Class Structure in the Dark Blue Chess Engine
By Ryan King

Piece.java:

This file contains the basic, abstract definition of a chess piece. It contains a color enum, row integer, column integer, and an `ArrayList` of current legal moves. All fields are final and immutable except for the `ArrayList`, whose space is final when initialized but can have elements added, removed, cleared, and so on at any time.

Methods in this file:

Concrete:

GetColor() - Accessor to obtain the piece's color enum.

GetCurrentRow() - Accessor to obtain the piece's current row.

GetCurrentColumn() - Accessor to obtain the piece's current column.

CanMove() - Boolean flag indicating if this piece has at least one legal move.

HasMoved() - Boolean flag indicating if this piece has been moved before.

HowManyMoves() - Accessor that returns the number of moves this piece has made.

IsWhite() - Boolean flag returning if this piece's color enum is set to white.

IsBlack() - Boolean flag returning if this piece's color enum is set to black.

IsAlly() - Boolean flag returning if another piece is the same color as this one.

IsEnemy() - Boolean flag returning if another piece is the opposite color as this one.

IsSameType() - Boolean flag indicating if another piece is the same type as this one.

Equals() - Determines if this piece and another have the same color and type.

toString() - returns a String representation of this piece by displaying the tile it is resting on, followed by the color and type. A white pawn resting on e4 will read: “e4 white pawn”.

Abstract:

GetPieceType() - Abstract accessor to return the piece’s type as an enum.

GetIcon() - Abstract accessor to return the piece’s icon in English standard algebraic notation (P for pawns despite the fact that this character is not used in algebraic notation, R for rooks, N for knights, B for bishops, Q for queens, and K for kings). Returns a capital letter for white and a lowercase letter for black.

GetBoardIcon() - Abstract accessor to return the piece’s board icon, which is a Unicode character that shows the actual pictorial representation of the piece. This varies between the piece’s type and color.

IsPawn() - Abstract Boolean field to determine if this piece is a pawn.

IsRook() - Abstract Boolean field to determine if this piece is a rook.

IsKnight() - Abstract Boolean field to determine if this piece is a knight.

IsBishop() - Abstract Boolean field to determine if this piece is a bishop.

IsQueen() - Abstract Boolean field to determine if this piece is a queen.

IsKing() - Abstract Boolean field to determine if this piece is a king.

AddCurrentLegalMoves() - Abstract mutator that adds all the piece’s legal moves for the current turn.

Knight.java:

This class contains the implementation of a knight chess piece. It contains all fields and methods from the Piece class.

Methods in this file:

AddCurrentLegalMoves() - Concrete definition of the Piece class method that adds up to eight moves which a knight can make on the current turn.

GetPieceType() - Always returns `PieceType.KNIGHT`.

GetIcon() - Always returns ‘N’ if white or ‘n’ if black.

GetBoardIcon() - Always returns ‘♘’ if white or ‘♞’ if black.

IsPawn() - Always returns false.

IsRook() - Always returns false.

IsKnight() - Always returns true.

IsBishop() - Always returns false.

IsQueen() - Always returns false.

IsKing() - Always returns false.

King.java:

This class contains the implementation of a king piece. It contains all fields and methods from the `Piece` class, in addition to an `ArrayList` of current castling moves and two Boolean fields which can tell if it can perform a kingside or queenside castle. More methods exist for the determination of castling rights.

Methods in this file:

AddCurrentLegalMoves() - Concrete definition of the Piece class method that adds up to eight moves which a king can make on the current turn.

GetPieceType() - Always returns `PieceType.KING`.

GetIcon() - Always returns ‘K’ if white or ‘k’ if black.

GetBoardIcon() - Always returns ‘♔’ if white or ‘♚’ if black.

IsPawn() - Always returns false.

IsRook() - Always returns false.

IsKnight() - Always returns false.

IsBishop() - Always returns false.

IsQueen() - Always returns false.

IsKing() - Always returns true.

CanKingsideCastle() - Accessor that tells if this king can perform a kingside castle.

CanQueensideCastle() - Accessor that tells if this king can perform a queenside castle.

RemoveCastlingMoves() - Removes the king’s castling moves from both his castling move ArrayList and his legal move ArrayList.

IsInOriginalSpot() - Boolean flag determining if the king is in his original spot.

GetCurrentCastlingMoves() - Accessor returning the ArrayList of the king’s castling moves.

CanKingsideCastleOnThisTurn() - Determines if this king can perform a kingside castle on this turn given the current state of the board.

CanQueensideCastleOnThisTurn() - Determines if this king can perform a queenside castle on this turn given the current state of the board.

Rook.java:

This class contains the implementation of a rook chess piece. It contains all fields and methods from the Piece class as well as four ArrayLists for legal up, down, left, and right moves.

Methods in this file:

AddCurrentLegalMoves() - Concrete definition of the Piece class method that adds all horizontal and vertical moves this piece can make on the current turn. Each direction is added one at a time.

GetPieceType() - Always returns `PieceType.ROOK`.

GetIcon() - Always returns ‘R’ if white or ‘r’ if black.

GetBoardIcon() - Always returns ‘♖’ if white or ‘♜’ if black.

IsPawn() - Always returns false.

IsRook() - Always returns true.

IsKnight() - Always returns false.

IsBishop() - Always returns false.

IsQueen() - Always returns false.

IsKing() - Always returns false.

Bishop.java:

This class contains the implementation of a bishop chess piece. It contains all fields and methods from the Piece class as well as four ArrayLists for legal up and right, down and right, up and left, and down and left moves.

Methods in this file:

AddCurrentLegalMoves() - Concrete definition of the Piece class method that adds all diagonal moves this piece can make on the current turn. Each direction is added one at a time.

GetPieceType() - Always returns `PieceType.BISHOP`.

GetIcon() - Always returns ‘B’ if white or ‘b’ if black.

GetBoardIcon() - Always returns ‘♗’ if white or ‘♝’ if black.

IsPawn() - Always returns false.

IsRook() - Always returns false.

IsKnight() - Always returns false.

IsBishop() - Always returns true.

IsQueen() - Always returns false.

IsKing() - Always returns false.

Queen.java:

This class contains the implementation of a queen chess piece. It contains all fields and methods from the Piece class as well as eight ArrayLists for legal up and right, down and right, up and left, down and left, up, down, left, and right moves.

Methods in this file:

AddCurrentLegalMoves() - Concrete definition of the Piece class method that adds all horizontal and diagonal moves this piece can make on the current turn. Each direction is added one at a time.

GetPieceType() - Always returns `PieceType.QUEEN`.

GetIcon() - Always returns ‘Q’ if white or ‘q’ if black.

GetBoardIcon() - Always returns ‘♕’ if white or ‘♛’ if black.

IsPawn() - Always returns false.

IsRook() - Always returns false.

IsKnight() - Always returns false.

IsBishop() - Always returns false.

IsQueen() - Always returns true.

IsKing() - Always returns false.

Pawn.java:

This class contains the implementation of a pawn chess piece. It contains all fields and methods from the Piece class as well as three ArrayLists for legal regular, attacking, and en passant moves.

Methods in this file:

AddCurrentLegalMoves() - Concrete definition of the Piece class method that adds all regular, attacking, and en passant moves this piece can make on the current turn. Each type is added one at a time.

AddCurrentRegularMoves() - Adds this pawn’s regular moves. Adds two if the pawn has not moved and there is no piece in front of it for at least two squares, or one if another piece is blocking the second square or the pawn has already moved. Will not add anything if the pawn is blocked.

AddCurrentAttackingMoves() - Adds up to two attacking moves by checking either diagonal and seeing if they are valid and have an enemy piece on it that is not the opponent's king. Will not do anything if both diagonals are empty, have friendly pieces on them, or are invalid.

AddCurrentEnPassantMoves() - Adds a special pawn capture if this piece is on its fifth rank and the piece sitting directly left or right of it is an enemy pawn that just moved two tiles on its first move. This will not apply if the enemy pawn has sat next to this pawn for more than one turn.

GetPieceType() - Always returns `PieceType.PAWN`.

GetIcon() - Always returns 'P' if white or 'p' if black.

GetBoardIcon() - Always returns ' ♙ ' if white or ' ♟ ' if black.

IsPawn() - Always returns true.

IsRook() - Always returns false.

IsKnight() - Always returns false.

IsBishop() - Always returns false.

IsQueen() - Always returns false.

IsKing() - Always returns false.

PieceType.java:

This enumeration represents the type of a piece.

Methods in this file:

toString() - Returns a string representation of the piece type.

ToCharacter() - Returns a character representing the piece's type in English algebraic notation. All pieces except the knight return the piece's

first letter. The knight returns the second letter because ‘K’ is already being used by the king.

GetLimit() - Returns the maximum number of times this piece can occur on a FEN string. This is only done from the perspective of one color. A player can have 10 rooks, bishop, or knights or 9 queens if every pawn gets promoted, up to 8 pawns, and exactly 1 king.

ChessColor.java:

This enumeration contains an easy way to refer to the color of a piece or a tile without confusing it with `java.awt.Color`.

Methods in this file:

toString() - Returns “WHITE” or “BLACK”.

Tile.java:

This class contains a representation of a chessboard tile. It contains a row integer, a column integer, a `ChessColor` enum color, and a Piece object for whatever piece may be occupying it. All tile objects have final fields and are immutable.

Methods in this file:

GetColor() - Accessor to return the color of this tile, either `ChessColor.WHITE` or `ChessColor.BLACK`.

GetPiece() - Accessor to return the piece occupying the tile, if any. If no piece is occupying it, this returns `null`.

GetRow() - Accessor to return the row of this tile.

GetColumn() - Accessor to return the column of this tile.

IsOccupied() - Returns true if the `Piece` object is not `null` and false otherwise.

IsEmpty() - Returns true if the `Piece` object is `null` and false otherwise.

toString() - Returns this tile in algebraic notation, e.g. “e2”.

Board.java:

This class contains an 8 by 8 array of sixty-four Tile objects. It also has a flag representing which side can move as well as an en passant tile.

Methods in this file:

GetEmptyBoard() - Returns an empty Board object.

GetStartingPosition() - Returns a Board object with all thirty-two pieces placed in the usual starting position of chess. Turn flag is set to white.

GetStalemateTest() - Returns a Board object in a test configuration where white’s next move will place black into stalemate.

GetCheckmateTest() - Returns a Board object in a test configuration where white can mate in one move.

GetCastlingTest() - Returns a Board object in a test configuration where both kings and their rooks may castle.

GetPromotionTest() - Returns a Board object in a test configuration where both sides can promote at least one pawn.

GetEnPassantTest() - Returns a Board object in a test configuration where both sides can perform an en passant capture with their pawns.

GetWhiteBoard() - Returns a visual representation of the board as a string with white oriented toward the bottom.

GetBlackBoard() - Returns a visual representation of the board as a string with black oriented toward the bottom.

GetTile() - Returns a board tile by row and column.

GetKing() - Returns a player’s king by color.

GetBoard() - Returns the array object.

PieceCount() - Returns the number of pieces on the board.

GetEnPassantTile() - Returns the en passant tile if any, otherwise returns `null`.

GetCurrentBoard() - Returns a deep copy of the current board.

GetDeepCopy() - Returns a deep copy of the board. Invoked in lieu of a copy constructor because all constructors are private.

AdjustCastlingPrivileges() - Compares current king castling privileges with those that have recently been determined and adjusts the privileges if needed.

Move() - Performs a `RegularMove` on this board.

Attack() - Performs an `AttackingMove` on this board.

Castle() - Performs a `CastlingMove` on this board.

EnPassant() - Performs an `EnPassantMove` on this board.

Promote() - Performs a promotion on this board.
toString() - Returns a FEN configuration of this board with the en passant tile and castling rights listed but does not include halfmove and fullmove clocks.

SerializeBoardContents() - Parses the board contents into an FEN string.

SerializeTurn() - Returns a lowercase “b” or “w” to indicate whose turn it is.

SerializeCastlingRights() - Returns some combination of “KQkq” in that order or “-” if no castling rights exist.

SerializeEnPassantTile() - Returns the algebraic notation of the en passant tile or “-” if none exists.

BoardBuilder (internal class inside Board.java):

This is a builder class located inside the `Board` class. It contains its own array of tiles and a turn flag.

Methods in this class:

InitializeEmptyBoard() - Initializes an empty `Board` object.

GetBuilderBoard() - Returns the builder's own board array.

GetTile() - Returns one of the builder's tiles by row and column.

SetPiece() - Sets the given piece on the given tile.

RemovePiece() - Removes the piece on the given tile.

SetWhoseTurn() - Sets the turn flag.

SetTile() - Sets a tile object at the given coordinates.

GetTile() - Returns a tile at the given row and column.

Build() - Returns a new `Board` object using the configuration of this builder.

Move.java:

This class is an abstract representation of a move to be made on the chessboard. It contains a piece object representing the piece to be moved, two integers representing the piece's current row and column, two more integers representing the piece's desired row and column, two more integers representing the coordinates of the victim piece (if any), and the initial board on which this move will be made.

Methods in this file:

Concrete:

GetPiece() - Returns the piece that will be moving.

GetOldRow() - Returns the piece’s current row.

GetOldColumn() - Returns the piece’s current column.

GetNewRow() - Returns the destination row.

GetNewColumn() - Returns the destination column.

GetInitialBoard() - Returns the initial board on which this move will be made.

PlacesOpponentIntoCheck() - Boolean flag returning if the transitional board places the opponent into check.

PlacesOpponentIntoCheckmate() - Boolean flag returning if the transitional board places the opponent into checkmate.

Abstract:

IsRegular() - Abstract Boolean flag to return if this move is regular.

IsAttacking() - Abstract Boolean flag to return if this move is attacking.

IsCastling() - Abstract Boolean flag to return if this move is castling.

IsEnPassant() - Abstract Boolean flag to return if this move is en passant.

GetVictim() - Returns the victim of this move, if any; otherwise, returns `null`.

HasVictim() - Abstract Boolean flag to return if the victim of this move is not set to `null`.

GetTransitionalBoard() - Returns a version of the initial board field with this move made on it.

RegularMove.java:

This class contains a concrete implementation of a move made by any piece that does not capture another piece, except for castling.

Methods in this file:

toString() - Returns a string representation of this move. For example, moving a pawn from e2 to e4 returns “e4”. Moving a knight to d6 returns “Nd6”. The pawn icon is never returned.

GetVictim() - Always returns `null`, since this move is not supposed to have a victim.

GetMoveType() - Always returns `MoveType.REGULAR`.

GetTransitionalBoard() - Returns a version of the initial board field with this move made on it.

HasVictim() - Always returns false.

IsRegular() - Always returns true.

IsAttacking() - Always returns false.

IsCastling() - Always returns false.

IsEnPassant() - Always return false.

AttackingMove.java:

This class contains a concrete implementation of a move made by any piece that captures any other piece, except for en passant captures.

Methods in this file:

toString() - Returns a string representation of this move. For example, a pawn capturing from e2 to d3 returns “exd3”. Capturing a piece on d6 with a knight returns “Nxd6”. The pawn icon is never

returned, only the letter of tile from which it moved. Attacking moves made with any other type of piece will return that respective icon as a capital letter, regardless of the piece's color. An "x" precedes the destination tile in any case.

GetVictim() - Always returns a non-`null` piece of the opposite color located at the destination coordinates, since this move must have a victim.

GetMoveType() - Always returns `MoveType.ATTACKING`.

GetTransitionalBoard() - Returns a version of the initial board field with this move made on it.

HasVictim() - Always returns true.

IsRegular() - Always returns false.

IsAttacking() - Always returns true.

IsCastling() - Always returns false.

IsEnPassant() - Always return false.

CastlingMove.java:

This class contains a concrete implementation of a move made by the king that involves him moving two tiles to his left or right and having a rook on the side to which he moves jump to the adjacent tile the king passed over.

Methods in this file:

toString() - Returns a "0-0" for a kingside castle and "0-0-0" for a queenside castle.

GetVictim() - Always returns `null`, since this move is not supposed to have a victim.

GetMoveType() - Always returns `MoveType.CASTLING`.

GetTransitionalBoard() - Returns a version of the initial board field with this move made on it.

HasVictim() - Always returns false.

IsRegular() - Always returns false.

IsAttacking() - Always returns false.

IsCastling() - Always returns true.

IsEnPassant() - Always return false.

EnPassantMove.java:

This class contains a concrete implementation of a move made by a pawn on its fifth rank capturing a pawn of the opposite color that just moved two tiles on its first move as if it had only moved one tile. It requires a pawn victim to be passed into its constructor and contains two additional row and column integer fields for that pawn.

Methods in this file:

toString() - Returns a string representation of this move. For example, a pawn capturing *en passant* from e5 to d6 returns “exd6 e.p.”. The pawn icon is never returned, only the letter of tile from which it moved. The source column letter is followed by an “x” then the destination tile, then “e.p.”.

GetVictim() - Always returns a non-`null` pawn of the opposite color located at the captured pawn row and column, since this move must have a pawn victim.

GetMoveType() - Always returns `MoveType.EN_PASSANT`.

GetTransitionalBoard() - Returns a version of the initial board field with this move made on it.

HasVictim() - Always returns true.

IsRegular() - Always returns false.

IsAttacking() - Always returns false.

IsCastling() - Always returns false.

IsEnPassant() - Always return true.

MoveType.java:

This enum represents the type of move an object is. This could be regular, attacking, castling, or en passant.

Method in this file:

None

Player.java:

This class represents a player playing chess. It contains a list of active pieces, captured pieces, its color, and its king. It can be extended to either a human or computer object.

Methods in this file:

Refresh() - Clears and recalculates all active pieces and all current legal moves for each piece.

HowManyMoves() - Returns the total number of moves every piece has.

InitializePieces() - Clears and reinitializes all active pieces given the current state of the board argument.

AddActivePiece() - Adds a friendly piece to the active piece roster.

RemoveActivePiece() - Removes a captured piece from the active piece roster.

InitializeCurrentLegalMoves() - Clears and recalculates all legal moves as per the state of the board passed in as an argument.

GetColor() - Returns this player's color - black or white.

IsHuman() - Returns true if this player is a human, otherwise false.

IsComputer() - Returns true if this player is a computer, otherwise false.

IsWhite() - Returns true if the player's color is white, otherwise false.

IsBlack() - Returns true if the player's color is black, otherwise false.

GetKing() - Returns this player's king.

HasBareKing() - Returns true if the player only has a king in their active piece roster.

HasKingAndBishop() - Returns true if the player only has a king and a bishop in their active piece roster.

HasKingAndKnight() - Returns true if the player only has a king and a knight in their active piece roster.

IsInCheck() - Returns true if the player's king is not safe but the player has at least one legal move.

IsInStalemate() - Returns true if the player's king is safe but the player has no legal moves.

IsInCheckmate() - Returns true if the player's king is not safe and the player has no legal moves.

CanKingsideCastle() - Returns true if the player can perform a kingside castle.

CanQueensideCastle() - Returns true if the player can perform a queenside castle.

HasPotentialCastlingKing() - Returns true if the player has a king that has not moved, is not in check, and can slide two empty tiles to the left or right.

HasPotentialCastlingRook() - Returns true if a rook on one corner of the board has not moved and can castle with the king.

GetPromotedPawn() - Returns a pawn on its last rank ready to be promoted, otherwise `null`.

HasCastled() - Returns true if a player has castled and false otherwise.

GetPlayerType() - Returns the player's type: human or computer.

GetActivePieces() - Returns an `ArrayList` of the player's active pieces.

GetCapturedPieces() - Returns an `ArrayList` of the player's captured pieces.

UglyMoves() - Returns an unsorted `ArrayList` containing every move the player can make on the current turn. Will return an empty list if no moves exist.

RegularMoves() - Returns an `ArrayList` containing every regular move the player can make on the current turn. Will return an empty list if no such moves exist.

AttackingMoves() - Returns an `ArrayList` containing every attacking move the player can make on the current turn. Will return an empty list if no such moves exist.

CastlingMoves() - Returns an `ArrayList` containing every castling move the player can make on the current turn. Will return an empty list if no such moves exist.

EnPassantMoves() - Returns an `ArrayList` containing every en passant move the player can make on the current turn. Will return an empty list if no such moves exist.

CheckMoves() - Returns an `ArrayList` containing every move the player can make on the current turn that places the opponent into check. Will return an empty list if no such moves exist.

CheckmateMoves() - Returns an `ArrayList` containing every move the player can make on the current turn that places the opponent into checkmate. Will return an empty list if no such moves exist.

Human.java:

This class represents a human player (that is, controlled by the user). It contains all fields and non-abstract methods present in the `Player` class.

Methods in this file:

IsHuman() - Always returns true.

IsComputer() - Always returns false.

GetPlayerType() - Always returns `PlayerType.HUMAN`.

Computer.java:

This class represents a computer player (that is, controlled by the program). It contains all fields and non-abstract methods present in the Player class.

Methods in this file:

IsHuman() - Always returns false.

IsComputer() - Always returns true.

GetPlayerType() - Always returns `PlayerType.COMPUTER`.

PlayerType.java:

This enum represents a type of player: either human or computer.

Methods in this file:

None

DarkBlue.java:

This class contains many of the GUI components for the Dark Blue chess engine, as well as the main fields used for playing the game.

Methods in this file:

SetUpScrollPanes() - Sets the preferred sizes and viewports of both `JScrollPane`s so the move history panels will be dynamically scrollable.

GetDate() - Returns the current date and twenty-four-hour time as a string in the format YYYYMMDDHHMMSS. Used when a file is serialized. Leading zeros will be inserted for month, day, hour, minute, and second values less than ten.

HighlightLegalMoves() - Called when a piece has been selected to move. This will highlight the tile the piece is resting on and all legal move destination tiles in bright green.

UndoHighlighting() - Called either when the human deselects a piece or moves a selected piece. This reverts the selected piece's original tile and all of its legal move tiles back to their original colors.

AddComponentsToPane() - Adds all `JPanel`s to the Dark Blue content pane and sets their layouts.

CreateAndShowGUI() - Calls AddComponentsToPane(), packs the frame, makes it impossible to resize, and sets it visible to start the program.

GetHumanColor() - Returns the color the human selected at the beginning of the most recent game.

GetComputerColor() - Returns the color taken by the computer at the beginning of the most recent game.

GetEnPassantTile() - Returns a string representation of an en passant destination tile if one exists, otherwise returns `null`.

CheckForPromotions() - Sees if one player has any pawns that must be promoted. If there is a pawn that can be promoted, the player must choose a piece for it to become immediately. If none exist, this does absolutely nothing.

GetPromotedPawnIndex() - Finds the index in a player's `ArrayList` of active pieces where a promoted pawn is. Returns -1 if none exist.

GetPreviouslyMoved() - Returns the piece that was previously moved the prior turn.

GetMaxSearchDepth() - Returns the maximum search depth for the AI.

Serialize() - Returns a string representing the board configuration, next player, castling rights, en passant tile, halfmove clock, and fullmove clock. This is used when parsing a game into an FEN file.

Deserialize() - Allows the user to pick a file with a `JFileChooser` and parse the FEN string inside. Returns true if successful and false if it failed.

ReadFile() - Takes in a filename and attempts to read the file with an `InputStreamReader`.

WriteFile() - Takes in a filename and an FEN string and attempts to write a new file of that name with an `OutputStreamWriter`. Returns the filename.

AttemptToParseFile() - Attempts to validate and parse an FEN string, returning true on success or false if the file was invalid or gave an unplayable game.

CheckStatusAndResume() - Sees if the currently moving player is in check after beginning a game from a serialization string. If the player is in

check, their castling moves get removed. If the player is also human, a “Check!” dialog box is displayed on the screen to warn them.

ParseFEN() - Parses a serialization string row by row, determines the next player, sets the castling rights, en passant tile, halfmove clock, and fullmove clock then resumes the game. Throws an exception if it encounters any unrecoverable problems.

ParseTurn() - Takes a letter “w” or “b” and turns it into the appropriate `ChessColor` to use as the turn flag. Throws an exception if anything goes wrong.

ParseCastlingRights() - Takes the string of castling rights and assigns all four Boolean variables individually depending on the configuration of the string.

ParseEnPassantTile() - Takes a string with an algebraic tile and plugs it in as the en passant tile for the given turn.

ParseMoveClocks() - Parses the halfmove and fullmove clocks. Throws an exception if at least one clock is not a valid integer.

SetUpTextAreas() - Adds ellipses (“...”) to the appropriate text areas when a game is being resumed from a file or custom FEN string.

EvaluatePreviouslyMoved() - Makes a deep copy of the piece that just moved.

SpawnThinkingDialog() - Spawns a dialog box with no buttons that says “Thinking…”. This is used both when the computer makes its move and when the human asks for help.

ComputerPlay() - Enables the computer to play by using a minimax algorithm. Since this can run a long time, it gets switched over to a worker thread by a `SwingWorker`. A “Thinking…” dialog box shows up to notify the human player. The EDT waits for the worker thread to return the move and then assigns the mover, victim, source tile, and destination tile fields. The dialog box is then destroyed. The move is then sent to a string and added into the proper window. The game state is evaluated. If the game

state is either check or normal, the game proceeds to modify the halfmove counter and then the fullmove counter if the mover is black. The observer gets called.

AssignMovingFields() - Assigns the source tile and destination tile when the computer is playing. Removes a victim from the opponent and adds it to the computer. Refreshes the computer's captured piece panel.

AppendMove() - Appends a move string to the proper text area.

UpdateMoveClocks() - Increments the halfmove clock if there was no capture or pawn movement or else resets it to zero. Increments the fullmove clock if the last player to move was black.

ResetMoveFields() - Resets the source tile, destination tile, and move objects to `null`.

AssignBoard() - Serialize the board to get its configuration and add it to the hash. Make the next move on the board and reassign it to the `GUIBoard` for repainting.

AssignEnPassantTile() - Checks to see if the board's en passant tile field is initialized and then assigns the engine's en passant tile string to the tile's algebraic notation string. If this field is `null`, the string also becomes `null`.

EnablesEnPassantMove() - Boolean flag that checks to see if the next move is a pawn moving two tiles forward as its first move and thus would enable an en passant move, even if no pawn can take advantage of it.

ChooseColor() - Pops up a dialog box asking which color the human player would like to choose. This box will keep reopening until the human chooses "White" or "Black".

InitializePlayers() - Instantiates each player object as either a concrete human or computer, depending upon what color was chosen in ChooseColor(). This also initializes each player's active pieces and legal moves.

RefreshPlayers() - Clears and reinitializes the pieces and legal moves of both players.

GetOriginalRow() - Returns the source row of the previously moved piece.

GetOriginalColumn() - Returns the source column of the previously moved piece.

GetInstance() - Returns a single instance of the Dark Blue chess engine if none exists, or returns the instance if it does exist.

HasInstance() - Boolean flag that returns if there is a copy of the Dark Blue engine in existence.

main() - Starts the Dark Blue chess engine.

ScrollTest() - Tests the scroll functionality by placing the text of the Gettysburg Address into the box each time a button is clicked.

TransitionBoardTest() - Testing the functionality of the `Move.GetTransitionalBoard()` method.

UpdatePanel() - Updates the player's newly captured piece, removes it from the opponent's active piece roster, and inserts a new piece image into a `CapturedPiecePanel`.

AdjustCapturedPieces() - Removes captured pieces from the captured piece panels and places them into the player's roster when a move is undone.

NumberOfNewlines() - Counts the number of "\n" characters found in a move history string. Useful for removing the last part of a move when undone.

RemoveCopy() - Removes a copy of the given FEN string from the hash.

AdjustMoveHistoryPanels() - Removes the most recent move from both move history panels and restores any pieces that were captured as part of said moves.

Internal classes in this file:

GUITile:
This class contains definitions for a visual representation of a tile. It contains a `Tile` object and a `JLabel` that may contain a picture. It can either be colored “black” (light reddish brown) or “white” (light apricot brown). It can also have one of twelve different images of chess pieces if its tile has one. It also contains a `java.awt.Color` for the background. All fields except the `JLabel` are immutable.

Methods in this class:

SetPiece() - Reads the image of a chess piece onto the tile.

DrawTile() - Calls the SetPiece() method and repaints the tile.

LightUp() - Turns the tile’s color to bright green.

Revert() - Turns the tile’s color back to its original color.

mouseClicked() - Responds to a mouse click. See Respond() for more information. Overridden from `MouseAdapter`.

Respond() - Processes mouse events, finds desired pieces to move, and makes moves. Sets the source and destination tile fields and finds pieces on them. Checks to see if moves are legal before making them. Alerts the user of any illegal action with a `JOptionPane` dialog box.

ShowSourceErrorMessage() - Displays an error message if the user either selects an empty tile to move, selects a piece that is not his/her color, or selects one of his/her pieces that has no legal moves.

ResetHumanFields() - Clears the source tile and destination tile fields to `null`. This is called when the user deselects a piece that has already been selected.

IsMoveDone() - Checks to see if all the fields that keep track of the source tile, destination tile, mover, victim, etc. are initialized and that the move has been executed.

PlayThrough() - This method turns off the “should highlight legal moves” flag, gets the victim and next move, makes the move on the board, makes a copy of the moved piece, and assigns the en passant tile. It then modifies both move clocks if need be and updates the human’s captured piece panel.

GetTile() - Returns the `Tile` object.

GetRow() - Returns the tile’s row.

GetColumn() - Returns the tile’s column.

toString() - Returns this tile as a string in algebraic notation using the tile’s own toString() method.

UpdateCapturedPiecePanel() - Refreshes the current player’s captured piece panel.

GUIBoard:
This class represents a grid of `GUITile`s and functions the way a chessboard would. It contains a 10-by-10 array of `GUITile`s and a `Board` object. The two extra rows and columns on the edges are saved for adding letters and numbers for easier use of notation. This varies depending on which side the player chooses, so there are two different methods that handle each type of drawing.

Methods in this class:

DrawBoard() - Draws the board by instantiating and repainting every tile. This also sets up notation on the sides of the board and makes changes in regards to highlighting or undoing the highlighting of legal moves if a piece has been selected or deselected.

BuildWhiteBoard() - This builds a `GUIBoard` with all white pieces oriented at the bottom and all column letters are in alphabetical order from left to right. All row numbers placed on the outside are ascending when going from bottom to top. This shows a direct representation of how the `Board` class looks in memory.

AddWhiteBorderTile() - This adds a reddish brown “border tile” on the perimeter of the `GUIBoard` and gives it a letter or number for usage in algebraic notation. This sets all numbers ascending from bottom to top and all letters ascending from left to right, just as a real board would be if white were viewing it in real life. Corner tiles have nothing on them.

BuildBlackBoard() - This builds a `GUIBoard` with all black pieces oriented at the bottom and all column letters are in alphabetical order from right to left. All row numbers placed on the outside are descending when going from bottom to top. This shows an upside-down representation of how the `Board` class looks in memory.

AddBlackBorderTile() - This adds a reddish brown “border tile” on the perimeter of the `GUIBoard` and gives it a letter or number for usage in algebraic notation. This sets all numbers ascending from top to bottom and all letters ascending from right to left, just as a real board would be if black were viewing it in real life. Corner tiles have nothing on them.

SetBoard() - Mutator to set the `Board` object. Usually this is followed by a repaint.

PieceCount() - Convenient way of calling the PieceCount() method in the Board class. See the original PieceCount() method in the `Board` class for more details.

GetBoard() - Accessor to return the model board.

WhoseTurnIsIt() - Convenient way of calling the model board’s WhoseTurnIsIt() method. See the original method in the `Board` class for more information.

GetTile() - Returns a tile from the GUIBoard. Unlike its cousin in the Board class, the coordinates that are considered valid in this accessor are [0, 9] as opposed to [0, 7] in the model.

DarkBlueMenuBar:
This is my custom class for the `JMenuBar`. It contains a File tab that has options for starting a new game, loading a pre-existing game either from a file or from a custom string, an Undo button in case the human player thinks they made a mistake, a Save button to save the game as a FEN file for later use, and a Quit button if the user wants to end the program. It also contains a Help tab that shows instructions on how to use this engine, the official FIDE laws of chess, and a help function to help the player move. The Undo, Save, and Help Me Move buttons are disabled by default if no game is in progress.

Methods in this class:

AddMnemonics() - Adds hotkeys to each option for ease of use.

AddActionListeners() - Adds the `ActionListener` interface to each button.

AddItemsToMenu() - Adds every item to the menu.

actionPerformed() - Overridden from `ActionListener`. Called when a button is clicked.

NewGameClicked() - Starts a new game, clears all fields, allows the human player to choose his/her color, and calls the observer to watch for game-ending conditions. This determines who moves first and which way the board will be oriented. Called when the “New Game” option is clicked.

FromFileClicked() - Opens a `JFileChooser` that allows the player to navigate to a custom FEN file anywhere on the local machine. If the file is incorrectly parsed, the user will see a `JOptionPane` error message saying the file could not be parsed. If successful, it allows the player to choose a color and the game resumes as normal. Called when the “From File” button is clicked.

SaveClicked() - Saves the current game into a FEN file with the words “Dark Blue” followed by the current timestamp. This goes into the directory “*eclipse-workspace*/DarkBlue/DarkBlue/src/com/DarkBlue/Serial”, where “*eclipse-workspace*” contains the absolute path to the user’s current Eclipse workspace. The board is cleared and the user is notified of the filename chosen by the program. Called when the “Save” button is clicked.

CustomFENClicked() - Allows the user to enter a string of text for any FEN file s/he wants. If it is incorrectly parsed, the user will see a `JOptionPane` error message saying the string could not be parsed. If successful, the player must choose his/her color, then the observer is called to check for game-ending conditions or to allow the computer to move. Play continues as normal. Called when the “From Custom FEN” button is clicked.

QuitClicked() - Allows the user to quit the program. A warning message with a yes or no option will pop up, asking the user if s/he really wants to quit. If the answer is yes, the program will terminate and say the other side won by resignation if a game is in progress. Called when the “Quit” button is clicked.

RulesOrInstructionsClicked() - Opens up a local PDF of either the official FIDE laws of chess or the chess engine instruction manual on the user’s browser. Called when the “Rules of Chess” or “Instructions” buttons are clicked.

HelpMeMoveClicked() - Calls the computer’s minimax algorithm to compute a potential move for the human player. The result is displayed in a dialog box. Called when the “Help Me Move” button is clicked.

UndoClicked() - Undoes the previous two halfmoves and removes their boards from the hash. Does not work when there have only been one or two halfmoves at the current state of the game. Called when the “Undo” button is clicked.

EnableSave() - Enables the “Save” button.

EnableUndo() - Enables the “Undo” button.

EnableHelpMeMove() - Enables the “Help Me Move” button.

DisableSave() - Disables the “Save” button.

DisableUndo() - Disables the “Undo” button.

DisableHelpMeMove() - Disables the “Help Me Move” button.

EnableLiveGameButtons() - Enables the above 3 buttons.

DisableLiveGameButtons() - Disables the above 3 buttons.

MoveTextArea:
This is my custom extension for the `JTextArea`. It stores moves that have been made in algebraic notation, with extra characters to denote check (“+”), checkmate (“#”), and a draw (“½-½”).

Methods in this class:

None

GameWatcher:
This class contains two methods that help determine the state of the game.

Methods in this class:

Observe() - This observer evaluates the game state and updates any changes in castling rights, en passant captures, and the like. It is responsible for stopping the game if checkmate, stalemate, or any other type of draw is reached, as well as notifying the human if s/he is in check. Internal methods described below take care of proper textbox notation and formatting. It swaps the player to look at game-ending conditions, but this is not done if the board is either a new game, a game that was just deserialized from a file, or a board state that was brought back as the result of an Undo. It also allows the computer player to make its move. This method is called after either player moves and after a game is deserialized from the hash or from a file or custom string.

IsGameOver() - This checks to see if the game has ended by looking at the game state. This method is also partially responsible for disabling

mouse functionality either at the end of a game or when the computer player is thinking in the Respond() method.

AnnounceCheckmate() - This announces through a `JOptionPane` error dialog that the side specified by the color passed in has won the game by checkmate and appends a “#”, newline, and “1-0” to white’s textbox or “#”, newline, and “0-1” to black’s textbox.

AnnounceStalemate() - This announces through a `JOptionPane` information dialog that the game has been drawn by stalemate and appends a newline and “½-½” to the mover’s textbox.

AnnounceInsufficientMaterial() - This announces through a `JOptionPane` information dialog that the game has been drawn by insufficient material and appends a newline and “½-½” to the mover’s textbox.

AnnounceFiftyMoveRule() - This announces through a `JOptionPane` information dialog that the game has been drawn by the fifty-move rule and appends a newline and “½-½” to the mover’s textbox.

AnnounceThreefoldRepetition() - This announces through a `JOptionPane` information dialog that the game has been drawn by threefold repetition and appends a newline and “½-½” to the mover’s textbox.

AnnounceCheck() - Removes the current player’s castling moves and appends a “+” and a newline to the player’s textbox. If the current player is human, this announces through a `JOptionPane` warning dialog that the current player is in check.

SwapPlayers() - Swaps the players at the end of one’s turn.

RecordBoard() - Places the board into the move history `HashMap` with a value of 1 or increments an existing entry by 1.

AdjustCastlingRights() - Makes immediate changes to the opponent’s castling rights if the most recent move or undo affected them.

AppendPromotion() - Appends a promotional string “=Q”, “=R”, “=B”, or “=N” to the player’s move if it was a promotion, depending on which piece the player chose.

CapturedPiecePanel.java:

This is another custom JPanel extension that shows the board icons of every piece a side has captured.

Methods in this file:

Clear() - Clears out the array of JLabels placed into the panel by setting all of them to empty strings.

Refresh() - Resets the layout of all captured pieces in the panel. Ordering is kept like this: P P P P P P P P R R N N B B Q. This is called once a move has been made or once a move has been undone.

Interfaces in the Dark Blue Chess Engine

Utilities.java:

This interface contains symbolic constants and other methods that are used frequently in other classes.

Methods in this file:

IsLegal() - Checks to see if a move is legal by seeing if the source and destination coordinates match up with a move in the piece's current legal move `ArrayList`.

EnlargeFont() - Enlarges the font on the given `JComponent` by 5.

Minimax.java:

This interface contains all methods necessary for the computer to calculate its move.

Methods in this file:

MinimaxRoot() - Called when the minimax algorithm starts parsing. It sets up the first loop which calls the Recurse() method recursively. It searches for the best move and returns it. If a move places the opponent into checkmate it will automatically return and not calculate any further.

Recurse() - Iteratively makes every move a player has on a cloned board and calls itself to minimize the result or maximize the result, depending upon which option came first. The base case (when the depth reaches 0) will calculate the value of the material on the board. It will also terminate early if the best result found is higher or lower than other results not seen yet (alpha-beta pruning).

Evaluate() - Calculates a decimal sum of the values of all the nonempty tiles on the board, one tile at a time.

GetPieceValue() - Returns the value of a piece. ±10 for a pawn, ±30 for a bishop or a knight, ±50 for a rook, ±90 for a queen, and ±900 for a king. 0 is returned if the piece is invalid.

GetAbsoluteValue() - Returns 10 for a pawn, 30 for a bishop or knight, 50 for a rook, 90 for a queen, or 900 for a king. 0 is returned if the piece is invalid.

Sort() - Orders moves in the following sequence:
Moves that place the opponent into checkmate
Moves that place the opponent into check
Moves that capture an enemy piece
Moves that castle the player's king
Moves that do not capture an enemy piece
Moves that capture an enemy pawn en passant

BoardUtilities.java:

This interface contains methods to help out with board-related business such as translation from model to algebraic coordinates, GUI coloring, and the like.

Methods in this file:

HasValidCoordinates() - Returns true if both coordinates given are between 0 and 7.

Reverse() - Returns the reverse of the given ChessColor. Black returns white and white returns black.

ToAlgebraicColumn() - Turns a model column 0-7 to a letter from a-h.

ToAlgebraicRow() - Turns a model row 0-7 to a number from 1-8.

ToAlgebraic() - Turns a row and column to an algebraic tile.

ToBoardRow() - Turns a spot in algebraic notation into its model row.

ToBoardColumn() - Turns a spot in algebraic notation into its model column.

HasValidValue() - Determines if a single model coordinate is valid, i.e. between 0 and 7.

IsValidTileLetter() - Determines if a letter is between a-h.

IsValidTileNumber() - Determines if a number is between 1-8.

IsValidTile() - Determines if a string is a valid tile in algebraic notation.

MoveEvaluation.java:

This interface contains methods to calculate legal moves and king safety. It also has two-dimensional minimax value arrays for both sides.

Methods in this file:

AddCurrentDirectionalMoves() - Adds all the legal moves in the current direction given the arguments for differences between tiles.

AddCurrentSpectrumMoves() - Adds all legal moves surrounding a piece in a spectrum where the legality of each move is independent from all others. Used exclusively for kings and knights.

AddCurrentRegularMoves() - Adds all regular moves for a pawn.

AddCurrentAttackingMoves() - Adds all attacking moves for a pawn.

AddCurrentEnPassantMoves() - Adds all en passant moves for a pawn.

IsKingMovesSafe() - Looks at all eight spots surrounding the king for enemy pieces.

IsKnightMovesSafe() - Looks at all tiles that are a knight's move away from the king for enemy knights.

IsDirectionSafe() - Checks one direction away from the king for safety. Since this is passed deltas, this can apply to any direction.

IsKingSafe() - Determines if the king is safe by checking all spots surrounding him, all knight moves surrounding him, and then all 8 directions away from him.

EvaluateDirection() - Evaluates an enemy piece found in IsDirectionSafe().

EvaluateHorizontal() - Sees if an enemy piece is a rook or queen.

EvaluateDiagonal() - Sees if an enemy piece is a bishop or queen.

IsDiagonal() - Returns true if two deltas return a diagonal result and false otherwise.

IsHorizontal() - Returns true if two deltas return a diagonal result and false otherwise.

CopyCurrentMoves() - Copies all moves from an `ArrayList`.

IsEnPassantMove() - Determines if the given move is an en passant move.

IsCastlingMove() - Determines if the given move is a castling move.

Factory.java:

This interface contains methods that make copies of moves and pieces.

Methods in this file:

PieceFactory() - Makes a copy of a piece with its move count initialized to zero.

MovedPieceFactory() - First overload. Makes a copy of a piece with its move count initialized to the move count of the argument piece plus one.

MovedPieceFactory() - Second overload. Makes a copy of a piece with its row and column changed to the given arguments and its move count incremented by one.

PromotedPieceFactory() - Returns a new queen, rook, bishop, or knight given a color, row, and column. Returns null on error.

MoveFactory() - Makes any type of move given the moving piece, destination row, destination column, victim piece, and initial board. Returns null if any arguments are invalid.

KingFactory() - Creates a new king with the specified castling rights. If both are false, it returns a king with one move made. If one or neither are false, it returns an unmoved king with the castling rights set as they were passed in.

RookFactory() - Creates a new rook to be used in conjunction with the given castling rights. If the rook is being placed onto a corner tile, position as a kingside or queenside rook is checked and the given castling rights are considered. If the right that pertains to it is false, the rook is constructed with one move made to stop the king from castling with it. If the right is true, a rook with no moves made is constructed instead. This does not apply if the rook is not placed onto a corner.

GameUtilities.java:

This interface contains methods that validate and parse strings from Forsyth-Edwards Notation (FEN) files and determine the state of a game to see if it is playable.

Methods in this file:

ParseRank() - Adds all pieces and empty tiles of one FEN rank onto a `BoardBuilder`.

TestParse() - Parses an FEN file in an isolated environment similar to the one found in the actual engine. Returns true on success and false on failure.

EvaluateGameState() - Checks the state of a game and encodes it into a `GameState` enumeration. This can either be `GameState.CHECKMATE`, `GameState.STALEMATE`, `GameState.DRAW`, `GameState.CHECK`, or `GameState.NORMAL`. The special `GameState.EMPTY` enum cannot be assigned here, as it is saved for when the engine starts up and when a game gets saved to a file or stopped without saving.

IsDrawByInsufficientMaterial() - Checks to see if a game is a draw due to insufficient material: Bare kings, bare king and bishop, bare king and knight, or kings and bishops on identical tile colors.

IsDrawByFiftyMoveRule() - Checks to see if a game is drawn by the fifty-move rule by seeing if the current number of halfmoves since the start of the game or the last capture or pawn movement (whichever occurred sooner) is at least fifty.

IsDrawByThreefoldRepetition() - Checks to see if a board configuration has occurred three times during the current game. These occurrences need not be consecutive. All board configurations are hashed as keys with the number of times they have occurred as a value of at least one. If a new configuration is found, it gets added into the hash with a value of 1. If an existing board is found, it gets its value incremented by 1. The `Iterator` interface is used to check through each board in the hash and see if its value equals three.

ExpandRank() - Takes a FEN row string and expands all numbers into that number of hyphen minuses (“-”). For example, 8 gets expanded to “--------”.

IsValidRank() - Determines if a FEN rank is valid by checking to see if it only contains numbers from 1 to 8 and either cases of letters P, R, N, B, Q, or K, and the sum of all numbers plus the sum of all pieces equals exactly 8.

IsValidFEN() - Determines if a FEN string is valid by checking the following:

- The string must contain 6 distinct parts separated by spaces (“ “)
    - The board configuration must contain exactly eight ranks separated by forward slashes (“/”)
    - Each player must have between 1 and 16 pieces which will make all of the following statements true:
        - The player has between 0 to 8 pawns
        - The player has between 0 to 10 rooks
        - The player has between 0 to 10 knights
        - The player has between 0 to 10 bishops
        - The player has between 0 to 9 queens
        - The player has exactly 1 king
        - All ranks must be valid as described in IsValidRank()
    - The next player must either be a lowercase “w” or a lowercase “b”
    - Castling rights must be some combination of the letters “KQkq” in that order, or simply “-” if none exist

    - The en passant tile must either be a two-character string with a lowercase letter from a-h followed by a 2 or a 7, or “-” if an en passant move does not exist
    - The halfmove clock must be a nonnegative integer
    - The fullmove clock must be a positive integer

HasValidClock() - Determines if a clock value is valid. A Boolean flag passed in as true checks to see if the number is the fullmove clock and is positive. Passing in the Boolean flag as false checks to see if the number is the halfmove clock and is zero or positive.

HasValidPieces() - Determines if both players have from 1 to 16 pieces of their respective colors and the correct number of each piece as explained below.

HasValidCastlingRights() - Determines if the castling rights string contains some unique combination of the letters “KQkq” in that order or is just “-”.

Pieces() - Returns a count of a piece of a specified type and color.

HasCorrectNumberOfPieces() - Determines, by color and type, if a player has the correct number of a certain type of piece on the FEN board. The piece character computed by the method must appear on the board a specific number of times:

- 0 to 8 pawns
- 0 to 9 queens
- 0 to 10 rooks, bishops, or knights
- Exactly 1 king

GUITest.java:

This is a stripped-down testing environment for testing out the `GUITile` and `GUIBoard` classes.

Methods in this file:

GUITileTest() - Generates a GUITile given the arguments the user provides.

GUIBoardTest() - Generates the starting position in chess with white oriented to the bottom if the user chooses white, or black oriented to the bottom if the user chooses black.

PromptPiece() - Asks the user for what kind of piece they want.

PromptTile() - Asks the user for the coordinates of the tile they want to display.

PromptColor() - Asks the user for the color of their piece they want to display.

GetPieceByLetter() - Returns a piece given the letter the user gave.

GUITile.java:

This is a stripped-down test version of the `GUITile` class found in DarkBlue.java.

Methods in this file:

SetPiece() - Reads the image of a chess piece onto the tile.

DrawTile() - Calls the SetPiece() method and repaints the tile.

GetTile() - Returns the Tile object.

GetRow() - Returns the tile's row.

GetColumn() - Returns the tile's column.

toString() - Returns this tile as a string in algebraic notation.

GUIBoard.java:

This is a stripped-down test version of the `GUITile` class found in DarkBlue.java.

Methods in this file:

DrawBoard() - Draws the board by instantiating and repainting every tile. This also sets up notation on the sides of the board and makes changes in regards to highlighting or undoing the highlighting of legal moves if a piece has been selected or deselected.

BuildWhiteBoard() - This builds a `GUIBoard` with all white pieces oriented at the bottom and all column letters are in alphabetical order from left to right. All row numbers placed on the outside are ascending when going from bottom to top. This is a direct representation of how the `Board` class looks in memory.

BuildBlackBoard() - This builds a `GUIBoard` with all black pieces oriented at the bottom and all column letters are in alphabetical order from right to left. All row numbers placed on the outside are descending when going from bottom to top. This is an upside-down representation of how the `Board` class looks in memory.

SetBoard() - Mutator to set the `Board` object. Usually this is followed by a repaint.

PieceCount() - Convenient way of calling the PieceCount() method in the `Board` class. See the original PieceCount() method in the Board class for more details.

GetBoard() - Accessor to return the model Board.

WhoseTurnIsIt() - Convenient way of calling the model `Board`'s WhoseTurnIsIt() method. See the original method in the Board class for more information.

GetTile() - Returns a tile from the GUIBoard. Unlike its cousin in the `Board` class, the coordinates that are considered valid in this accessor are [0, 9] as opposed to [0, 7] in the model.