANDIDATE\_MOVE\_COORDINATES** or **CANDIDATE\_MOVE\_VECTOR\_COORDINATES** are static arrays representing possible move coordinates or vectors for the respective piece types.

# IV. GIT EXPLANATION

### Version Control

GitHub was used for source code management. The repository includes:

* **Branches**: Separate branches for different features and bug fixes.
* **Commits**: Regular commits with descriptive messages.
* **Pull Requests**: Used for code reviews and merging changes into the main branch.

### Collaboration

Multiple contributors worked on different aspects of the project, using Git for version control to ensure seamless collaboration and integration of changes.

# V. LECTURE APPLICATIONS

This project applies various concepts learned in lectures, including:

* **Data Structures**: Arrays, linked lists, and hash maps to manage game state and piece positions.
* **Algorithms**: Minimax and alpha-beta pruning for AI decision-making.
* **Object-Oriented Programming (OOP)**: Design patterns, encapsulation, inheritance, and polymorphism to create a modular and maintainable codebase.

### 

### Minimax Algorithm

#### Summary: The minimax algorithm is used for decision-making and game theory in our chess AI.

#### Objective: Minimax seeks to minimize the possible loss for a worst-case scenario. When dealing with gains, it seeks to maximize the minimum gain.

#### Implementation: The AI evaluates board positions up to a certain depth and chooses the move that maximizes its chances of winning while minimizing the opponent's chances. Minimax works by simulating all possible moves, assuming that the opponent will always make the best possible counter-move. This involves generating a game tree where each node represents a board state, and the algorithm recursively evaluates the possible moves to find the optimal move for the current player. The evaluation function assigns a numerical value to each board state, which reflects the advantage of the current player. By backtracking through the tree, minimax selects the move that leads to the most favorable outcome.

#### Why Minimax Works: Minimax works because it is based on the principle of zero-sum games, where one player's gain is another player's loss. By recursively evaluating the possible moves and their outcomes, the algorithm ensures that it considers both the player's and the opponent's best possible moves. This thorough analysis helps the AI make informed decisions that maximize its chances of winning while minimizing potential losses.

#### Space Complexity: The space complexity of the minimax algorithm is O(d), where d is the depth of the tree. This is because it requires storing the call stack during recursion.

#### Alpha-Beta Pruning

#### Summary: Alpha-beta pruning is an optimization technique for the minimax algorithm.

#### Objective: Reduce the number of nodes evaluated in the minimax algorithm tree.

#### Implementation: Alpha-beta pruning works by eliminating branches in the tree that do not need to be explored because they cannot influence the final decision. It keeps track of two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of, respectively. As the tree is traversed, branches that cannot possibly affect the final decision are pruned away, thus improving the efficiency of the minimax algorithm.

### Space Complexity: Similar to the minimax algorithm, the space complexity of alpha-beta pruning is O(d), where d is the depth of the tree. This is because the algorithm also relies on recursion and thus requires storing the call stack.

### Iterative Deepening

Iterative deepening is a technique used to enhance the performance of alpha-beta pruning. It involves progressively deepening the search one level at a time and storing the best move found at each level. This allows the algorithm to use the best move from the previous iteration as the first move to explore in the next iteration, which helps alpha-beta pruning to prune more effectively.

#### Best Case: The best case for alpha-beta pruning occurs when the algorithm evaluates the best moves first. This allows the pruning to happen at the earliest possible stage, reducing the number of nodes to be evaluated significantly. Iterative deepening assists in achieving this best-case scenario by using the move ordering from the previous search depth.

### Time Complexity Analysis

#### Summary: Analysis of the time complexity of key algorithms used in the game.

#### Minimax Algorithm: The time complexity is O(bd), where b is the branching factor (number of legal moves at a given point) and d is the depth of the tree.

#### Alpha-Beta Pruning: This optimization reduces the time complexity to O(bd/2) in the best case, effectively allowing deeper searches within the same time constraints.

#### Best Case: The best case for move ordering occurs when the move ordering heuristic is perfect, and the best moves are always evaluated first, maximizing the efficiency of alpha-beta pruning.

#### 

#### 

#### VI. LIMITATION

### Challenges

* **Complex Algorithms**: Handling complex game rules and scenarios using advanced algorithms.
* **Performance Optimization**: Ensuring the game runs smoothly by optimizing data structures and algorithms.
* **User Experience Issues**: Improving usability based on feedback.

### Limitations

* The AI might not cover all possible moves in deeper game states due to limited computational resources.
* The current version does not support online multiplayer mode.
* Customization options for the board and pieces are limited.

**VII. CONCLUSION**

In conclusion, this project has been a rewarding experience in creating a Chess game using Data Structures and Algorithms (DSA). We focused on building an efficient and intelligent system by leveraging key DSA principles such as minimax algorithm, alpha-beta pruning, and various heuristics for move evaluation.

The minimax algorithm was implemented to enable the AI to make strategic decisions by exploring potential future moves and evaluating their outcomes. Although computationally intensive, the algorithm ensures the AI plays optimally within its depth limitations. By incorporating alpha-beta pruning, we optimized the minimax algorithm, significantly reducing the number of nodes that needed evaluation and improving performance.

To manage the game state and logic efficiently, we utilized several data structures, including arrays and linked lists, to represent the board and pieces. Each chess piece, from the Pawn to the Queen, had its movement and interaction rules encapsulated in its respective class, allowing for clean and maintainable code.

Git played a crucial role in our collaboration, enabling us to manage code changes efficiently and work together seamlessly. Each team member contributed to different aspects of the project, from implementing the minimax algorithm to designing the graphical user interface.

Overall, this project has been an excellent learning experience, deepening our understanding of DSA concepts and enhancing our skills in algorithm design and software development. The successful integration of theoretical knowledge with practical implementation showcases our ability to create a complex and intelligent system.

**VIII. TASKING**

Phúc:

* StockAlphaBeta
* IterativeDeepening

Quốc:

* AlphaBetaWithMoveOrdering
* StandardBoardEvaluator
* LegalMove via BitBoards.

Liêm:

* Minimax
* MoveOrdering
* RookStructureAnalyzer

Hân:

* KingSafetyAnalyzer
* PawnStructureAnalyzer

**IX. REFERENCES**

Minimax:

[Algorithms Explained – minimax and alpha-beta pruning](https://www.youtube.com/watch?v=l-hh51ncgDI&t=1s)

Iterative Deepening:

[Iterative Deepening Search | IDS Search | DFS Algorithm in Artificial Intelligence by Mahesh Huddar](https://www.youtube.com/watch?v=BK8cEWKHCkY)

MoveOrdering: [This is why MOVE ORDERING is so critical in ALPHA-BETA search | Chess Programming](https://www.youtube.com/watch?v=JZOo1GuwVro)

Chebyshev Distance for KingSafety: <https://www.chessprogramming.org/King_Safety>