Project – Chess GUI with Engine  
Technical Solution  
Harley Reeves-Martin

Contents

[3.0.0 Core Modules Overview: 4](#_Toc8498843)

[3.1.0 File Organization and Processing: 10](#_Toc8498844)

[3.2.0 Explanation of complex algorithms with pseudocode comparisons: 13](#_Toc8498846)

[3.3.0 Source Code: 25](#_Toc8498847)

# References

No references.

* 1. Core Modules Overview:

|  |  |
| --- | --- |
| Input | Output |
| * Game settings (time, position, etc.) * Moves * Engine search depth * UI settings * Signup information (username, password, picture) * Login information (username, password) * Board positions from PGN * Board positions from FEN | * Result of input moves * Result of game settings selection * Result of UI settings selection * Account object when login occurs * Board positions to PGN * Board positions to FEN * Best move in the current position * Evaluation of current position * Board positions from selected moves * The name of the opening played |
| Processes | Storage |
| * Evaluate a position to the input depth * Generate a best move * Concurrently run clock whilst interface remains responsive * Display board on GUI * Display move list on GUI * Load position into board object from FEN * Load position into board object from PGN * Search account table to verify existence of account * Generate account object on sign up * Search plaintext ECO codes database and compare to current boards position * Load and make settings effective from a configuration file | * Save accounts to binary account files * Save accounts table to delimiter separated value file * Save board position to plain text FEN files * Save board position to plaintext PGN file * Store plaintext ECO codes database * Save settings to binary configuration file |

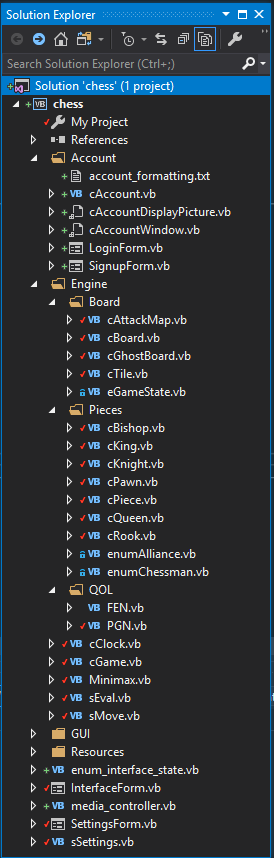
Modular structure chart:

My event is almost entirely event driven, in this structure charts events are shown as decisions, which the corresponding event name in the diamond.





Module definitions:

**../Account/**

**Account\_formatting.txt:** this is a documentation file wherein I described the formatting of the account and account table files. This will enable others to work on the project without having to search through code.

**cAccount.vb:** this is the account class containing the instance variables and methods for the instance of a class. It also contains all methods that manipulate the account table and account files, I have included these as shared methods in the interest of encapsulation.

**cAccountDisplayPicture.vb:** this is a control class which allows the image of any given account to be displayed as a circular picture box.

**cAccountWindow.vb:** this is a control class which allows details of an account to be displayed as a panel on the GUI. This is used when a user logs-in.

**LoginForm.vb:** this form is displayed when the user attempts to login. It contains text boxes allowing for the user to enter details to login with, when Ok is clicked, the form hands control to the cAccount class which determines whether the login attempt was successful or not.

**SignupForm.vb:** this form is displayed when the user attempts to sign up. It allows the user to enter required details for the creation of an account. Once Ok is clicked, the form hands control to the cAccount class which attempts to sign up with the given details.

**../Engine/  
 /Board/**

**cAttackMap.vb:** this class is used to generate an indication of what tiles are attacked by which pieces. It is used by the evaluation functions in processes such as generating scope evaluation and determining whether a piece is hanging or not.

**cBoard.vb:** this class is used to represent the board as an object. It contains all methods to do with the board, including methods to make moves, determining the set of possible and truly legal moves, and setting up positions.

**cGhostBoard.vb:** this class is composed of a board and move to be made, it is used to represent boards regardless of whether they are legal positions or not. With the given board and position, the move is made and the legality of the resulting position is tested. This class is used by the board class in determining whether a pseudo legal move is also a truly legal move.

**cTile.vb:** this class is used to represent the tile of a board. It only refers to an occupying piece as well as some informational functions such as “is\_occupied”. It is an aggregating type in the board class.

**eGameState.vb:** this is an enum type used by the board class. It provides the possible states for the board: “ongoing”, “white checkmate”, etc.

**/Pieces/**

**cPiece.vb:** this is an abstract type that details the requirements of any piece type. It contains methods that determine the set of pseudo legal moves for that instance of the piece on the given board.

**cPawn.vb & cKnight.vb & cBishop.vb & cQueen.vb:** these are all extensions of the abstract piece type. They implement the pseudo legal moves methods for themselves, and their constructors set their values depending on the piece type.

**cRook.vb & cKing.vb:** these are also extensions of the abstract piece type which implement the pseudo legal moves method. They also however contain an extra variable “has\_moved” which determines whether the piece has been moved before. This is used by the game when determining whether it is legal for the player to castle or not.

**enumAlliance.vb:** this is an enum that describes the alliance of either a piece or whose turn it is, its values are: “white”, or “black”.

**enumChessman.vb:** this is an enum that describes the title of any piece, i.e. “pawn”, “rook”, etc. It is used since it is more efficient than using the built-in “TypeOf” function whenever the type of a piece wants to be checked.

**/QOL/**

**PGN.vb:** this is a class that implements methods to both load and save boards in the PGN format.

**FEN.vb:** this is a class that implements methods to both load and save boards in the FEN format.

**../../**

**cClock.vb:** this is a class used by the game class which emulates a clock. It can be started and stopped using the toggle method. The game is associated with this class.

**cGame.vb:** this is the aggregated class which combines the functionality of all engine classes, this includes the board, clock, piece, FEN, PGN, and minimax classes. Events within the game such as a move being made are handled within this class.

**Minimax.vb:** this is a module that contains the “Minimax”, “Min” and “Max” functions that are used to search the game tree to a given depth. It is used by the game class in generating a best move for the position and uses the evaluation functions contained in “sEval”.

**sEval.vb:** this is a structure that contains the instance variables for specific aspects of the evaluation of a board. It takes in a board and finds the “Total Summed Eval” by getting the difference of the absolute values of the evaluations of the board for both white and black. It is mainly used by the Minmax module, however, is also used by the game class when an evaluation for just the current board is requested.

**sMove.vb:** this is a structure that represents a move in the game, i.e. starting tile coordinate and finishing tile coordinate. It also contains shared methods to be used by the board when a move is made, these methods are “Regular\_Move”, “Castle”, “Promotion”, etc. This is required since special moves such as castle and promotion and castle require a different implementation to a regular move.

**../GUI/**

**cBoardGUI.vb:** this is a control class which contains the methods required in representing a board object graphically. It contains event handlers for user interaction with the GUI, such as mouse down and mouse move which allow the user to input moves.

**cBoardWrapper.vb:** this is a class which implements a graphical wrapper around the board GUI. Its purpose is the aid the user in reading chess notation.

**cClockGUI:** this is a control class which contains the methods required in representing a clock graphically. It does not take any input from the user and is aggregated by a clock.

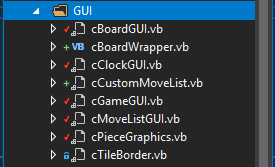
**cCustomMoveList.vb:** this is a control class which contains the methods required in representing a list of moves graphically for the user. It also contains event handlers such as “entry\_click”, which allow the user to select any point of the current game being played, and the board showed by the board GUI will reflect the position at the selected point. This is useful for players who are analysing their games or want to review how they reached a certain position.

**cGameGUI:** this is the control class which is aggregated by the other GUI classes. It a panel containing the other GUI classes, making it easy for anyone using this library to add a GUI that can display everything rather than having to create each individual GUI themselves. Furthermore, it has methods that disables the GUI when the game is over.

**cMoveListGUI.vb:** this class is no longer used.

**cPieceGraphics.vb:** this is a control class used by the board GUI class when a piece is selected and dragged. It represents the piece as a picture box; however the region of the picture box is trimmed to reflect the same of the piece.

**cTileBorder.vb:** this is a control class used by the board GUI class, it is used to highlight the outer side of a tile white when a piece is hovering over that piece.



**../..**

**Enum\_interface\_state.vb:** this is an enum that describes the current state of the interface. I.e. if the user is in the settings menu or analysis board.

**InterfaceForm.vb:** this is a form that is the start-up form of the application and is also where the main game GUI and game is instantiated. It contains multiple buttons allowing the user to select what to do: such as start a new game, or edit settings.

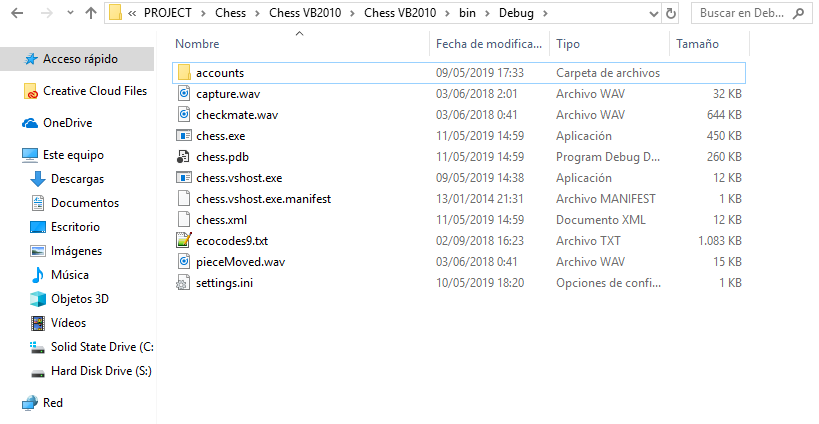
**SettingsForm.vb:** this is a form that allows the user to edit the interface settings in the program. It uses checkboxes and colour dialogs to allow the user to input these settings.

**sSettings.vb:** this is a structure that stores settings used by the application. It also contains shared methods that allow the application to read configuration files and write the instance of settings to a configuration file or generate default settings objects and files if a configuration file cannot be found.

https://i.gyazo.com/445d90f9ad0d274714cd5498e98c1037.png Below, I will describe the function of the two network classes implemented in my program. Despite the implementation of these classes, due to time constraints, I will be unable to implement them to allow play over a local area network in my program for now.

**Client.vb:** the client class is a class that allows the programmer to easily set up a TCP client, and connect that client to a socket on the local area network, where the IP-address and port are given in the constructor of the class. It features a send sub which allows the user to send a stream of data to the target socket, if a connection is successful.

**Controller.vb: the** controller class is essentially the host in the connection. Upon instantiation of a controller, the program will listen on the specified IP & port, until a message is received, from where it will raise an event so that the programmer can decide what to do when a message is received. In the case of my program, the event would be handled in the game classes event loop for online games. This is something I plan to implement in the future.

* 1. File Organization and Processing:

Executable file:

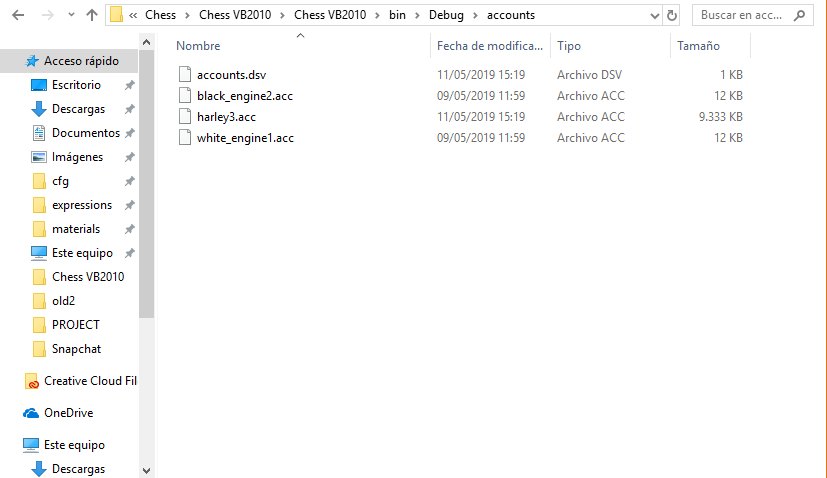
The executable file requires a package for full functionality. This includes the sound files that are used by the GUI, as well as an accounts file if the user does not have to want to create a new account. Despite this it is easy for anyone to place the executable with its dependencies on a public drive (such as the public drive at my school) so that users can use the application from within the school network. This is important as it allows pupils of a chess club to easily use the program without having to install anything, they will however need permission to edit files in the directory of the executable for full functionality.

Configuration file:

The configuration file is designed to be stored in the same directory as where the binary file is stored. Therefore, for the user to be able to edit the configuration file from within the program, the user must also have access to edit files in the directory of the executable. The configuration file will be stored as a binary file, and therefore will only be editable through my application. The user cannot choose where to save this file.

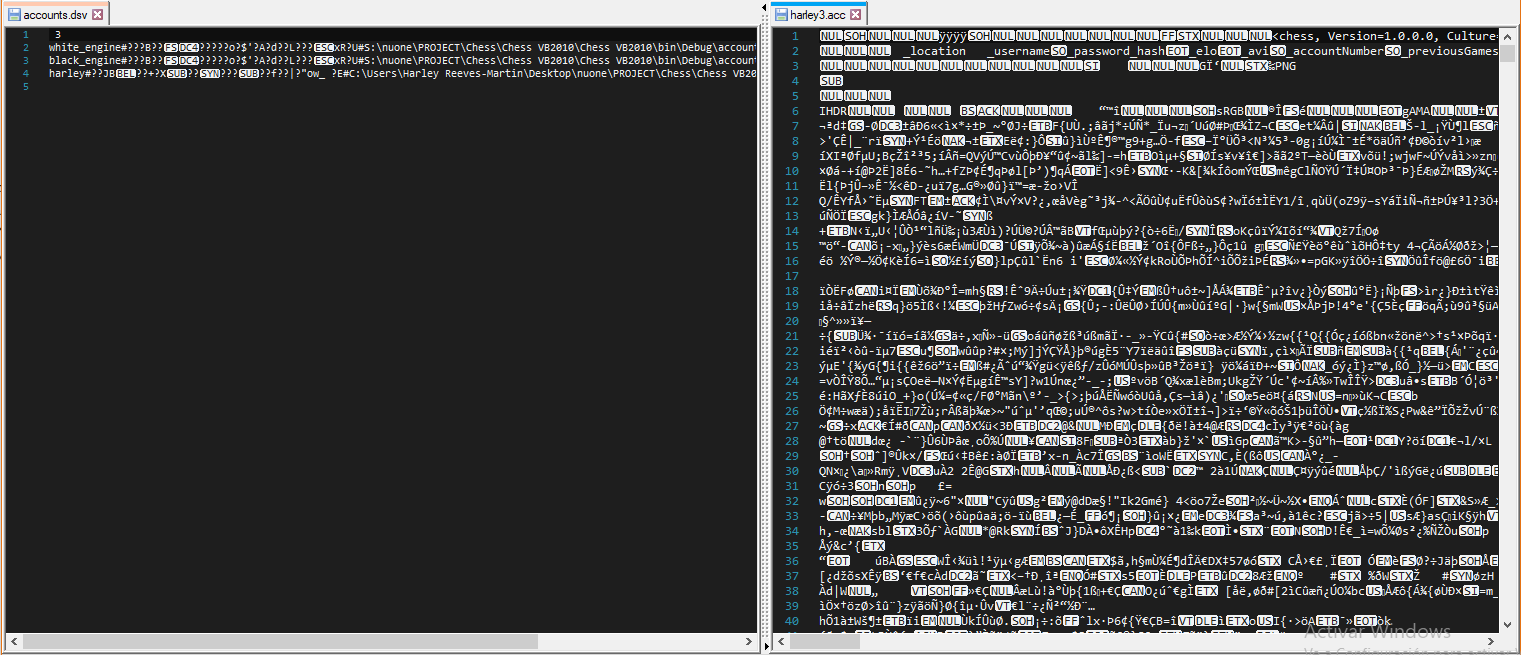
Ecocodes9 file:

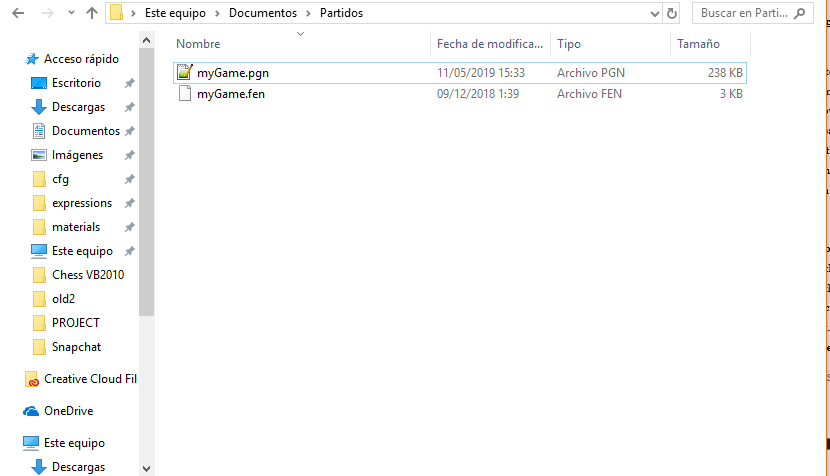
The ECO codes file will be stored as plaintext. This enables my program to easily parse the file, and allows users to easily add to the database given that they understand the format required.



Accounts files:

All files relating to accounts are stored in the same directory as the executable, in a folder called accounts. The account table file is labelled as a “.dsv” file and is stored in plaintext. Whereas account files are labelled as “.acc” files and are stored as binary files. The user cannot choose where the save these files. Below are screenshots of the contents of the files when opened by notepad++.





Game files:

Both PGN and FEN files are saved as text files to make it easy for the user to read the files using any text editor. Furthermore, the PGN and FEN format are defined to be plaintext also, this consistency makes the files portable across chess applications.

3.2.0 Explanation of complex algorithms with pseudocode comparisons:

Generate pseudo-legal move:

This algorithm is used to generate a pseudo legal move for a piece.

|  |
| --- |
| Pseudocode |
|  |

|  |
| --- |
| VB.NET code |
| In cPiece  Public MustOverride Function calc\_pseudo(BOARD As cBoard) As sMove()  In cPawn  Public Overrides Function calc\_pseudo(BOARD As cBoard) As sMove()  Dim possibleMoveOffsets() As Integer = {7, 8, 9}  Dim multiplier As Integer  Dim legalMoves() As sMove = Nothing  Dim counter As Integer = 0  If Me.\_alliance = Alliance.White Then multiplier = -1 Else multiplier = 1  For i = 0 To UBound(possibleMoveOffsets)  Dim targetCoordinate As Integer = Me.\_coordinate + possibleMoveOffsets(i) \* multiplier  If is\_valid\_tile(targetCoordinate, Me.\_coordinate, (possibleMoveOffsets(i) \* multiplier)) Then  'HANDLES CAPTURE MOVES  If possibleMoveOffsets(i) = 7 Or possibleMoveOffsets(i) = 9 Then  If BOARD.get\_tile(CByte(targetCoordinate)).is\_occupied AndAlso  BOARD.get\_tile(CByte(targetCoordinate)).get\_piece.get\_alliance <> Me.\_alliance Then  ReDim Preserve legalMoves(counter)  legalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoordinate))  counter += 1  ElseIf BOARD.get\_tile(CByte(targetCoordinate)).get\_coordinate = BOARD.get\_enpassent Then  'HANDLES CAPTURING EN PASSENT  ReDim Preserve legalMoves(counter)  legalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoordinate))  counter += 1  End If  Else  'HANDLES FORWARDS MOVES  If Not BOARD.get\_tile(CByte(targetCoordinate)).is\_occupied Then  ReDim Preserve legalMoves(counter)  legalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoordinate))  counter += 1  'HANDLES THRUSTING  targetCoordinate = targetCoordinate + (8 \* multiplier)  If is\_valid\_tile(targetCoordinate, Me.\_coordinate, 16 \* multiplier) Then  If ((Me.\_coordinate \ 8 + 1) = 7 AndAlso Me.\_alliance = Alliance.White AndAlso Not BOARD.get\_tile(CByte(targetCoordinate)).is\_occupied) \_  Or ((Me.\_coordinate \ 8 + 1) = 2 AndAlso Me.\_alliance = Alliance.Black AndAlso Not BOARD.get\_tile(CByte(targetCoordinate)).is\_occupied) Then  ReDim Preserve legalMoves(counter)  legalMoves(counter) = New sMove(Me.\_coordinate, CByte((targetCoordinate)))  counter += 1  End If  End If  End If  End If  End If  Next  Return legalMoves  End Function  In cQueen  Public Overrides Function calc\_pseudo(BOARD As cBoard) As sMove()  Dim possibleMoveOffsets As SByte() = {-8, -1, 1, 8, -9, -7, 7, 9}  Dim legalMoves() As sMove = Nothing  Dim counter As Integer = 0  For i = 0 To UBound(possibleMoveOffsets)  Dim targetCoord As Integer = Me.\_coordinate + possibleMoveOffsets(i)  While is\_valid\_tile(targetCoord, CByte(targetCoord - possibleMoveOffsets(i)), possibleMoveOffsets(i))  If BOARD.get\_tile(CByte(targetCoord)).is\_occupied Then  If BOARD.get\_tile(CByte(targetCoord)).get\_piece.get\_alliance <> Me.\_alliance Then  ReDim Preserve legalMoves(counter)  legalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoord))  counter += 1  End If  Exit While  Else  ReDim Preserve legalMoves(counter)  legalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoord))  counter += 1  End If  targetCoord += possibleMoveOffsets(i)  End While  Next  Return legalMoves  End Function  Private Function is\_valid\_tile(TARGET\_COORDIANTE As Integer, OG\_COORDINATE As Byte, OFFSET As Integer) As Boolean  Dim ogCol As Integer = (OG\_COORDINATE Mod 8 + 1)  Select Case ogCol  Case 1  If OFFSET = -1 Or OFFSET = -9 Or OFFSET = 7 Then Return False  Case 8  If OFFSET = 1 Or OFFSET = -7 Or OFFSET = 9 Then Return False  End Select  If TARGET\_COORDIANTE >= 0 And TARGET\_COORDIANTE <= 63 Then  Return True  End If  Return False  End Function |

It is evident that the real code used by my program is substantially longer than expected. This is because in the development of my program, I found a more effective solution to the problem in terms of computation time, however this did increase the complexity of the code. Despite this, because these are functions that should never need to be edited again, I believe that code being more complex is a valuable trade-off for a faster executing algorithm.

Generate legal moves:

This is used to generate the legal moves of a set of pseudo legal moves in the board class.

|  |
| --- |
| Pseudocode: |
|  |

|  |
| --- |
| VB.NET code: |
| ''' <summary>  ''' Returns moves which are allowed by each pieces normal moves, taking pins into account.  ''' </summary>  Private Function calc\_legal() As sMove()  Dim moveSet As sMove()  Dim temp As sMove() = Nothing  Dim counter As Integer = 0  If Me.\_whose\_turn = Alliance.White Then moveSet = Me.\_white\_pseudo Else moveSet = Me.\_black\_pseudo  For Each move As sMove In moveSet  Dim x As New cGhostBoard(Me, move)  If x.is\_legal Then  ReDim Preserve temp(counter)  temp(counter) = move  counter += 1  End If  Next  Return temp  End Function |

In this example, the true code is as expected.

Generate best move:

This will decide a best move in any given position.

|  |
| --- |
| Pseudocode |
|  |

|  |
| --- |
| VB.NET code |
| ''' <summary>  ''' Returns the best move as determined by looking the given number of half-plys ahead from a given position.  ''' </summary>  ''' <param name="BOARD">Inputs the position to begin evaluation from.</param>  ''' <param name="DEPTH">Specifies how many half-plies forward the engine will search to generate the evaluation score. Higher depths will score as position more accurately at the expense of time.</param>  ''' <returns></returns>  Public Shared Function calc\_best\_move(DEPTH As Integer, BOARD As cBoard) As sMove  Dim highest As Double = Double.MinValue  Dim lowest As Double = Double.MaxValue  Dim current As Double  Dim bestMove As sMove  For Each move As sMove In BOARD.get\_legal\_moves  Dim b As cBoard = BOARD.get\_deep\_clone(Of cBoard)(BOARD)  b.make\_move(move)  If b.get\_whose\_turn = Alliance.Black Then  current = max(DEPTH - 1, b)  If current >= highest Then  bestMove = move  highest = current  End If  ElseIf b.get\_whose\_turn = Alliance.White Then  current = min(DEPTH - 1, b)  If current <= lowest Then  bestMove = move  lowest = current  End If  End If  Next  Return bestMove  End Function  Private Shared Function min(DEPTH As Integer, BOARD As cBoard) As Double  Dim minimum As Double = Double.MaxValue  If DEPTH = 0 Then  Return New sEval(BOARD).get\_summed\_eval ''TODO: TEST  Else  Dim b As cBoard = BOARD.get\_deep\_clone(Of cBoard)(BOARD)  For Each move As sMove In BOARD.get\_legal\_moves  b.make\_move(move)  Dim val As Double = max(DEPTH - 1, b)  If val >= minimum Then minimum = val  Next  End If  Return minimum  End Function  Private Shared Function max(DEPTH As Integer, BOARD As cBoard) As Double  Dim maximum As Double = Double.MinValue  If DEPTH = 0 Then  Return New sEval(BOARD).get\_summed\_eval  Else  Dim b As cBoard = BOARD.get\_deep\_clone(Of cBoard)(BOARD)  For Each move As sMove In BOARD.get\_legal\_moves  b.make\_move(move)  Dim val As Double = min(DEPTH - 1, b)  If val <= maximum Then maximum = val  Next  End If  Return maximum  End Function |

There is one main difference between the planned implementation and actual implementation here: firstly, since a board is made up of reference types, before making a move on the board, I had to create a deep clone of the board and then make the move to ensure that members of the original passed in board were not altered during execution of the algorithm.

Evaluate position:

This is an algorithm that will give an evaluation for any given position.

|  |
| --- |
| Pseudocode |
|  |

|  |
| --- |
| VB.NET code |
| Public Sub New(BOARD As cBoard)  Me.\_board = BOARD  Me.\_attackMap = New cAttackMap(Me.\_board)  Me.\_wMaterial = eval\_material(Alliance.White) : Me.\_wScope = eval\_scope(Alliance.White) : Me.\_wKingSafety = eval\_king\_safety(Alliance.White)  Me.\_bMaterial = eval\_material(Alliance.Black) : Me.\_bScope = eval\_scope(Alliance.Black) : Me.\_bKingSafety = eval\_king\_safety(Alliance.Black)  Me.\_summedEval = calc\_summed\_eval()  End Sub  Public Function calc\_summed\_eval() As Double  Return (Me.\_wMaterial + Me.\_wScope + Me.\_wKingSafety + Me.\_bHangingMaterial) - (Me.\_bMaterial + Me.\_bScope + Me.\_bKingSafety + Me.\_wHangingMaterial)  End Function  #Region "Evaluation Functions"  Private Function eval\_material(ALLIANCE As Alliance) As Double  Dim val As Integer  For Each piece As cPiece In Me.\_board.get\_pieces(ALLIANCE)  If Not (TypeOf piece Is cKing) Then val += piece.get\_value  Next  Return val  End Function  Private Function eval\_king\_safety(ALLIANCE As Alliance) As Double  Dim moves As sMove() = Me.\_board.get\_pseudo\_legal(ALLIANCE)  Dim opponent As Alliance : If ALLIANCE = chess.Alliance.White Then opponent = chess.Alliance.Black Else opponent = chess.Alliance.White  Dim kingLoc As Byte = Me.\_board.find\_king(opponent).get\_coordinate  Dim attacks As Byte = 0  If moves IsNot Nothing Then  For Each move As sMove In moves  If move.dest = kingLoc Then attacks += CByte(1)  Next  End If  Return 0 ''REMOVE THIS AFTER TESTING''  Return CDbl(0.05 \* attacks)  End Function  Private Function eval\_scope(ALLIANCE As Alliance) As Double  Dim moves As sMove() = Me.\_board.get\_pseudo\_legal(ALLIANCE)  Return CDbl(moves.Length \* 0.05)  End Function  #End Region |

Once again, the actual implementation of this algorithm was much more complicated than expected. For the minimax algorithm to work effectively, it must be supported by an accurate heuristic, therefore I decided to add more in-depth evaluation functions to increase the accuracy of my engine. Also, instead of creating just an algorithm for the evaluation, I made it a class allowing for more usage of the evaluation once it is made.

Make Move:

This method is required to make any given move on a board.

|  |
| --- |
| Pseudocode |
|  |

|  |
| --- |
| VB.NET code |
| ''' <summary>  ''' Updates the current instance of cBoard to reflect the move attempted, if the move is not legal, it will not be made.  ''' </summary>  ''' <param name="MOVE"></param>  ''' <param name="FROM\_TRANSITION"></param>  Public Sub make\_move(MOVE As sMove, Optional FROM\_TRANSITION As Boolean = False)  If MOVE.ogCoord = 4 AndAlso MOVE.dest = 2 Then MsgBox("TEST")  Dim movingPiece As cPiece = Me.get\_tile(MOVE.ogCoord).get\_piece  Dim targetTile As cTile = Me.get\_tile(MOVE.dest)  Dim wasCapture As Boolean  wasCapture = Me.\_tiles(MOVE.dest).is\_occupied  If FROM\_TRANSITION Then  Dim castlingTiles As Byte() = {58, 62, 2, 6}  If movingPiece.get\_title = Chessman.Pawn AndAlso targetTile.get\_coordinate = Me.\_en\_passent\_coord Then 'en passent capture  sMove.enpassent(MOVE, Me)  ElseIf movingPiece.get\_title = Chessman.Pawn AndAlso ((targetTile.get\_coordinate \ 8 + 1 = 1) AndAlso movingPiece.get\_alliance = Alliance.White) Then 'black promotion  sMove.promotion(MOVE, Me)  ElseIf movingPiece.get\_title = Chessman.Pawn AndAlso (targetTile.get\_coordinate \ 8 + 1 = 8) AndAlso movingPiece.get\_alliance = Alliance.Black Then 'white promotion  sMove.promotion(MOVE, Me)  ElseIf movingPiece.get\_title = Chessman.King AndAlso Not CType(movingPiece, cKing).get\_moved AndAlso castlingTiles.Contains(MOVE.dest) Then 'castling  sMove.castle(MOVE, Me)  Else 'normal move66  sMove.regular(MOVE, Me)  End If  If movingPiece.get\_title = Chessman.King Then CType(movingPiece, cKing).set\_moved(True)  Me.\_move\_list\_string.AddLast(MOVE.ToString)  Me.update\_board\_members(wasCapture, movingPiece, FROM\_TRANSITION)  Else  Dim castlingTiles As Byte() = {58, 62, 2, 6}  If Me.\_legal\_moves.Contains(MOVE) Then  'assume the move is added to the move list by the GUI  If movingPiece.get\_title = Chessman.Pawn AndAlso targetTile.get\_coordinate = Me.\_en\_passent\_coord Then 'en passent capture  Me.\_en\_passent\_coord = 255  sMove.enpassent(MOVE, Me)  wasCapture = True  ElseIf movingPiece.get\_title = Chessman.Pawn AndAlso ((targetTile.get\_coordinate \ 8 + 1 = 1) AndAlso movingPiece.get\_alliance = Alliance.White) Then 'black promotion  sMove.promotion(MOVE, Me)  ElseIf movingPiece.get\_title = Chessman.Pawn AndAlso (targetTile.get\_coordinate \ 8 + 1 = 8) AndAlso movingPiece.get\_alliance = Alliance.Black Then 'white promotion  sMove.promotion(MOVE, Me)  ElseIf movingPiece.get\_title = Chessman.King AndAlso Not CType(movingPiece, cKing).get\_moved AndAlso castlingTiles.Contains(MOVE.dest) Then 'castling  sMove.castle(MOVE, Me)  Else 'normal move66  sMove.regular(MOVE, Me)  End If  'update the en-passent tile  If movingPiece.get\_title = Chessman.Pawn Then  If 9 - (MOVE.ogCoord \ 8 + 1) = 2 Then  If MOVE.dest = MOVE.ogCoord - 16 Then  Me.\_en\_passent\_coord = CByte(MOVE.ogCoord - 8)  Else  Me.\_en\_passent\_coord = 255  End If  End If  If 9 - (MOVE.ogCoord \ 8 + 1) = 7 Then  If MOVE.dest = MOVE.ogCoord + 16 Then  Me.\_en\_passent\_coord = CByte(MOVE.ogCoord + 8)  Else  Me.\_en\_passent\_coord = 255  End If  End If  Else  Me.\_en\_passent\_coord = 255  End If  If movingPiece.get\_title = Chessman.King Then CType(movingPiece, cKing).set\_moved(True)  If Not FROM\_TRANSITION Then Me.\_move\_list.AddLast(MOVE)  'updates all members of the board that need to be updated after a move.  Me.update\_board\_members(wasCapture, movingPiece, FROM\_TRANSITION)  End If  End If  End Sub  In sMove:  #Region "Moves"  Public Shared Sub regular(MOVE As sMove, BOARD As cBoard)  Dim movingPiece As cPiece = BOARD.get\_tile(MOVE.ogCoord).get\_piece 'error @ 1. e4, d5 2. exd5, c6 3. dxc6, Bd7 4. cxb7, Be6 5. bxa8=Q, Qd7 6. Qaf3  Dim targetTile As cTile = BOARD.get\_tile(MOVE.dest)  movingPiece.set\_coordinate(targetTile.get\_coordinate) 'update the pieces coordinate  targetTile.set\_piece(movingPiece) 'update the tiles piece reference  BOARD.get\_tiles(MOVE.ogCoord).set\_piece(Nothing) 'update the og tiles piece reference  End Sub  Public Shared Sub enpassent(MOVE As sMove, BOARD As cBoard)  Dim movingPiece As cPiece = BOARD.get\_tile(MOVE.ogCoord).get\_piece  Dim targetTile As cTile = BOARD.get\_tile(MOVE.dest)  movingPiece.set\_coordinate(MOVE.dest)  targetTile.set\_piece(movingPiece)  If BOARD.get\_whose\_turn = Alliance.White Then  BOARD.get\_tile(CByte(MOVE.dest + 8)).set\_piece(Nothing)  Else  BOARD.get\_tile(CByte(MOVE.dest - 8)).set\_piece(Nothing)  End If  BOARD.get\_tile(MOVE.ogCoord).set\_piece(Nothing)  End Sub  Public Shared Sub castle(MOVE As sMove, BOARD As cBoard)  sMove.regular(MOVE, BOARD)  Dim rookMove As sMove  Select Case MOVE.dest  Case 58  rookMove = New sMove(56, 59)  Case 62  rookMove = New sMove(63, 61)  Case 2  rookMove = New sMove(0, 3)  Case 6  rookMove = New sMove(7, 5)  End Select  sMove.regular(rookMove, BOARD)  End Sub  Public Shared Sub promotion(MOVE As sMove, BOARD As cBoard)  Dim movingPiece As cPiece = BOARD.get\_tile(MOVE.ogCoord).get\_piece  Dim targetTile As cTile = BOARD.get\_tile(MOVE.dest)  targetTile.set\_piece(New cQueen(movingPiece.get\_alliance, MOVE.dest)) 'puts the new queen on the board  BOARD.get\_tile(MOVE.ogCoord).set\_piece(Nothing) 'removes the promoting piece from the board  BOARD.update\_pieces(Alliance.White) 'updates the piece list of the board for both players.  BOARD.update\_pieces(Alliance.Black)  End Sub  #End Region |

In making the move for a piece, the algorithm was as complicated as expected. However, in order to utilize encapsulation properly in my program, I implemented the special moves and how they would be handled as shared methods in the move structure. This encapsulation lead to bugs being easier to fix when they were found and meant that code is easier to read.

Load/Save settings file:

This algorithm must allow the user to save and load settings files.

|  |
| --- |
| Pseudocode |
|  |

|  |
| --- |
| VB.NET code |
| Public Sub load()  If Not System.IO.File.Exists(System.AppDomain.CurrentDomain.BaseDirectory & "\settings.ini") Then  sSettings.gen\_default\_file()  End If  Dim r As New System.IO.BinaryReader(System.IO.File.Open(System.AppDomain.CurrentDomain.BaseDirectory & "\settings.ini", IO.FileMode.Open))  Dim cR, cG, cB As Byte  cR = r.ReadByte : cG = r.ReadByte : cB = r.ReadByte 'reads lights colour  Me.lightSquare = Color.FromArgb(cR, cG, cB)  cR = r.ReadByte : cG = r.ReadByte : cB = r.ReadByte 'reads dark colour  Me.darkSquare = Color.FromArgb(cR, cG, cB)  Me.showLegalMoves = r.ReadBoolean  Me.enableCheats = r.ReadBoolean  Me.nightMode = r.ReadBoolean  r.Close()  End Sub  Public Sub save()  Dim w As New System.IO.BinaryWriter(System.IO.File.Open(System.AppDomain.CurrentDomain.BaseDirectory & "\settings.ini", IO.FileMode.OpenOrCreate)) 'open or create the .ini file  With Me.lightSquare  w.Write(.R) : w.Write(.G) : w.Write(.B)  End With  With Me.darkSquare  w.Write(.R) : w.Write(.G) : w.Write(.B)  End With  w.Write(showLegalMoves)  w.Write(enableCheats)  w.Write(nightMode)  w.Close()  InterfaceForm.settings.load()  End Sub |

This algorithm was implemented as expected.

3.3.0 Source Code:

Imports System.IO

Imports System.Runtime.Serialization.Formatters.Binary

''' <summary>

''' Serves as an object to store accounts in the application.

''' Also contains required shared methods to setup an account storage system to be used with the application.

''' </summary>

<Serializable()>

Public Class cAccount

Private \_location As String

Private \_username As String

Private \_password\_hash As String

Private \_elo As Integer

Private \_avi As Bitmap

Private \_accountNumber As Integer

'statistics

Private \_previousGames As List(Of PGN)

Private \_wins As Integer

Private \_losses As Integer

Private \_draws As Integer

Public Enum login\_response

valid

username\_invalid

password\_invalid

End Enum

#Region "Static members"

Public Shared numberOfAccounts As Integer

Public Shared accounts\_path As String = System.AppDomain.CurrentDomain.BaseDirectory & "accounts\" 'concatenate the name of the file + .acc to access the account file.

Public Shared table\_path As String = System.AppDomain.CurrentDomain.BaseDirectory & "accounts\accounts.dsv"

Public Shared Sub initialize\_account\_management(Optional force\_change As Boolean = False)

If Not cAccount.account\_table\_exists OrElse (force\_change) Then

cAccount.generate\_account\_table()

Else

cAccount.update\_no\_of\_accounts()

End If

End Sub

Private Shared Sub update\_no\_of\_accounts()

FileOpen(1, cAccount.table\_path, OpenMode.Input)

cAccount.numberOfAccounts = CType(LineInput(1), Integer)

FileClose(1)

End Sub

Public Shared Function account\_table\_exists() As Boolean

If IO.File.Exists(cAccount.table\_path) Then Return True Else Return False

End Function

Public Shared Sub generate\_account\_table()

If cAccount.account\_table\_exists Then

My.Computer.FileSystem.DeleteFile(cAccount.table\_path) ''deletes the existing accounts file if it already exists.

End If

FileOpen(1, cAccount.table\_path, OpenMode.Output) 'create the account table, printline creates the first entry, which indicates number of accounts stored by the table

PrintLine(1, 0)

FileClose(1)

If Not File.Exists(cAccount.accounts\_path & "white\_engine.acc") Then

cAccount.generate\_account("white\_engine", cAccount.hash\_pword(""), 1000, My.Resources.white\_engine\_avi)

End If

If Not File.Exists(cAccount.accounts\_path & "black\_engine.acc") Then

cAccount.generate\_account("black\_engine", cAccount.hash\_pword(""), 950, My.Resources.black\_engine\_avi)

End If

End Sub

Public Shared Function generate\_account(un As String, ph As String, e As Integer, aloc As String) As cAccount

Dim a As cAccount = New cAccount(un, ph, e, aloc)

Using w As BinaryWriter = New BinaryWriter(File.Open(a.\_location, FileMode.Create))

w.Write(a.to\_stream)

End Using

cAccount.add\_account\_to\_table(a)

Return a

End Function

Public Shared Function generate\_account(un As String, ph As String, e As Integer, avi As Bitmap) As cAccount

Dim a As cAccount = New cAccount(un, ph, e, avi)

Using w As BinaryWriter = New BinaryWriter(File.Open(a.\_location, FileMode.Create))

w.Write(a.to\_stream)

End Using

cAccount.add\_account\_to\_table(a)

Return a

End Function

Private Shared Sub add\_account\_to\_table(a As cAccount)

FileOpen(1, cAccount.table\_path, OpenMode.Append)

Dim s As String = a.\_username & "#" & a.\_password\_hash & "#" & a.\_location

PrintLine(1, s)

FileClose(1)

cAccount.increment\_account\_counter()

End Sub

Private Shared Sub increment\_account\_counter()

cAccount.numberOfAccounts += 1

If Not File.Exists(cAccount.table\_path) Then Throw New Exception("Account database not found, please generate a new one.")

Dim inputs As List(Of String) = New List(Of String)

FileOpen(1, cAccount.table\_path, OpenMode.Input)

Dim [end] As Boolean = False

Do While Not [end]

Try

inputs.Add(LineInput(1))

Catch ex As Exception

[end] = True

End Try

Loop

FileClose(1)

FileOpen(1, cAccount.table\_path, OpenMode.Output)

For Each s As String In inputs

If inputs.IndexOf(s) = 0 Then

PrintLine(1, CInt(s) + 1)

Else

PrintLine(1, s)

End If

Next

FileClose(1)

End Sub

Public Shared Function hash\_pword(p As String) As String

Dim r As String = Nothing

Dim provider As System.Security.Cryptography.SHA256CryptoServiceProvider = New Security.Cryptography.SHA256CryptoServiceProvider

Dim bytes As Byte() = provider.ComputeHash(System.Text.Encoding.ASCII.GetBytes(p))

r = System.Text.Encoding.ASCII.GetString(bytes)

Return r

End Function

Public Shared Function login(username As String, hash As String) As Tuple(Of login\_response, cAccount)

Dim r1 As login\_response = login\_response.username\_invalid

Dim r2 As cAccount = Nothing

Dim inputs As List(Of String) = New List(Of String)

FileOpen(1, cAccount.table\_path, OpenMode.Input)

LineInput(1)

Do While True

Try

inputs.Add(LineInput(1))

Catch ex As Exception

Exit Do

End Try

Loop

FileClose(1)

For Each s As String In inputs

Dim ss As String() = Split(s, "#")

If ss(0) = username AndAlso ss(1) = hash Then

r1 = login\_response.valid

r2 = cAccount.fetch\_profile(ss(2))

Exit For

ElseIf ss(0) = username AndAlso ss(1) <> hash Then

r1 = login\_response.password\_invalid

Exit For

End If

Next

Return New Tuple(Of login\_response, cAccount)(r1, r2)

End Function

Public Shared Function fetch\_profile(path As String) As cAccount

Dim r As cAccount

Using reader As BinaryReader = New BinaryReader(File.Open(path, FileMode.Open))

r = cAccount.from\_stream(reader.BaseStream)

End Using

Return r

End Function

Public Shared Function from\_stream(stream As Stream) As cAccount

If (Object.ReferenceEquals(stream, Nothing)) Then Return Nothing

Dim formatter As New BinaryFormatter()

Return CType(formatter.Deserialize(stream), cAccount)

End Function

#End Region

#Region "Mutators & Accessors"

Public Sub set\_username(v As String)

Me.\_username = v

End Sub

Public Function get\_username() As String

Return Me.\_username

End Function

Public Sub set\_elo(v As Integer)

Me.\_elo = v

End Sub

Public Function get\_elo() As Integer

Return Me.\_elo

End Function

Public Function get\_previous\_games() As List(Of PGN)

Return Me.\_previousGames

End Function

Public Sub set\_wins(v As Integer)

Me.\_wins = v

End Sub

Public Function get\_wins() As Integer

Return Me.\_wins

End Function

Public Sub set\_draws(v As Integer)

Me.\_draws = v

End Sub

Public Function get\_draws() As Integer

Return Me.\_draws

End Function

Public Sub set\_losses(v As Integer)

Me.\_losses = v

End Sub

Public Function get\_losses() As Integer

Return Me.\_losses

End Function

Public Function get\_avi() As Bitmap

Return Me.\_avi

End Function

#End Region

''Only having a private constructor makes it impossible to create an

'' account in the standard idea of object instantiation, the generate\_account method must be used instead which will

'' also add the account to the account to the account table. And ensure the account is valid.

Private Sub New(un As String, ph As String, e As Integer, avi As String)

cAccount.update\_no\_of\_accounts()

With (Me)

.\_username = un : .\_password\_hash = ph : .\_elo = e

.\_previousGames = New List(Of PGN) : .\_wins = 0 : .\_draws = 0 : .\_losses = 0

.\_accountNumber = (cAccount.numberOfAccounts + 1)

.\_avi = New Bitmap(avi)

.\_location = cAccount.accounts\_path & .\_username & .\_accountNumber & ".acc"

End With

End Sub

Private Sub New(un As String, ph As String, e As Integer, avi As Bitmap)

cAccount.update\_no\_of\_accounts()

With (Me)

.\_username = un : .\_password\_hash = ph : .\_elo = e

.\_previousGames = New List(Of PGN) : .\_wins = 0 : .\_draws = 0 : .\_losses = 0

.\_accountNumber = (cAccount.numberOfAccounts + 1)

.\_avi = avi

.\_location = cAccount.accounts\_path & .\_username & .\_accountNumber & ".acc"

End With

End Sub

Private Function to\_stream() As Byte()

If (Object.ReferenceEquals(Me, Nothing)) Then Return Nothing

Dim formatter As New BinaryFormatter()

Dim stream As New MemoryStream()

formatter.Serialize(stream, Me)

stream.Seek(0, SeekOrigin.Begin)

Return stream.ToArray

End Function

End Class

Imports System.Drawing

Imports System.Drawing.Drawing2D

Public Class cAccountDisplayPicture

Inherits PictureBox

Private Sub set\_default()

Me.Image = My.Resources.default\_dop

End Sub

Private Sub update\_region()

If (Me.Region IsNot Nothing) Then

Me.Region.Dispose()

Me.Region = Nothing

End If

Dim path As GraphicsPath = New GraphicsPath : path.Reset()

path.AddEllipse(0, 0, Me.Width, Me.Height)

Me.Region = New Region(path)

End Sub

Private Sub e\_size\_changed(sender As Object, e As EventArgs) Handles MyBase.SizeChanged

Me.update\_region()

End Sub

Public Sub New(ByRef Optional loc As String = "")

Me.SizeMode = PictureBoxSizeMode.StretchImage

Me.init\_image(loc)

End Sub

Public Sub reset\_image()

Me.set\_default()

End Sub

Public Sub init\_image(ByRef Optional loc As String = "")

If loc = "" Then

Me.set\_default()

Else

If IO.File.Exists(loc) Then

Try

Dim b As Bitmap = New Bitmap(loc)

Me.Image = b

Catch ex As Exception

Me.set\_default()

MsgBox("Image could not be loaded, please try a different image.")

End Try

Else

MsgBox("File does not exist. Please try a different image.")

Me.set\_default()

End If

End If

End Sub

End Class

Public Class cAccountWindow

Inherits Panel

Private \_account As cAccount

Private \_dp As cAccountDisplayPicture

Private \_nameplate As Label

Private \_eloplate As Label

Private Event account\_changed As EventHandler

#Region "Mutators & Accessors"

Public Sub set\_account(ByRef val As cAccount)

Me.\_account = val

RaiseEvent account\_changed(Me, Nothing)

End Sub

#End Region

Public Sub New(a As cAccount)

Me.\_account = a

Me.BackColor = Color.FromArgb(40, 40, 40)

Me.init\_components()

End Sub

Private Sub init\_components()

Me.\_dp = New cAccountDisplayPicture : Me.\_nameplate = New Label : Me.\_eloplate = New Label

Me.\_dp.Image = Me.\_account.get\_avi

Me.\_nameplate.Font = New Font("Microsoft Sans Serif", 8, FontStyle.Bold) : Me.\_nameplate.ForeColor = Color.White

Me.\_eloplate.Font = New Font("Microsoft Sans Serif", 8, FontStyle.Bold) : Me.\_eloplate.ForeColor = Color.White

Me.\_nameplate.AutoSize = True : Me.\_eloplate.AutoSize = True

Me.Controls.Add(Me.\_dp) : Me.Controls.Add(Me.\_nameplate) : Me.Controls.Add(Me.\_eloplate)

Me.\_nameplate.Text = Me.\_account.get\_username()

Me.\_eloplate.Text = Me.\_account.get\_elo.ToString & " (" & Me.\_account.get\_wins().ToString & "-" & Me.\_account.get\_draws().ToString & "-" & Me.\_account.get\_losses.ToString & ")"

End Sub

Private Sub e\_account\_changed(sender As Object, e As EventArgs) Handles Me.account\_changed

Me.\_dp.Image = Me.\_account.get\_avi : Me.\_nameplate.Text = Me.\_account.get\_username

Me.\_eloplate.Text = Me.\_account.get\_elo.ToString & Me.\_account.get\_wins().ToString & "-" & Me.\_account.get\_draws().ToString & "-" & Me.\_account.get\_losses.ToString

End Sub

Private Sub e\_resize(sender As Object, e As EventArgs) Handles MyBase.Resize

Dim max As Integer : If Me.Width > Me.Height Then max = Me.Height Else max = Me.Width

Me.\_dp.Size = New Size(max - 3, max - 3) : Me.\_nameplate.Location = New Point(max + 5, 8)

Me.\_eloplate.Location = New Point(max + 5, Me.\_eloplate.Height + Me.\_nameplate.Height)

End Sub

End Class

Imports System.Security

Imports System.Security.Cryptography

Imports System.Text

Public Class LoginForm

''drag-drop functionality''

Private \_xStart As Integer

Private \_yStart As Integer

Private Sub e\_form\_mousedown(sender As Object, e As MouseEventArgs) Handles TitleStripPanel.MouseDown, MyBase.MouseDown, TitleLabel.MouseDown

\_xStart = e.X : \_yStart = e.Y

End Sub

Private Sub e\_form\_dragdrop(sender As Object, e As MouseEventArgs) Handles TitleStripPanel.MouseMove, MyBase.MouseMove, TitleLabel.MouseMove

If e.Button = MouseButtons.Left Then

Me.Location = New Point(Me.Left + e.X - \_xStart, Me.Top + e.Y - \_yStart)

End If

End Sub

Private Sub e\_closebutton\_click(sender As Object, e As EventArgs) Handles CloseButton.Click

Me.DialogResult = DialogResult.Cancel

Me.Dispose()

End Sub

Private Sub e\_loginbutton\_click(sender As Object, e As EventArgs) Handles LoginButton.Click

Dim username As String = Me.usernameTextbox.Text

Dim password As String = Me.passwordTextbox.Text

Dim pSource As String = cAccount.hash\_pword(password)

Dim response As Tuple(Of cAccount.login\_response, cAccount) = cAccount.login(username, pSource)

If response.Item1 = cAccount.login\_response.valid Then

End If

Select Case response.Item1

Case cAccount.login\_response.valid

InterfaceForm.e\_login\_successful(response.Item2)

Me.Close()

Me.Dispose()

Case cAccount.login\_response.username\_invalid

MessageBox.Show("Username is invalid. Please try again.", "Invalid login:")

Case cAccount.login\_response.password\_invalid

MessageBox.Show("Password is invalid. Please try again", "Invalid login:")

End Select

End Sub

End Class

Public Class SignupForm

Private WithEvents dp As cAccountDisplayPicture

#Region "drag-drop"

''drag-drop functionality''

Private \_xStart As Integer

Private \_yStart As Integer

Private Sub e\_form\_mousedown(sender As Object, e As MouseEventArgs) Handles TitleStripPanel.MouseDown, MyBase.MouseDown, TitleLabel.MouseDown

\_xStart = e.X : \_yStart = e.Y

End Sub

Private Sub e\_form\_dragdrop(sender As Object, e As MouseEventArgs) Handles TitleStripPanel.MouseMove, MyBase.MouseMove, TitleLabel.MouseMove

If e.Button = MouseButtons.Left Then

Me.Location = New Point(Me.Left + e.X - \_xStart, Me.Top + e.Y - \_yStart)

End If

End Sub

#End Region

Private Sub e\_closebutton\_click(sender As Object, e As EventArgs) Handles CloseButton.Click

Me.DialogResult = DialogResult.Cancel

Me.Dispose()

End Sub

Private Sub e\_load(sender As Object, e As EventArgs) Handles MyBase.Load

dp = New cAccountDisplayPicture

Me.Controls.Add(dp)

dp.Size = New Size(100, 100)

dp.Location = New Point(CInt((Me.Width / 2) - 50), 40)

dp.Cursor = Cursors.Hand

Me.DialogResult = DialogResult.None

End Sub

Private Sub e\_dp\_clicked(sender As Object, e As MouseEventArgs) Handles dp.MouseClick

If (e.Button = MouseButtons.Left) Then

Using dialog As OpenFileDialog = New OpenFileDialog

dialog.Filter = "JPEG Files (\*.jpg)|\*.jpg|PNG Files (\*.png)|\*.png|All Files (\*.\*)|\*.\*"

If (dialog.ShowDialog = DialogResult.OK) Then

Dim path As String = dialog.FileName

If (IO.File.Exists(path)) Then

Try

Dim b As Bitmap = New Bitmap(path)

Me.dp.Image = b

Catch ex As Exception

MsgBox("Image could not be loaded, please try a different image.")

End Try

End If

End If

End Using

ElseIf (e.Button = MouseButtons.Right) Then

Me.dp.reset\_image()

End If

End Sub

Private Function valid\_username() As Boolean

If Me.username\_textbox.Text.Length > 3 AndAlso Me.username\_textbox.Text.Length < 21 Then

Return True

End If

Return False

End Function

Private Function valid\_password() As Boolean

If password\_textbox.Text.Length > 4 AndAlso password\_textbox.Text.Length < 21 Then

If password\_textbox.Text = password\_confirm\_textbox.Text Then

Return True

Else

Return False

End If

End If

Return False

End Function

Private Sub e\_signup\_click(sender As Object, e As EventArgs) Handles create\_button.Click

Dim errormsg As String = Nothing

If valid\_username() AndAlso valid\_password() Then

Dim uname As String = Me.username\_textbox.Text : Dim phash As String = cAccount.hash\_pword(Me.password\_textbox.Text)

Dim elo As Integer = CInt(Me.elo\_nud.Value) : Dim pic As Bitmap = New Bitmap(Me.dp.Image)

cAccount.generate\_account(uname, phash, elo, pic)

Me.DialogResult = DialogResult.Yes

Me.Close()

Me.Dispose()

Else

MsgBox("ERROR:" & vbCrLf & "Please ensure your password is between 5-20 characters, and your username is between 4-20 characters.")

End If

End Sub

End Class

<Serializable()>

Public Class cAttackMap

Private \_board As cBoard

Private \_attackers(63) As List(Of cPiece) 'an attacker is defined as a piece that could attack a tile, even if pinned.

Private \_whiteAttackCount(63) As Byte

Private \_blackAttackCount(63) As Byte

#Region "Mutators and Accessors"

Public Function getAttackers() As List(Of cPiece)()

Return Me.\_attackers

End Function

Public Function getAttackCount(ALLIANCE As Alliance) As Byte()

If ALLIANCE = chess.Alliance.White Then Return Me.\_whiteAttackCount Else Return Me.\_blackAttackCount

End Function

#End Region

Public Sub New(BOARD As cBoard)

For i = 0 To 63

\_attackers(i) = New List(Of cPiece)

Next

Me.\_board = BOARD.get\_deep\_clone(Of cBoard)(BOARD)

Me.update\_attackers\_and\_count()

End Sub

Public Sub update\_map(BOARD As cBoard)

Me.\_board = Me.\_board.get\_deep\_clone(Of cBoard)(BOARD)

Me.update\_attackers\_and\_count()

End Sub

Private Sub update\_attackers\_and\_count()

For Each piece As cPiece In Me.\_board.get\_pieces(Alliance.White)

If piece.calc\_pseudo(Me.\_board) IsNot Nothing Then

For Each move As sMove In piece.calc\_pseudo(Me.\_board)

Me.\_attackers(move.dest).Add(piece)

Me.\_whiteAttackCount(move.dest) += CByte(1)

Next

End If

Next

For Each piece As cPiece In Me.\_board.get\_pieces(Alliance.Black)

If piece.calc\_pseudo(Me.\_board) IsNot Nothing Then

For Each move As sMove In piece.calc\_pseudo(Me.\_board)

Me.\_attackers(move.dest).Add(piece)

Me.\_blackAttackCount(move.dest) += CByte(1)

Next

End If

Next

End Sub

End Class

Imports System.IO

Imports System.Runtime.Serialization.Formatters.Binary

<Serializable()>

Public Class cBoard

Private \_tiles(63) As cTile

Private \_white\_pseudo As sMove()

Private \_black\_pseudo As sMove()

Private \_white\_pieces As cPiece()

Private \_black\_pieces As cPiece()

Private \_legal\_moves As sMove()

Private \_whose\_turn As Alliance

Private \_state As GameState

Private \_move\_list As LinkedList(Of sMove)

Private \_move\_list\_string As LinkedList(Of String)

'MANAGEMENT

Private \_en\_passent\_coord As Byte

Private \_half\_move\_timer As Integer

Private \_ply\_counter As Integer

#Region "Mutators and Accessors"

Public Function get\_tiles() As cTile()

Return Me.\_tiles

End Function

Public Function get\_tile(COORDINATE As Byte) As cTile

Return Me.\_tiles(COORDINATE)

End Function

Public Function get\_whose\_turn() As Alliance

Return Me.\_whose\_turn

End Function

Public Sub set\_whose\_turn(VALUE As Alliance)

Me.\_whose\_turn = VALUE

End Sub

Public Function get\_legal\_moves() As sMove()

Return Me.\_legal\_moves

End Function

Public Function get\_pseudo\_legal(ALLIANCE As Alliance) As sMove()

If ALLIANCE = Alliance.White Then Return Me.\_white\_pseudo Else Return Me.\_black\_pseudo

End Function

Public Function get\_state() As GameState

Return Me.\_state

End Function

Public Function get\_enpassent() As Byte

Return Me.\_en\_passent\_coord

End Function

Public Function get\_pieces(ALLIANCE As Alliance) As cPiece()

If ALLIANCE = Alliance.White Then Return Me.\_white\_pieces Else Return Me.\_black\_pieces

End Function

Public Function get\_move\_list\_string() As LinkedList(Of String)

Return Me.\_move\_list\_string

End Function

Public Function get\_move\_list() As LinkedList(Of sMove)

Return Me.\_move\_list

End Function

Public Function get\_half\_move\_timer() As Integer

Return Me.\_half\_move\_timer

End Function

#End Region

#Region "Constructor"

Public Sub New()

Me.\_move\_list = New LinkedList(Of sMove)

Me.\_move\_list\_string = New LinkedList(Of String)

Me.init\_tiles()

Me.initialize\_board()

Me.\_white\_pieces = Me.update\_pieces(Alliance.White)

Me.\_black\_pieces = Me.update\_pieces(Alliance.Black)

Me.\_white\_pseudo = Me.calc\_pseudo\_moves(Alliance.White)

Me.\_black\_pseudo = Me.calc\_pseudo\_moves(Alliance.Black)

Me.\_legal\_moves = Me.calc\_legal()

End Sub

#End Region

Public Sub set\_position(FEN As String)

'Forsyth-Edwards Notation FORMAT:

'PIECE LOCATIONS/EMPTY SQUARES : WHOS TURN : CASTLING RIGHTS : EN-PASSENT TARGET SQUARE : HALFMOVE CLOCK : FULLMOVE NUMBER

Dim counter As Integer = 0

For Each tile As cTile In Me.\_tiles

tile.set\_piece(Nothing)

Next

Dim endOf As Integer

For i = 0 To FEN.Length - 1

If FEN(i) = CChar(" ") Then

endOf = i

Exit For

End If

If FEN(i) = CChar("/") Then

ElseIf Asc(FEN(i)) >= 48 AndAlso Asc(FEN(i)) <= 57 Then

counter += Val(FEN(i))

Else

Me.\_tiles(counter).set\_piece(generate\_piece(FEN(i), CByte(counter)))

counter += 1

End If

Next

If FEN(endOf + 1) = "w" Then

Me.\_whose\_turn = Alliance.White

Else

Me.\_whose\_turn = Alliance.Black

End If

End Sub

Public Function generate\_piece(C As Char, COORDINATE As Byte) As cPiece

Dim temp As Char = C

Dim alliance As Alliance

If LCase(C) = temp Then alliance = Alliance.Black Else alliance = Alliance.White

Select Case LCase(C)

Case CChar("r")

Return New cRook(alliance, COORDINATE)

Case CChar("b")

Return New cBishop(alliance, COORDINATE)

Case CChar("q")

Return New cQueen(alliance, COORDINATE)

Case CChar("k")

Return New cKing(alliance, COORDINATE)

Case CChar("p")

Return New cPawn(alliance, COORDINATE)

Case CChar("n")

Return New cKnight(alliance, COORDINATE)

End Select

Return Nothing

End Function

Public Sub init\_tiles()

Dim count As Integer = 0

Dim isLightSquare As Boolean = False

For i = 0 To 63

count += 1

If count = 9 Then

count = 1

Else

isLightSquare = Not (isLightSquare)

End If

Me.\_tiles(i) = New cTile(CByte(i), isLightSquare)

Next

Me.\_en\_passent\_coord = 255

End Sub

Public Function find\_king(ALLIANCE As Alliance) As cKing

If ALLIANCE = Alliance.White Then

For Each piece As cPiece In Me.get\_pieces(Alliance.White)

If piece.get\_title = Chessman.King Then Return CType(piece, cKing)

Next

Else

For Each piece As cPiece In Me.get\_pieces(Alliance.Black)

If piece.get\_title = Chessman.King Then Return CType(piece, cKing)

Next

End If

Return Nothing

End Function

''' <summary>

''' Will return an updated cPiece array of all of the pieces currently on the instance of board from which it is called.

''' </summary>

''' <param name="ALLIANCE">Alliance of the set of pieces intended to be updated.</param>

''' <returns>Hey</returns>

Public Function update\_pieces(ALLIANCE As Alliance) As cPiece()

Dim temp As cPiece() = Nothing

Dim counter As Integer = 0

For Each t As cTile In \_tiles

If t.is\_occupied AndAlso t.get\_piece.get\_alliance = ALLIANCE Then

ReDim Preserve temp(counter)

temp(counter) = t.get\_piece

counter += 1

End If

Next

Return temp

End Function

''' <summary>

''' Creates and returns a bit-wise clone of the input board, after eliminating references.

''' </summary>

''' <typeparam name="cBoard"></typeparam>

''' <param name="orig"></param>

''' <returns></returns>

Public Function get\_deep\_clone(Of cBoard)(ByRef orig As cBoard) As cBoard

'SHALLOW CLONE VS DEEP CLONE

'A shallow clone will be a bit-wise copy of an object, with all of the same values that the object had, however if there is a reference to another object in the original object, then only the reference

' is copied. With a deep clone, it is still a bit-wise copy of the object, however when a reference is encountered in that object, that object is cloned in order create a new clone with all of the same values

' as that object within the object, this allows a deep clone to be edited without affecting memory locations that may be being used by other objects that you do not intend to edit.

'if the object is a NULL reference, then return that objects 'nothing'.

If (Object.ReferenceEquals(orig, Nothing)) Then Return Nothing

Dim formatter As New BinaryFormatter()

Dim stream As New MemoryStream()

'the serialize function will serialize the object to the inputted stream.

formatter.Serialize(stream, orig)

'this then sets the position within the stream "stream" to the beginning of the stream so that the entire stream can be deserialized and returned at the end of the function.

stream.Seek(0, SeekOrigin.Begin)

Return CType(formatter.Deserialize(stream), cBoard)

End Function

''' <summary>

''' Updates the board to represent the standard chess starting position.

''' </summary>

Private Sub initialize\_board()

'black major pieces

\_tiles(0).set\_piece(New cRook(Alliance.Black, 0))

\_tiles(1).set\_piece(New cKnight(Alliance.Black, 1))

\_tiles(2).set\_piece(New cBishop(Alliance.Black, 2))

\_tiles(3).set\_piece(New cQueen(Alliance.Black, 3))

\_tiles(4).set\_piece(New cKing(Alliance.Black, 4))

\_tiles(5).set\_piece(New cBishop(Alliance.Black, 5))

\_tiles(6).set\_piece(New cKnight(Alliance.Black, 6))

\_tiles(7).set\_piece(New cRook(Alliance.Black, 7))

'black pawns

For i = 8 To 15

\_tiles(i).set\_piece(New cPawn(Alliance.Black, CByte(i)))

Next

'white pawns

For i = 48 To 55

\_tiles(i).set\_piece(New cPawn(Alliance.White, CByte(i)))

Next

'white major pieces

\_tiles(56).set\_piece(New cRook(Alliance.White, 56))

\_tiles(57).set\_piece(New cKnight(Alliance.White, 57))

\_tiles(58).set\_piece(New cBishop(Alliance.White, 58))

\_tiles(59).set\_piece(New cQueen(Alliance.White, 59))

\_tiles(60).set\_piece(New cKing(Alliance.White, 60))

\_tiles(61).set\_piece(New cBishop(Alliance.White, 61))

\_tiles(62).set\_piece(New cKnight(Alliance.White, 62))

\_tiles(63).set\_piece(New cRook(Alliance.White, 63))

End Sub

''' <summary>

''' Returns the pseudo-legal moves of pieces of the specified alliance.

''' </summary>

''' <param name="ALLIANCE"></param>

Private Function calc\_pseudo\_moves(ALLIANCE As Alliance) As sMove()

Dim pieceSet As cPiece()

If ALLIANCE = Alliance.White Then pieceSet = Me.\_white\_pieces Else pieceSet = Me.\_black\_pieces

Dim temp As sMove() = Nothing

Dim counter As Integer = 0

For Each piece As cPiece In pieceSet

If piece.calc\_pseudo(Me) IsNot Nothing Then

For Each move As sMove In piece.calc\_pseudo(Me)

ReDim Preserve temp(counter)

temp(counter) = move

counter += 1

Next

End If

Next

Return temp

End Function

''' <summary>

''' Returns moves which are allowed by each pieces normal moves, taking pins into account.

''' </summary>

Private Function calc\_legal() As sMove()

Dim moveSet As sMove()

Dim temp As sMove() = Nothing

Dim counter As Integer = 0

If Me.\_whose\_turn = Alliance.White Then moveSet = Me.\_white\_pseudo Else moveSet = Me.\_black\_pseudo

For Each move As sMove In moveSet

Dim x As New cGhostBoard(Me, move)

If x.is\_legal Then

ReDim Preserve temp(counter)

temp(counter) = move

counter += 1

End If

Next

Return temp

End Function

Public Function is\_in\_check(ALLIANCE As Alliance) As Boolean

Dim moveSet As sMove()

Dim king As cKing

If ALLIANCE = Alliance.White Then moveSet = Me.\_black\_pseudo Else moveSet = Me.\_white\_pseudo

If ALLIANCE = Alliance.White Then king = Me.find\_king(Alliance.White) Else king = Me.find\_king(Alliance.Black)

If moveSet Is Nothing Then Return False 'temp

For Each move As sMove In moveSet

If move.dest = king.get\_coordinate Then Return True

Next

Return False

End Function

Public Function is\_legal\_move(MOVE As sMove) As Boolean

If Me.\_legal\_moves IsNot Nothing AndAlso Me.\_legal\_moves.Contains(MOVE) Then Return True Else Return False

End Function

''' <summary>

''' Updates the current instance of cBoard to reflect the move attempted, if the move is not legal, it will not be made.

''' </summary>

''' <param name="MOVE"></param>

''' <param name="FROM\_TRANSITION"></param>

Public Sub make\_move(MOVE As sMove, Optional FROM\_TRANSITION As Boolean = False)

If MOVE.ogCoord = 4 AndAlso MOVE.dest = 2 Then MsgBox("TEST")

Dim movingPiece As cPiece = Me.get\_tile(MOVE.ogCoord).get\_piece

Dim targetTile As cTile = Me.get\_tile(MOVE.dest)

Dim wasCapture As Boolean

wasCapture = Me.\_tiles(MOVE.dest).is\_occupied

If FROM\_TRANSITION Then

Dim castlingTiles As Byte() = {58, 62, 2, 6}

If movingPiece.get\_title = Chessman.Pawn AndAlso targetTile.get\_coordinate = Me.\_en\_passent\_coord Then 'en passent capture

sMove.enpassent(MOVE, Me)

ElseIf movingPiece.get\_title = Chessman.Pawn AndAlso ((targetTile.get\_coordinate \ 8 + 1 = 1) AndAlso movingPiece.get\_alliance = Alliance.White) Then 'black promotion

sMove.promotion(MOVE, Me)

ElseIf movingPiece.get\_title = Chessman.Pawn AndAlso (targetTile.get\_coordinate \ 8 + 1 = 8) AndAlso movingPiece.get\_alliance = Alliance.Black Then 'white promotion

sMove.promotion(MOVE, Me)

ElseIf movingPiece.get\_title = Chessman.King AndAlso Not CType(movingPiece, cKing).get\_moved AndAlso castlingTiles.Contains(MOVE.dest) Then 'castling

sMove.castle(MOVE, Me)

Else 'normal move66

sMove.regular(MOVE, Me)

End If

If movingPiece.get\_title = Chessman.King Then CType(movingPiece, cKing).set\_moved(True)

Me.\_move\_list\_string.AddLast(MOVE.ToString)

Me.update\_board\_members(wasCapture, movingPiece, FROM\_TRANSITION)

Else

Dim castlingTiles As Byte() = {58, 62, 2, 6}

If Me.\_legal\_moves.Contains(MOVE) Then

'assume the move is added to the move list by the GUI

If movingPiece.get\_title = Chessman.Pawn AndAlso targetTile.get\_coordinate = Me.\_en\_passent\_coord Then 'en passent capture

Me.\_en\_passent\_coord = 255

sMove.enpassent(MOVE, Me)

wasCapture = True

ElseIf movingPiece.get\_title = Chessman.Pawn AndAlso ((targetTile.get\_coordinate \ 8 + 1 = 1) AndAlso movingPiece.get\_alliance = Alliance.White) Then 'black promotion

sMove.promotion(MOVE, Me)

ElseIf movingPiece.get\_title = Chessman.Pawn AndAlso (targetTile.get\_coordinate \ 8 + 1 = 8) AndAlso movingPiece.get\_alliance = Alliance.Black Then 'white promotion

sMove.promotion(MOVE, Me)

ElseIf movingPiece.get\_title = Chessman.King AndAlso Not CType(movingPiece, cKing).get\_moved AndAlso castlingTiles.Contains(MOVE.dest) Then 'castling

sMove.castle(MOVE, Me)

Else 'normal move66

sMove.regular(MOVE, Me)

End If

'update the en-passent tile

If movingPiece.get\_title = Chessman.Pawn Then

If 9 - (MOVE.ogCoord \ 8 + 1) = 2 Then

If MOVE.dest = MOVE.ogCoord - 16 Then

Me.\_en\_passent\_coord = CByte(MOVE.ogCoord - 8)

Else

Me.\_en\_passent\_coord = 255

End If

End If

If 9 - (MOVE.ogCoord \ 8 + 1) = 7 Then

If MOVE.dest = MOVE.ogCoord + 16 Then

Me.\_en\_passent\_coord = CByte(MOVE.ogCoord + 8)

Else

Me.\_en\_passent\_coord = 255

End If

End If

Else

Me.\_en\_passent\_coord = 255

End If

If movingPiece.get\_title = Chessman.King Then CType(movingPiece, cKing).set\_moved(True)

If Not FROM\_TRANSITION Then Me.\_move\_list.AddLast(MOVE)

'updates all members of the board that need to be updated after a move.

Me.update\_board\_members(wasCapture, movingPiece, FROM\_TRANSITION)

End If

End If

End Sub

''' <summary>

''' This updates all necessary members of board after a move is made.

''' </summary>

''' <param name="WAS\_CAPTURE"></param>

''' <param name="MOVED\_PIECE"></param>

''' <param name="FROM\_TRANSITION"></param>

Private Sub update\_board\_members(WAS\_CAPTURE As Boolean, MOVED\_PIECE As cPiece, FROM\_TRANSITION As Boolean)

If WAS\_CAPTURE Then

Me.\_white\_pieces = Me.update\_pieces(Alliance.White)

Me.\_black\_pieces = Me.update\_pieces(Alliance.Black)

Debug.WriteLineIf(FROM\_TRANSITION = False, "Move was recorded as a capture.")

End If

If Me.\_whose\_turn = Alliance.White Then

Me.\_whose\_turn = Alliance.Black

If Not WAS\_CAPTURE AndAlso MOVED\_PIECE.get\_title <> Chessman.Pawn Then \_half\_move\_timer += CByte(1)

Else

Me.\_whose\_turn = Alliance.White

\_ply\_counter += 1

End If

Me.\_white\_pseudo = Me.calc\_pseudo\_moves(Alliance.White)

Me.\_black\_pseudo = Me.calc\_pseudo\_moves(Alliance.Black)

If Not FROM\_TRANSITION Then Me.\_legal\_moves = Me.calc\_legal()

Me.\_state = calculate\_board\_state()

If Not FROM\_TRANSITION Then Me.show\_debug\_report()

End Sub

Public Function calculate\_board\_state() As GameState

If Me.\_legal\_moves Is Nothing Then

If Me.is\_in\_check(Alliance.Black) Then

Dim temp As String = Me.\_move\_list\_string.Last.Value

Me.\_move\_list\_string.Last.Value = temp & "#"

Return GameState.BlackMated

ElseIf Me.is\_in\_check(Alliance.White) Then

Dim temp As String = Me.\_move\_list\_string.Last.Value

Me.\_move\_list\_string.Last.Value = temp & "#"

Return GameState.WhiteMated

Else

Return GameState.Stalemate

End If

End If

Return GameState.ongoing

End Function

Public Function is\_game\_over() As Boolean

With (Me)

If .\_state <> GameState.ongoing Then Return True Else Return False

End With

End Function

Public Sub show\_debug\_report()

Try

Debug.WriteLine("##################################################")

Debug.WriteLine("##################################################")

Debug.WriteLine("##################################################")

Debug.WriteLine("##################################################")

Debug.WriteLine("##################################################")

Debug.WriteLine("##################################################")

Debug.WriteLine("")

Debug.WriteLine("")

Dim ctr As Integer = 1

For i = 0 To 63

If ctr > 8 Then

Debug.Write(vbCrLf)

ctr = 1

End If

If Me.\_tiles(i).is\_occupied Then

Debug.Write("[" & Me.\_tiles(i).get\_piece.get\_char & "]")

Else

Debug.Write("[ ]")

End If

ctr += 1

Next

Debug.Write(vbCrLf & vbCrLf)

Debug.WriteLine("BOARD REPORT")

Debug.WriteLine("-----------------")

If Me.\_en\_passent\_coord <> 255 Then

Debug.WriteLine("En-passent coordinate: " & Me.\_tiles(Me.\_en\_passent\_coord).get\_file & Me.\_tiles(Me.\_en\_passent\_coord).get\_rank)

Else

Debug.WriteLine("En-passent coordinate: 255 [no en-passent]")

End If

If Me.\_legal\_moves IsNot Nothing Then Debug.WriteLine("Legal move count: " & Me.\_legal\_moves.Length)

Debug.WriteLine("Last move: " & Me.\_move\_list\_string.Last.Value)

Debug.WriteLine("Whos turn: " & Me.\_whose\_turn)

Debug.WriteLine("")

Debug.WriteLine("")

Catch ex As Exception

End Try

End Sub

End Class

Public Class cGhostBoard

Private \_move As sMove

Private \_board As cBoard

Private \_legal As Boolean

Public Function is\_legal() As Boolean

Return Me.\_legal

End Function

Public Function get\_board() As cBoard

Return Me.\_board

End Function

Public Sub New(OG\_BOARD As cBoard, MOVE As sMove)

Me.\_move = MOVE

Me.\_board = OG\_BOARD.get\_deep\_clone(Of cBoard)(OG\_BOARD)

Me.\_board.make\_move(MOVE, True)

Me.\_legal = calc\_legality()

End Sub

Public Function calc\_legality() As Boolean

With (Me.\_board)

If .is\_in\_check(Alliance.White) AndAlso .get\_whose\_turn = Alliance.Black Then

Return False

End If

If .is\_in\_check(Alliance.Black) AndAlso .get\_whose\_turn = Alliance.White Then

Return False

End If

End With

Return True

End Function

End Class

<Serializable()>

Public Class cTile

Private \_isLightSquare As Boolean

Private \_coordinate As Byte

Private \_piece As cPiece

#Region "Mutators and Accessors"

Public Function get\_coordinate() As Byte

Return Me.\_coordinate

End Function

Public Function get\_piece() As cPiece

Return Me.\_piece

End Function

Public Sub set\_piece(VALUE As cPiece)

Me.\_piece = VALUE

End Sub

Public Function get\_is\_lightsquare() As Boolean

Return Me.\_isLightSquare

End Function

#End Region

Public Sub New(COORDINATE As Byte, IS\_LIGHT\_SQUARE As Boolean)

If COORDINATE > 255 Then Throw New Exception("Too many tiles.")

Me.\_coordinate = COORDINATE

Me.\_isLightSquare = IS\_LIGHT\_SQUARE

End Sub

Public Function get\_file() As Char

Return LCase(Chr(64 + (Me.\_coordinate Mod 8 + 1)))

End Function

Public Function get\_rank() As Byte

Return CByte(9 - (Me.\_coordinate \ 8 + 1))

End Function

Public Function is\_occupied() As Boolean

If Me.\_piece Is Nothing Then Return False Else Return True

End Function

End Class

Public Enum GameState

ongoing

WhiteMated

BlackMated

Stalemate

Draw

End Enum

<Serializable()>

Public Class cBishop

Inherits cPiece

Public Sub New(ALLIANCE As Alliance, COORDINATE As Byte)

MyBase.New(ALLIANCE, COORDINATE)

Me.\_title = Chessman.Bishop

Me.\_value = 315

End Sub

Public Overrides Function get\_char() As Char

If Me.\_alliance = Alliance.White Then

Return CChar("B")

Else

Return CChar("b")

End If

End Function

Public Overrides Function calc\_pseudo(BOARD As cBoard) As sMove()

Dim possibleMoveOffsets As SByte() = {-9, -7, 7, 9}

Dim legalMoves() As sMove = Nothing

Dim counter As Integer = 0

For i = 0 To UBound(possibleMoveOffsets)

Dim targetCoord As Integer = Me.\_coordinate + possibleMoveOffsets(i)

While is\_valid\_tile(targetCoord, CByte(targetCoord - possibleMoveOffsets(i)), possibleMoveOffsets(i))

If BOARD.get\_tile(CByte(targetCoord)).is\_occupied Then

If BOARD.get\_tile(CByte(targetCoord)).get\_piece.get\_alliance <> Me.\_alliance Then

ReDim Preserve legalMoves(counter)

legalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoord))

counter += 1

End If

Exit While

Else

ReDim Preserve legalMoves(counter)

legalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoord))

counter += 1

End If

targetCoord += possibleMoveOffsets(i)

End While

Next

Return legalMoves

End Function

Public Function is\_valid\_tile(TARGET\_COORDIANTE As Integer, OG\_COORDINATE As Byte, OFFSET As Integer) As Boolean

Dim ogCol As Integer = (OG\_COORDINATE Mod 8 + 1)

Select Case ogCol

Case 1

If OFFSET = -9 Or OFFSET = 7 Then Return False

Case 8

If OFFSET = -7 Or OFFSET = 9 Then Return False

End Select

If TARGET\_COORDIANTE >= 0 And TARGET\_COORDIANTE <= 63 Then

Return True

End If

Return False

End Function

End Class

<Serializable()>

Public Class cKing

Inherits cPiece

Private \_moved As Boolean = False

Public Function get\_moved() As Boolean

Return Me.\_moved

End Function

Public Sub set\_moved(VALUE As Boolean)

Me.\_moved = VALUE

End Sub

Public Sub New(ALLIANCE As Alliance, COORDINATE As Byte)

MyBase.New(ALLIANCE, COORDINATE)

Me.\_title = Chessman.King

Me.\_value = 100000

End Sub

Public Overrides Function get\_char() As Char

If Me.\_alliance = Alliance.White Then

Return CChar("K")

Else

Return CChar("k")

End If

End Function

Public Overrides Function calc\_pseudo(BOARD As cBoard) As sMove()

Dim possibleMoveOffsets As SByte() = {-9, -8, -7, -1, 1, 7, 8, 9}

Dim legalMoves() As sMove = Nothing

Dim counter As Integer = 0

For i = 0 To UBound(possibleMoveOffsets)

Dim targetCoord As Integer = Me.\_coordinate + possibleMoveOffsets(i)

If is\_valid\_tile(targetCoord, Me.\_coordinate, possibleMoveOffsets(i)) Then

If Not BOARD.get\_tile(CByte(targetCoord)).is\_occupied Then

ReDim Preserve legalMoves(counter)

legalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoord))

counter += 1

Else

'HANDLES CAPTURING

If BOARD.get\_tile(CByte(targetCoord)).get\_piece.get\_alliance <> Me.\_alliance Then

ReDim Preserve legalMoves(counter)

legalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoord))

counter += 1

End If

End If

End If

Next

If Not \_moved AndAlso Not BOARD.is\_in\_check(Me.\_alliance) Then

'CASTLING STUFF

Dim allowShort As Boolean = True

Dim allowLong As Boolean = True

Dim moveSet As sMove()

Dim longCoords As Byte()

Dim shortCoords As Byte()

If Me.\_alliance = Alliance.White Then

longCoords = {59, 58}

shortCoords = {61, 62}

moveSet = BOARD.get\_pseudo\_legal(Alliance.Black)

Else

longCoords = {2, 3}

shortCoords = {5, 6}

moveSet = BOARD.get\_pseudo\_legal(Alliance.White)

End If

'checks to see that the tiles are not occupied

For Each move As Byte In longCoords

If BOARD.get\_tile(move).is\_occupied Then allowLong = False

Next

For Each move As Byte In shortCoords

If BOARD.get\_tile(move).is\_occupied Then allowShort = False

Next

'checks to see if a tile the king will move through will put it in check

If moveSet IsNot Nothing Then

For Each move As sMove In moveSet

If longCoords.Contains(move.dest) Then

allowLong = False

ElseIf shortCoords.Contains(move.dest) Then

allowShort = False

End If

Next

End If

'checks to see that the rooks have not moved.

If Me.\_alliance = Alliance.White Then

If BOARD.get\_tiles()(63).is\_occupied Then

If TypeOf (BOARD.get\_tiles()(63).get\_piece) Is cRook AndAlso CType(BOARD.get\_tiles()(63).get\_piece, cRook).get\_moved Then

allowShort = False

ElseIf TypeOf (BOARD.get\_tiles()(63).get\_piece) IsNot cRook Then

allowShort = False

ElseIf Not (BOARD.get\_tiles()(63).is\_occupied) Then

allowShort = False

End If

Else

allowShort = False

End If

If BOARD.get\_tiles()(56).is\_occupied Then

If TypeOf (BOARD.get\_tiles()(56).get\_piece) Is cRook AndAlso CType(BOARD.get\_tiles()(56).get\_piece, cRook).get\_moved Then

allowLong = False

ElseIf TypeOf (BOARD.get\_tiles()(56).get\_piece) IsNot cRook Then

allowLong = False

ElseIf Not (BOARD.get\_tiles()(56).is\_occupied) Then

allowLong = False

End If

End If

ElseIf Me.\_alliance = Alliance.Black Then

If BOARD.get\_tiles()(7).is\_occupied Then

If TypeOf (BOARD.get\_tiles()(7).get\_piece) Is cRook AndAlso CType(BOARD.get\_tiles()(7).get\_piece, cRook).get\_moved Then

allowShort = False

ElseIf TypeOf (BOARD.get\_tiles()(7).get\_piece) IsNot cRook Then

allowShort = False

ElseIf Not (BOARD.get\_tiles()(7).is\_occupied) Then

allowShort = False

End If

Else

allowShort = False

End If

If BOARD.get\_tiles()(0).is\_occupied Then

If TypeOf (BOARD.get\_tiles()(0).get\_piece) Is cRook AndAlso CType(BOARD.get\_tiles()(0).get\_piece, cRook).get\_moved Then

allowLong = False

ElseIf TypeOf (BOARD.get\_tiles()(0).get\_piece) IsNot cRook Then

allowLong = False

ElseIf Not (BOARD.get\_tiles()(0).is\_occupied) Then

allowLong = False

End If

End If

End If

Select Case Me.\_alliance

Case Alliance.Black

If allowShort Then

ReDim Preserve legalMoves(counter)

legalMoves(counter) = New sMove(Me.\_coordinate, 6)

counter += 1

End If

If allowLong Then

ReDim Preserve legalMoves(counter)

legalMoves(counter) = New sMove(Me.\_coordinate, 2)

counter += 1

End If

Case Alliance.White

If allowShort Then

ReDim Preserve legalMoves(counter)

legalMoves(counter) = New sMove(Me.\_coordinate, 62)

counter += 1

End If

If allowLong Then

ReDim Preserve legalMoves(counter)

legalMoves(counter) = New sMove(Me.\_coordinate, 58)

counter += 1

End If

End Select

End If

Return legalMoves

End Function

Private Function is\_valid\_tile(TARGET\_COORDIANTE As Integer, OG\_COORDINATE As Byte, OFFSET As Integer) As Boolean

Dim ogCol As Integer = (OG\_COORDINATE Mod 8 + 1)

Select Case ogCol

Case 1

If OFFSET = -9 Or OFFSET = -1 Or OFFSET = 7 Then Return False

Case 8

If OFFSET = 9 Or OFFSET = 1 Or OFFSET = -7 Then Return False

End Select

If TARGET\_COORDIANTE >= 0 And TARGET\_COORDIANTE <= 63 Then

Return True

End If

Return False

End Function

End Class

<Serializable()>

Public Class cKnight

Inherits cPiece

Public Sub New(ALLIANCE As Alliance, COORDINATE As Byte)

MyBase.New(ALLIANCE, COORDINATE)

Me.\_title = Chessman.Knight

Me.\_value = 300

End Sub

Public Overrides Function get\_char() As Char

If Me.\_alliance = Alliance.White Then

Return CChar("N")

Else

Return CChar("n")

End If

End Function

Public Overrides Function calc\_pseudo(BOARD As cBoard) As sMove()

Dim offsets As SByte() = {-17, -15, -10, -6, 6, 10, 15, 17}

Dim pseudoLegalMoves() As sMove = Nothing

Dim counter As Integer = 0

For Each offset As SByte In offsets

Dim targetCoord As Integer = Me.\_coordinate + offset

If is\_valid\_tile(targetCoord) Then

If Not BOARD.get\_tile(CByte(targetCoord)).is\_occupied Then

ReDim Preserve pseudoLegalMoves(counter)

pseudoLegalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoord))

counter += 1

Else

If BOARD.get\_tile(CByte(targetCoord)).get\_piece.get\_alliance <> Me.\_alliance Then

ReDim Preserve pseudoLegalMoves(counter)

pseudoLegalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoord))

counter += 1

End If

End If

End If

Next

Return pseudoLegalMoves

End Function

Public Function is\_valid\_tile(TARGET\_COORDINATE As Integer) As Boolean

If TARGET\_COORDINATE <= 63 And TARGET\_COORDINATE >= 0 Then

If Not Math.Abs((Me.\_coordinate Mod 8 + 1) - (TARGET\_COORDINATE Mod 8 + 1)) > 2 Then

Return True

End If

End If

Return False

End Function

End Class

<Serializable()>

Public Class cPawn

Inherits cPiece

Public Sub New(ALLIANCE As Alliance, COORDINATE As Byte)

MyBase.New(ALLIANCE, COORDINATE)

Me.\_title = Chessman.Pawn

Me.\_value = 100

End Sub

Public Overrides Function get\_char() As Char

If Me.\_alliance = Alliance.White Then

Return CChar("P")

Else

Return CChar("p")

End If

End Function

Public Overrides Function calc\_pseudo(BOARD As cBoard) As sMove()

Dim possibleMoveOffsets() As Integer = {7, 8, 9}

Dim multiplier As Integer

Dim legalMoves() As sMove = Nothing

Dim counter As Integer = 0

If Me.\_alliance = Alliance.White Then multiplier = -1 Else multiplier = 1

For i = 0 To UBound(possibleMoveOffsets)

Dim targetCoordinate As Integer = Me.\_coordinate + possibleMoveOffsets(i) \* multiplier

If is\_valid\_tile(targetCoordinate, Me.\_coordinate, (possibleMoveOffsets(i) \* multiplier)) Then

'HANDLES CAPTURE MOVES

If possibleMoveOffsets(i) = 7 Or possibleMoveOffsets(i) = 9 Then

If BOARD.get\_tile(CByte(targetCoordinate)).is\_occupied AndAlso

BOARD.get\_tile(CByte(targetCoordinate)).get\_piece.get\_alliance <> Me.\_alliance Then

ReDim Preserve legalMoves(counter)

legalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoordinate))

counter += 1

ElseIf BOARD.get\_tile(CByte(targetCoordinate)).get\_coordinate = BOARD.get\_enpassent Then

'HANDLES CAPTURING EN PASSENT

ReDim Preserve legalMoves(counter)

legalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoordinate))

counter += 1

End If

Else

'HANDLES FORWARDS MOVES

If Not BOARD.get\_tile(CByte(targetCoordinate)).is\_occupied Then

ReDim Preserve legalMoves(counter)

legalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoordinate))

counter += 1

'HANDLES THRUSTING

targetCoordinate = targetCoordinate + (8 \* multiplier)

If is\_valid\_tile(targetCoordinate, Me.\_coordinate, 16 \* multiplier) Then

If ((Me.\_coordinate \ 8 + 1) = 7 AndAlso Me.\_alliance = Alliance.White AndAlso Not BOARD.get\_tile(CByte(targetCoordinate)).is\_occupied) \_

Or ((Me.\_coordinate \ 8 + 1) = 2 AndAlso Me.\_alliance = Alliance.Black AndAlso Not BOARD.get\_tile(CByte(targetCoordinate)).is\_occupied) Then

ReDim Preserve legalMoves(counter)

legalMoves(counter) = New sMove(Me.\_coordinate, CByte((targetCoordinate)))

counter += 1

End If

End If

End If

End If

End If

Next

Return legalMoves

End Function

Public Function is\_valid\_tile(TARGET\_COORDIANTE As Integer, OG\_COORDINATE As Byte, OFFSET As Integer) As Boolean

Dim ogCol As Integer = (OG\_COORDINATE Mod 8 + 1)

Select Case ogCol

Case 1

If OFFSET = -9 Or OFFSET = 7 Then Return False

Case 8

If OFFSET = 9 Or OFFSET = -7 Then Return False

End Select

If TARGET\_COORDIANTE >= 0 And TARGET\_COORDIANTE <= 63 Then

Return True

End If

Return False

End Function

End Class

Imports System.IO

Imports System.Runtime.Serialization.Formatters.Binary

<Serializable()>

Public MustInherit Class cPiece

Protected \_title As Chessman

Protected \_coordinate As Byte

Protected \_alliance As Alliance

Protected \_value As Integer

Public Sub New()

End Sub

Public Sub New(ALLIANCE As Alliance, COORDINATE As Byte)

With (Me)

.\_alliance = ALLIANCE

.\_coordinate = COORDINATE

End With

End Sub

Public MustOverride Function get\_char() As Char

Public MustOverride Function calc\_pseudo(BOARD As cBoard) As sMove()

#Region "Mutators and Accessors"

Public Function get\_title() As Chessman

Return Me.\_title

End Function

Public Function get\_coordinate() As Byte

Return Me.\_coordinate

End Function

Public Sub set\_coordinate(VALUE As Byte)

Me.\_coordinate = VALUE

End Sub

Public Function get\_alliance() As Alliance

Return Me.\_alliance

End Function

Public Function get\_value() As Integer

Return Me.\_value

End Function

#End Region

End Class

<Serializable()>

Public Class cQueen

Inherits cPiece

Public Sub New(ALLIANCE As Alliance, COORDINATE As Byte)

MyBase.New(ALLIANCE, COORDINATE)

Me.\_title = Chessman.Queen

Me.\_value = 900

End Sub

Public Overrides Function get\_char() As Char

If Me.\_alliance = Alliance.White Then

Return CChar("Q")

Else

Return CChar("q")

End If

End Function

Public Overrides Function calc\_pseudo(BOARD As cBoard) As sMove()

Dim possibleMoveOffsets As SByte() = {-8, -1, 1, 8, -9, -7, 7, 9}

Dim legalMoves() As sMove = Nothing

Dim counter As Integer = 0

For i = 0 To UBound(possibleMoveOffsets)

Dim targetCoord As Integer = Me.\_coordinate + possibleMoveOffsets(i)

While is\_valid\_tile(targetCoord, CByte(targetCoord - possibleMoveOffsets(i)), possibleMoveOffsets(i))

If BOARD.get\_tile(CByte(targetCoord)).is\_occupied Then

If BOARD.get\_tile(CByte(targetCoord)).get\_piece.get\_alliance <> Me.\_alliance Then

ReDim Preserve legalMoves(counter)

legalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoord))

counter += 1

End If

Exit While

Else

ReDim Preserve legalMoves(counter)

legalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoord))

counter += 1

End If

targetCoord += possibleMoveOffsets(i)

End While

Next

Return legalMoves

End Function

Private Function is\_valid\_tile(TARGET\_COORDIANTE As Integer, OG\_COORDINATE As Byte, OFFSET As Integer) As Boolean

Dim ogCol As Integer = (OG\_COORDINATE Mod 8 + 1)

Select Case ogCol

Case 1

If OFFSET = -1 Or OFFSET = -9 Or OFFSET = 7 Then Return False

Case 8

If OFFSET = 1 Or OFFSET = -7 Or OFFSET = 9 Then Return False

End Select

If TARGET\_COORDIANTE >= 0 And TARGET\_COORDIANTE <= 63 Then

Return True

End If

Return False

End Function

End Class

<Serializable()>

Public Class cRook

Inherits cPiece

Private \_moved As Boolean = False

Public Function get\_moved() As Boolean

Return Me.\_moved

End Function

Public Sub set\_moved(VALUE As Boolean)

Me.\_moved = VALUE

MsgBox("MOVED")

End Sub

Public Sub New(ALLIANCE As Alliance, COORDINATE As Byte)

MyBase.New(ALLIANCE, COORDINATE)

Me.\_title = Chessman.Rook

Me.\_value = 500

End Sub

Public Overrides Function get\_char() As Char

If Me.\_alliance = Alliance.White Then

Return CChar("R")

Else

Return CChar("r")

End If

End Function

Public Overrides Function calc\_pseudo(BOARD As cBoard) As sMove()

Dim possibleMoveOffsets As SByte() = {-8, -1, 1, 8}

Dim legalMoves() As sMove = Nothing

Dim counter As Integer = 0

For i = 0 To UBound(possibleMoveOffsets)

Dim targetCoord As Integer = Me.\_coordinate + possibleMoveOffsets(i)

While is\_valid\_tile(targetCoord, CByte(targetCoord - possibleMoveOffsets(i)), possibleMoveOffsets(i))

If BOARD.get\_tile(CByte(targetCoord)).is\_occupied Then

If BOARD.get\_tile(CByte(targetCoord)).get\_piece.get\_alliance <> Me.\_alliance Then

ReDim Preserve legalMoves(counter)

legalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoord))

counter += 1

End If

Exit While

Else

ReDim Preserve legalMoves(counter)

legalMoves(counter) = New sMove(Me.\_coordinate, CByte(targetCoord))

counter += 1

End If

targetCoord += possibleMoveOffsets(i)

End While

Next

Return legalMoves

End Function

Public Function is\_valid\_tile(TARGET\_COORDIANTE As Integer, OG\_COORDINATE As Byte, OFFSET As Integer) As Boolean

Dim ogCol As Integer = (OG\_COORDINATE Mod 8 + 1)

Select Case ogCol

Case 1

If OFFSET = -1 Then Return False

Case 8

If OFFSET = 1 Then Return False

End Select

If TARGET\_COORDIANTE >= 0 And TARGET\_COORDIANTE <= 63 Then

Return True

End If

Return False

End Function

End Class

Public Enum Alliance

White

Black

End Enum

Public Enum Chessman

Pawn

Knight

Bishop

Rook

Queen

King

End Enum

Imports System.Threading

<Serializable()>

Public Class cClock

Public Event time\_up(sender As cClock)

Public game As cGame

Public count\_thread As Thread

Public \_time As Double 'in seconds.

Public \_increment As Integer 'in seconds.

Private \_counting As Boolean

#Region "Mutators and Acessors"

Public Function getCounting() As Boolean

Return Me.\_counting

End Function

Public Function getTime() As Double

Return Me.\_time

End Function

Public Function getIncrement() As Integer

Return Me.\_increment

End Function

#End Region

''' <summary>

''' Instantiates a new clock with time given in seconds and optionally increment in seconds.

''' </summary>

''' <param name="TIME"></param>

''' <param name="INCREMENT"></param>

Public Sub New(g As cGame, TIME As Integer, Optional INCREMENT As Integer = 0)

Me.game = g : Me.\_time = TIME : Me.\_increment = INCREMENT

\_counting = False

End Sub

''' <summary>

''' Toggles the state of the clock, if the clock is set to counting then it is paused, and vice-versa.

''' </summary>

Public Sub toggle()

If \_counting Then

count\_thread.Abort()

count\_thread = Nothing

Me.\_time += \_increment

Else

count\_thread = New System.Threading.Thread(AddressOf count)

count\_thread.Start()

End If

End Sub

Private Sub count()

Do Until count\_thread.ThreadState = ThreadState.Suspended Or count\_thread.ThreadState = ThreadState.Aborted

Threading.Thread.Sleep(500)

Me.\_time -= 0.5

If Me.\_time < 0 Then

Me.\_counting = False

RaiseEvent time\_up(Me)

End If

Loop

End Sub

End Class

<Serializable()>

Public Class cGame

Private WithEvents \_board As cBoard

Private WithEvents \_whiteClock As cClock

Private WithEvents \_blackClock As cClock

Private \_user1 As cAccount

Private \_user2 As cAccount

Private \_gameOver As Boolean

Public boardList As LinkedList(Of cBoard)

Public WithEvents \_gameGUI As cGameGUI

#Region "Mutators & Accessors"

Public Function get\_gui() As cGameGUI

Return Me.\_gameGUI

End Function

Public Function get\_board() As cBoard

Return Me.\_board

End Function

Public Function get\_user1() As cAccount

Return Me.\_user1

End Function

Public Sub set\_user1(ByRef val As cAccount)

Me.\_user1 = val

End Sub

Public Function get\_user2() As cAccount

Return Me.\_user2

End Function

Public Sub set\_user2(ByRef val As cAccount)

Me.\_user2 = val

End Sub

#End Region

Public Sub New(TARGET As Form, SETTINGS As sSettings)

Me.\_board = New cBoard

Me.boardList = New LinkedList(Of cBoard)

Me.boardList.AddLast(Me.\_board.get\_deep\_clone(Of cBoard)(Me.\_board))

Me.\_gameGUI = New cGameGUI(TARGET, Me, SETTINGS)

End Sub

Public Sub setup\_clock(seconds As Integer, Optional increment As Integer = 0)

Me.\_whiteClock = New cClock(Me, seconds, increment)

Me.\_blackClock = New cClock(Me, seconds, increment)

End Sub

Public Sub undo\_move()

If boardList.Last.Previous IsNot Nothing Then

Me.\_board = Me.\_board.get\_deep\_clone(Of cBoard)(boardList.Last.Previous.Value)

boardList.RemoveLast()

Me.\_gameGUI.getBoardGUI.set\_board(Me.\_board)

Me.\_gameGUI.getMoveList.ClearAndUpdate(Me.\_board)

End If

End Sub

End Class

'SHANNONS NUMBER; 10^(120)

Public Class Minimax

''' <summary>

''' Returns the best move as determined by looking the given number of half-plys ahead from a given position.

''' </summary>

''' <param name="BOARD">Inputs the position to begin evaluation from.</param>

''' <param name="DEPTH">Specifies how many half-plies forward the engine will search to generate the evaluation score. Higher depths will score as position more accurately at the expense of time.</param>

''' <returns></returns>

Public Shared Function calc\_best\_move(DEPTH As Integer, BOARD As cBoard) As sMove

Dim highest As Double = Double.MinValue

Dim lowest As Double = Double.MaxValue

Dim current As Double

Dim bestMove As sMove

For Each move As sMove In BOARD.get\_legal\_moves

Dim b As cBoard = BOARD.get\_deep\_clone(Of cBoard)(BOARD)

b.make\_move(move)

If b.get\_whose\_turn = Alliance.Black Then

current = max(DEPTH - 1, b)

If current >= highest Then

bestMove = move

highest = current

End If

ElseIf b.get\_whose\_turn = Alliance.White Then

current = min(DEPTH - 1, b)

If current <= lowest Then

bestMove = move

lowest = current

End If

End If

Next

Return bestMove

End Function

Private Shared Function min(DEPTH As Integer, BOARD As cBoard) As Double

Dim minimum As Double = Double.MaxValue

If DEPTH = 0 Then

Return New sEval(BOARD).get\_summed\_eval ''TODO: TEST

Else

Dim b As cBoard = BOARD.get\_deep\_clone(Of cBoard)(BOARD)

For Each move As sMove In BOARD.get\_legal\_moves

b.make\_move(move)

Dim val As Double = max(DEPTH - 1, b)

If val >= minimum Then minimum = val

Next

End If

Return minimum

End Function

Private Shared Function max(DEPTH As Integer, BOARD As cBoard) As Double

Dim maximum As Double = Double.MinValue

If DEPTH = 0 Then

Return New sEval(BOARD).get\_summed\_eval

Else

Dim b As cBoard = BOARD.get\_deep\_clone(Of cBoard)(BOARD)

For Each move As sMove In BOARD.get\_legal\_moves

b.make\_move(move)

Dim val As Double = min(DEPTH - 1, b)

If val <= maximum Then maximum = val

Next

End If

Return maximum

End Function

End Class

Public Structure sEval

Private \_board As cBoard

Private \_attackMap As cAttackMap

Private \_wMaterial, \_bMaterial As Double

Private \_wScope, \_bScope As Double

Private \_wKingSafety, \_bKingSafety As Double

Private \_wHangingMaterial, \_bHangingMaterial As Double

Private \_summedEval As Double 'positive values represent a white advantage, negative a black advantage and 0 indicates a draw.

Public Function get\_summed\_eval() As Double

Return Me.\_summedEval

End Function

Public Sub New(BOARD As cBoard)

Me.\_board = BOARD

Me.\_attackMap = New cAttackMap(Me.\_board)

Me.\_wMaterial = eval\_material(Alliance.White) : Me.\_wScope = eval\_scope(Alliance.White) : Me.\_wKingSafety = eval\_king\_safety(Alliance.White)

Me.\_bMaterial = eval\_material(Alliance.Black) : Me.\_bScope = eval\_scope(Alliance.Black) : Me.\_bKingSafety = eval\_king\_safety(Alliance.Black)

Me.\_summedEval = calc\_summed\_eval()

End Sub

Public Function calc\_summed\_eval() As Double

Return (Me.\_wMaterial + Me.\_wScope + Me.\_wKingSafety + Me.\_bHangingMaterial) - (Me.\_bMaterial + Me.\_bScope + Me.\_bKingSafety + Me.\_wHangingMaterial)

End Function

#Region "Evaluation Functions"

Private Function eval\_material(ALLIANCE As Alliance) As Double

Dim val As Integer

For Each piece As cPiece In Me.\_board.get\_pieces(ALLIANCE)

If Not (TypeOf piece Is cKing) Then val += piece.get\_value

Next

Return val

End Function

Private Function eval\_king\_safety(ALLIANCE As Alliance) As Double

Dim moves As sMove() = Me.\_board.get\_pseudo\_legal(ALLIANCE)

Dim opponent As Alliance : If ALLIANCE = chess.Alliance.White Then opponent = chess.Alliance.Black Else opponent = chess.Alliance.White

Dim kingLoc As Byte = Me.\_board.find\_king(opponent).get\_coordinate

Dim attacks As Byte = 0

If moves IsNot Nothing Then

For Each move As sMove In moves

If move.dest = kingLoc Then attacks += CByte(1)

Next

End If

Return 0 ''REMOVE THIS AFTER TESTING''

Return CDbl(0.05 \* attacks)

End Function

Private Function eval\_scope(ALLIANCE As Alliance) As Double

Dim moves As sMove() = Me.\_board.get\_pseudo\_legal(ALLIANCE)

Return CDbl(moves.Length \* 0.05)

End Function

#End Region

Public Overrides Function toString() As String

Dim msg As String = Nothing

Select Case Me.\_summedEval

Case Is = 0

msg = "Draw."

Case Is > 0

msg = "White is winning."

Case Is < 0

msg = "Black is winning."

End Select

Return String.Concat("(W): {", "M: ", Me.\_wMaterial, ", S: ", Me.\_wScope, ", SAFE: ", Me.\_wKingSafety, ", HANGING: ", Me.\_wHangingMaterial, "}", vbCrLf, "(B): {M: ", Me.\_bMaterial, ", S: ", Me.\_bScope, ", SAFE: ", Me.\_bKingSafety, "HANGING: ", Me.\_bHangingMaterial, "}", vbCrLf, "Summed: ", Me.\_summedEval, vbCrLf, msg)

End Function

End Structure

<Serializable()>

Public Structure sMove

Public ogCoord As Byte

Public dest As Byte

Public Sub New(OG\_COORD As Byte, DEST As Byte)

Me.ogCoord = OG\_COORD

Me.dest = DEST

End Sub

#Region "Operator Overloads"

Public Shared Operator =(v1 As sMove, v2 As sMove) As Boolean

If v1.dest = v2.dest And v1.ogCoord = v2.ogCoord Then Return True Else Return False

End Operator

Public Shared Operator <>(v1 As sMove, v2 As sMove) As Boolean

If v1.dest = v2.dest And v1.ogCoord = v2.ogCoord Then Return False Else Return True

End Operator

#End Region

Public Overloads Function ToString(BOARD As cBoard) As String

'THIS FUNCTION WILL ONLY WORK IF CALLED BEFORE THE MOVE IS MADE ON THE BOARD, AKA THE PARAMETER IS THE BOARD BEFORE THE MOVE

Dim movingPiece As cPiece = BOARD.get\_tiles(Me.ogCoord).get\_piece

Dim isCapture As Boolean = BOARD.get\_tile(Me.dest).is\_occupied

Dim isAmbiguous As Boolean = Me.is\_ambigious(BOARD, movingPiece)

Dim castlingTiles As Byte() = {58, 62, 2, 6}

'a move is deemed to be ambiguous if another piece of the same alliance and piece type can also move the the destination square.

If isAmbiguous Then

Dim cap As String

If isCapture Then cap = "x" Else cap = Nothing

ToString = calc\_char(movingPiece, isCapture) & BOARD.get\_tile(movingPiece.get\_coordinate).get\_file & cap & BOARD.get\_tile(Me.dest).get\_file & BOARD.get\_tile(Me.dest).get\_rank

Else

If movingPiece.get\_title = Chessman.Pawn AndAlso BOARD.get\_tile(Me.dest).get\_coordinate = BOARD.get\_enpassent Then 'en passent capture

ToString = LCase(BOARD.get\_tile(Me.ogCoord).get\_file & "x" & BOARD.get\_tile(Me.dest).get\_file & BOARD.get\_tile(Me.dest).get\_rank)

ElseIf movingPiece.get\_title = Chessman.Pawn AndAlso ((BOARD.get\_tile(Me.dest).get\_coordinate \ 8 + 1 = 1) AndAlso movingPiece.get\_alliance = Alliance.White) Then 'black promotion

ToString = LCase(BOARD.get\_tile(Me.dest).get\_file & BOARD.get\_tile(Me.dest).get\_rank) & "=Q"

ElseIf movingPiece.get\_title = Chessman.Pawn AndAlso (BOARD.get\_tile(Me.dest).get\_coordinate \ 8 + 1 = 8) AndAlso movingPiece.get\_alliance = Alliance.Black Then 'white promotion

ToString = LCase(BOARD.get\_tile(Me.dest).get\_file & BOARD.get\_tile(Me.dest).get\_rank) & "=Q"

ElseIf movingPiece.get\_title = Chessman.King AndAlso Not CType(movingPiece, cKing).get\_moved AndAlso castlingTiles.Contains(Me.dest) Then 'castling

If Me.dest = 2 Or Me.dest = 58 Then ToString = "O-O-O" Else ToString = "O-O"

Else 'normal move66

Dim cap As String

If isCapture Then cap = CChar("x") Else cap = ""

ToString = calc\_char(movingPiece, isCapture) & cap & BOARD.get\_tile(Me.dest).get\_file & BOARD.get\_tile(Me.dest).get\_rank

End If

End If

Dim t As cBoard = BOARD.get\_deep\_clone(Of cBoard)(BOARD)

t.make\_move(Me)

Dim s As GameState = t.calculate\_board\_state()

If s = GameState.WhiteMated Or s = GameState.BlackMated Then Return ToString & "#" 'if there was a mate then update and return the string

If t.is\_in\_check(Alliance.White) Or t.is\_in\_check(Alliance.Black) Then Return ToString & "+" 'if the move resulted in check, then update and return the string.

Return ToString

End Function

Private Function calc\_char(PIECE As cPiece, IS\_CAPTURE As Boolean) As String

Select Case PIECE.get\_title

Case Chessman.Pawn

If IS\_CAPTURE Then Return LCase(Chr(64 + (PIECE.get\_coordinate Mod 8 + 1)))

Case Chessman.Knight

Return CChar("N")

Case Chessman.Bishop

Return CChar("B")

Case Chessman.Rook

Return CChar("R")

Case Chessman.Queen

Return CChar("Q")

Case Chessman.King

Return CChar("K")

End Select

Return ""

End Function

Private Function is\_ambigious(BOARD As cBoard, MOVING\_PIECE As cPiece) As Boolean

Dim alliance As Alliance = MOVING\_PIECE.get\_alliance

For Each move As sMove In BOARD.get\_legal\_moves

If move <> Me Then

If BOARD.get\_tile(move.ogCoord).get\_piece.get\_alliance = alliance Then

If move.dest = Me.dest AndAlso BOARD.get\_tile(move.ogCoord).get\_piece.get\_title = BOARD.get\_tile(Me.ogCoord).get\_piece.get\_title Then Return True

End If

End If

Next

Return False

End Function

#Region "Moves"

Public Shared Sub regular(MOVE As sMove, BOARD As cBoard)

Dim movingPiece As cPiece = BOARD.get\_tile(MOVE.ogCoord).get\_piece 'error @ 1. e4, d5 2. exd5, c6 3. dxc6, Bd7 4. cxb7, Be6 5. bxa8=Q, Qd7 6. Qaf3

Dim targetTile As cTile = BOARD.get\_tile(MOVE.dest)

movingPiece.set\_coordinate(targetTile.get\_coordinate) 'update the pieces coordinate

targetTile.set\_piece(movingPiece) 'update the tiles piece reference

BOARD.get\_tiles(MOVE.ogCoord).set\_piece(Nothing) 'update the og tiles piece reference

End Sub

Public Shared Sub enpassent(MOVE As sMove, BOARD As cBoard)

Dim movingPiece As cPiece = BOARD.get\_tile(MOVE.ogCoord).get\_piece

Dim targetTile As cTile = BOARD.get\_tile(MOVE.dest)

movingPiece.set\_coordinate(MOVE.dest)

targetTile.set\_piece(movingPiece)

If BOARD.get\_whose\_turn = Alliance.White Then

BOARD.get\_tile(CByte(MOVE.dest + 8)).set\_piece(Nothing)

Else

BOARD.get\_tile(CByte(MOVE.dest - 8)).set\_piece(Nothing)

End If

BOARD.get\_tile(MOVE.ogCoord).set\_piece(Nothing)

End Sub

Public Shared Sub castle(MOVE As sMove, BOARD As cBoard)

sMove.regular(MOVE, BOARD)

Dim rookMove As sMove

Select Case MOVE.dest

Case 58

rookMove = New sMove(56, 59)

Case 62

rookMove = New sMove(63, 61)

Case 2

rookMove = New sMove(0, 3)

Case 6

rookMove = New sMove(7, 5)

End Select

sMove.regular(rookMove, BOARD)

End Sub

Public Shared Sub promotion(MOVE As sMove, BOARD As cBoard)

Dim movingPiece As cPiece = BOARD.get\_tile(MOVE.ogCoord).get\_piece

Dim targetTile As cTile = BOARD.get\_tile(MOVE.dest)

targetTile.set\_piece(New cQueen(movingPiece.get\_alliance, MOVE.dest)) 'puts the new queen on the board

BOARD.get\_tile(MOVE.ogCoord).set\_piece(Nothing) 'removes the promoting piece from the board

BOARD.update\_pieces(Alliance.White) 'updates the piece list of the board for both players.

BOARD.update\_pieces(Alliance.Black)

End Sub

#End Region

End Structure

<Serializable()>

Public Class cBoardGUI

Inherits Panel

Private WithEvents \_game As cGame

Private \_board As cBoard

Private \_gameGUI As cGameGUI

Private \_tiles(63) As PictureBox

Private \_lightSquare As Color

Private \_darkSquare As Color

Private \_whiteBottom As Boolean

'ui mechanics

Private \_selectedPiece As cPiece

Private \_lastMove As sMove

Private \_ogCol As Color

Private \_destCol As Color

Private \_showLegal As Boolean

'drag drop graphics

Private \_ogTile As PictureBox

Private \_heldPiece As cPieceGraphics

'mouse enter graphics

Private \_tileBorder As cTileBorder

Private \_showingGhostBoard As Boolean

Public Function calc\_size() As Integer

Return CInt(Me.Size.Height / 8)

End Function

#Region "Mutators & Accessors"

Public Function get\_light\_colour() As Color

Return Me.\_lightSquare

End Function

Public Sub set\_light\_colour(VALUE As Color)

Me.\_lightSquare = VALUE

End Sub

Public Function get\_dark\_colour() As Color

Return Me.\_darkSquare

End Function

Public Sub set\_dark\_colour(VALUE As Color)

Me.\_darkSquare = VALUE

End Sub

Public Sub set\_show\_legal(VALUE As Boolean)

Me.\_showLegal = VALUE

End Sub

Public Sub set\_board(VALUE As cBoard)

Me.\_board = VALUE

Me.update\_graphics()

End Sub

Public Function get\_board() As cBoard

Return Me.\_board

End Function

#End Region

Public Sub New(GAME As cGame, BOARD As cBoard, BOARD\_DIMENSION As Integer, SETTINGS As sSettings, GAME\_GUI As cGameGUI)

Me.\_lightSquare = SETTINGS.lightSquare

Me.\_darkSquare = SETTINGS.darkSquare

Me.\_showLegal = SETTINGS.showLegalMoves

Me.\_ogCol = Color.Gold

Me.\_destCol = Color.Yellow

Me.\_tileBorder = New cTileBorder(CInt(BOARD\_DIMENSION / 8), Color.White)

Me.Controls.Add(Me.\_tileBorder)

Me.\_tileBorder.BringToFront()

Me.\_tileBorder.Hide()

Me.\_gameGUI = GAME\_GUI

Me.\_whiteBottom = True

'instantiates the held piece object, for drag dropping.

Me.\_heldPiece = New cPieceGraphics(CInt(BOARD\_DIMENSION / 8))

Me.Controls.Add(\_heldPiece)

With (Me.\_heldPiece)

.SizeMode = PictureBoxSizeMode.CenterImage

.Hide()

End With

Me.\_board = BOARD

Me.\_game = GAME

Me.Size = New Size(BOARD\_DIMENSION, BOARD\_DIMENSION)

Me.init\_tiles(CInt(BOARD\_DIMENSION / 8))

Me.add\_handlers()

Me.update\_graphics()

End Sub

Public Sub resize\_all(NEW\_TILE\_SIZE As Integer)

Me.Size = New Size(NEW\_TILE\_SIZE \* 8, NEW\_TILE\_SIZE \* 8)

Dim col As Color = Me.\_lightSquare

Dim x As Integer = 0

Dim y As Integer = 0

Me.\_tileBorder.Hide()

For i = 0 To 63

\_tiles(i) = New PictureBox

With \_tiles(i)

.Size = New Size(NEW\_TILE\_SIZE, NEW\_TILE\_SIZE)

.Location = New Point(x, y)

x += NEW\_TILE\_SIZE

If x = 8 \* NEW\_TILE\_SIZE Then

y += NEW\_TILE\_SIZE

x = 0

End If

End With

Next

Me.\_heldPiece.Dispose()

Me.\_heldPiece = New cPieceGraphics(CInt(NEW\_TILE\_SIZE))

End Sub

Private Sub init\_tiles(TILE\_SIZE As Integer)

Dim col As Color = Me.\_lightSquare

Dim x As Integer = 0

Dim y As Integer = 0

For i = 0 To 63

\_tiles(i) = New PictureBox

With \_tiles(i)

Me.Controls.Add(\_tiles(i))

.SizeMode = PictureBoxSizeMode.StretchImage

.BackgroundImageLayout = ImageLayout.Center

.BackColor = col

.Size = New Size(TILE\_SIZE, TILE\_SIZE)

.Location = New Point(x, y)

.Cursor = Cursors.Hand

x += TILE\_SIZE

If x = 8 \* TILE\_SIZE Then

y += TILE\_SIZE

x = 0

If col = Me.\_lightSquare Then col = Me.\_darkSquare Else col = Me.\_lightSquare

End If

If col = Me.\_lightSquare Then col = Me.\_darkSquare Else col = Me.\_lightSquare

End With

Next

End Sub

Private Function calc\_image(ALLIANCE As Alliance, TITLE As Chessman) As Bitmap

Select Case ALLIANCE

Case Alliance.White

Select Case TITLE

Case Chessman.Pawn

Return My.Resources.awP

Case Chessman.Knight

Return My.Resources.awN

Case Chessman.Bishop

Return My.Resources.awB

Case Chessman.Rook

Return My.Resources.awR

Case Chessman.Queen

Return My.Resources.awQ

Case Chessman.King

Return My.Resources.awK

End Select

Case Alliance.Black

Select Case TITLE

Case Chessman.Pawn

Return My.Resources.abP

Case Chessman.Knight

Return My.Resources.abN

Case Chessman.Bishop

Return My.Resources.abB

Case Chessman.Rook

Return My.Resources.abR

Case Chessman.Queen

Return My.Resources.abQ

Case Chessman.King

Return My.Resources.abK

End Select

End Select

Return Nothing

End Function

Public Sub update\_graphics()

If Me.\_board IsNot Nothing AndAlso Me.\_board.get\_move\_list.Last IsNot Nothing Then Me.\_lastMove = Me.\_board.get\_move\_list.Last.Value

'update the back colour and resets the background image

For i = 0 To 63

If \_selectedPiece IsNot Nothing AndAlso \_selectedPiece.get\_coordinate = i Then

Me.\_tiles(i).BackColor = Color.Orange

Else

If Me.\_board.get\_tiles(i).get\_is\_lightsquare Then

Me.\_tiles(i).BackColor = \_lightSquare

Else

Me.\_tiles(i).BackColor = \_darkSquare

End If

End If

Me.\_tiles(i).BackgroundImage = Nothing

Next

'update the image on each tile

For i = 0 To 63

If Me.\_board.get\_tiles(i).is\_occupied Then

Dim x As New Bitmap(calc\_image(Me.\_board.get\_tiles(i).get\_piece.get\_alliance, Me.\_board.get\_tiles(i).get\_piece.get\_title), New Size(Me.\_tiles(0).Height - 5, Me.\_tiles(0).Height - 5))

Me.\_tiles(i).Image = x

Else

Me.\_tiles(i).Image = Nothing

End If

Next

'update the possible moves

If Me.\_selectedPiece IsNot Nothing AndAlso Me.\_selectedPiece.calc\_pseudo(Me.\_board) IsNot Nothing Then

For Each move As sMove In \_selectedPiece.calc\_pseudo(Me.\_board)

If \_board.is\_legal\_move(move) Then

If Me.\_showLegal Then

If \_board.get\_tiles(move.dest).is\_occupied Then

Me.\_tiles(move.dest).BackgroundImageLayout = ImageLayout.Stretch

Me.\_tiles(move.dest).BackgroundImage = My.Resources.attackedOccupied

Else

Me.\_tiles(move.dest).BackgroundImageLayout = ImageLayout.Center

Me.\_tiles(move.dest).BackgroundImage = New Bitmap(My.Resources.attacked, New Size(CInt(Me.\_tiles(move.dest).Size.Height / 3), CInt(Me.\_tiles(move.dest).Size.Width / 3)))

End If

Else

Me.\_tiles(move.dest).BackgroundImage = Nothing

End If

End If

Next

End If

'update check indicators

If Me.\_board.is\_in\_check(Alliance.White) Then

Me.\_tiles(Me.\_board.find\_king(Alliance.White).get\_coordinate).BackColor = Color.IndianRed

End If

If Me.\_board.is\_in\_check(Alliance.Black) Then

Me.\_tiles(Me.\_board.find\_king(Alliance.Black).get\_coordinate).BackColor = Color.IndianRed

End If

'update move indicators

If Me.\_lastMove <> Nothing Then

Me.\_tiles(\_lastMove.ogCoord).BackColor = Me.\_ogCol

Me.\_tiles(\_lastMove.dest).BackColor = Me.\_destCol

End If

With Me.\_board

If .get\_state <> GameState.ongoing Then

If .get\_state = GameState.WhiteMated Then

My.Computer.Audio.Play(System.AppDomain.CurrentDomain.BaseDirectory & "\checkmate.wav")

MsgBox("Checkmate, black wins.")

ElseIf .get\_state = GameState.BlackMated Then

My.Computer.Audio.Play(System.AppDomain.CurrentDomain.BaseDirectory & "\checkmate.wav")

MsgBox("Checkmate, white wins.")

ElseIf .get\_state = GameState.Draw Or .get\_state = GameState.Stalemate Then

MsgBox("Draw.")

End If

End If

End With

End Sub

Private Sub add\_handlers()

For i = 0 To 63

AddHandler Me.\_tiles(i).MouseDown, AddressOf e\_mousedown

AddHandler Me.\_tiles(i).MouseMove, AddressOf e\_mousemove

AddHandler Me.\_tiles(i).MouseUp, AddressOf e\_mouseup

Next

End Sub

Private Function find\_tile\_index(sender As Object) As Integer

For i = 0 To 63

If CType(sender, PictureBox) Is Me.\_tiles(i) Then Return i

Next

End Function

#Region "Handlers"

Private Sub e\_mousedown(SENDER As Object, E As MouseEventArgs)

If E.Button = MouseButtons.Left AndAlso Not (Me.\_board.is\_game\_over) Then

Me.Cursor = Cursors.SizeAll

Me.\_ogTile = CType(SENDER, PictureBox)

Dim clickedTile As cTile = Me.\_board.get\_tiles(Me.find\_tile\_index(SENDER))

Dim clickedTilePB As PictureBox = CType(SENDER, PictureBox)

If clickedTile.is\_occupied AndAlso clickedTile.get\_piece.get\_alliance = Me.\_board.get\_whose\_turn Then

If Me.\_selectedPiece Is Nothing AndAlso clickedTile.is\_occupied AndAlso clickedTile.get\_piece.get\_alliance = Me.\_board.get\_whose\_turn Or Me.\_ogTile.Bounds.Contains(PointToClient(MousePosition)) Then

'if a piece hasn't already been selected and is a valid piece to move then select the clicked piece.

Me.\_selectedPiece = clickedTile.get\_piece

Me.update\_graphics()

With (Me.\_heldPiece)

.Show()

.UpdateImage(calc\_image(Me.\_selectedPiece.get\_alliance, Me.\_selectedPiece.get\_title))

.Location = clickedTilePB.Location

.BringToFront()

End With

Me.\_tileBorder.Show()

clickedTilePB.Image = Nothing

End If

ElseIf \_selectedPiece IsNot Nothing Then

Dim destTile As PictureBox

For i = 0 To UBound(Me.\_tiles)

If Me.\_tiles(i).Bounds.Contains(PointToClient(MousePosition)) Then destTile = Me.\_tiles(i)

Next 'finds the destination button

If destTile Is Nothing Then Throw New Exception

Dim m As New sMove(Me.\_selectedPiece.get\_coordinate, Me.\_board.get\_tiles(Me.find\_tile\_index(destTile)).get\_coordinate)

If Me.\_board.is\_legal\_move(m) Then

Me.move\_made(m)

Else

'if the move is not valid

Me.\_selectedPiece = Nothing

Me.\_ogTile = Nothing

Me.update\_graphics()

End If

Else

'if the move is not valid

Me.\_selectedPiece = Nothing

Me.\_ogTile = Nothing

Me.update\_graphics()

End If

ElseIf E.Button = MouseButtons.Right Then

If CType(SENDER, PictureBox).BackColor = Color.Green Or CType(SENDER, PictureBox).BackColor = Color.Red Or CType(SENDER, PictureBox).BackColor = Color.Blue Then

If Me.\_lastMove <> Nothing Then

If Me.\_lastMove.ogCoord = find\_tile\_index(SENDER) Then

CType(SENDER, PictureBox).BackColor = Me.\_ogCol

ElseIf Me.\_lastMove.dest = find\_tile\_index(SENDER) Then

CType(SENDER, PictureBox).BackColor = Me.\_destCol

Else

If Me.\_board.get\_tiles(Me.find\_tile\_index(SENDER)).get\_is\_lightsquare Then CType(SENDER, PictureBox).BackColor = Me.\_lightSquare Else CType(SENDER, PictureBox).BackColor = Me.\_darkSquare

End If

Else

If Me.\_board.get\_tiles(Me.find\_tile\_index(SENDER)).get\_is\_lightsquare Then CType(SENDER, PictureBox).BackColor = Me.\_lightSquare Else CType(SENDER, PictureBox).BackColor = Me.\_darkSquare

End If

Else

If My.Computer.Keyboard.AltKeyDown Then

CType(SENDER, PictureBox).BackColor = Color.Green

ElseIf My.Computer.Keyboard.ShiftKeyDown Then

CType(SENDER, PictureBox).BackColor = Color.Blue

Else

CType(SENDER, PictureBox).BackColor = Color.Red

End If

End If

End If

End Sub

Private Sub e\_mousemove(SENDER As Object, E As MouseEventArgs)

If Me.\_selectedPiece IsNot Nothing Then

Dim mp As Point = New Point(CInt(PointToClient(MousePosition).X - Me.\_heldPiece.Width / 2), CInt(PointToClient(MousePosition).Y - Me.\_heldPiece.Height / 2))

If E.Button = MouseButtons.Left Then

Me.\_heldPiece.Location = mp

Dim t As PictureBox

For Each temp As PictureBox In Me.\_tiles

If temp.Bounds.Contains(PointToClient(MousePosition)) Then

t = temp

Exit For

End If

Next

If t IsNot Nothing Then Me.\_tileBorder.Location = t.Location

End If

End If

End Sub

Private Sub e\_mouseup(SENDER As Object, E As MouseEventArgs)

If E.Button = MouseButtons.Left Then

Me.\_tileBorder.Hide()

Me.Refresh()

Me.\_heldPiece.Hide()

If \_selectedPiece IsNot Nothing Then

If Me.\_ogTile.Bounds.Contains(PointToClient(MousePosition)) Then

Me.\_ogTile.Image = New Bitmap(Me.calc\_image(Me.\_selectedPiece.get\_alliance, Me.\_selectedPiece.get\_title), New Size(70, 70))

Else

Dim destTile As PictureBox

For i = 0 To UBound(Me.\_tiles)

If Me.\_tiles(i).Bounds.Contains(PointToClient(MousePosition)) Then destTile = Me.\_tiles(i)

Next 'finds the destination button

Dim m As New sMove(Me.\_selectedPiece.get\_coordinate, Me.\_board.get\_tiles(Me.find\_tile\_index(destTile)).get\_coordinate)

If Me.\_board.is\_legal\_move(m) Then

Me.move\_made(m)

Else

'if the move is not valid

Me.\_selectedPiece = Nothing

Me.\_ogTile = Nothing

Me.update\_graphics()

End If

End If

End If

End If

End Sub

Private Sub move\_made(m As sMove)

Dim wasCapture As Boolean = Me.\_board.get\_tiles(m.dest).is\_occupied

Dim whosTurn As Alliance = \_board.get\_whose\_turn

Me.\_board.get\_move\_list\_string.AddLast(m.ToString(Me.\_board))

Me.\_gameGUI.getMoveList.AddMove(m.ToString(Me.\_board), \_board.get\_whose\_turn)

\_board.make\_move(m)

Me.\_game.boardList.AddLast(Me.\_board.get\_deep\_clone(Of cBoard)(Me.\_board)) 'update the board list

Me.\_gameGUI.\_moveListGUI.UpdateOpeningName()

Me.\_ogTile = Nothing

Me.\_selectedPiece = Nothing

Me.\_lastMove = m 'updates the last move made in the game.

Me.update\_graphics()

If whosTurn <> \_board.get\_whose\_turn Then 'this verifies that the move was actually played and legal

Dim ext As String = Nothing

If wasCapture Then ext = "\capture.wav" Else ext = "\pieceMoved.wav"

My.Computer.Audio.Play(System.AppDomain.CurrentDomain.BaseDirectory & ext)

End If

End Sub

#End Region

Public Sub flip\_board()

Me.\_gameGUI.getBoardWrapper.FlipRankMarkers()

Dim xGap As Integer

xGap = (Me.\_tiles(0).Location.X + Me.\_tiles(0).Width) - Me.\_tiles(1).Location.X

Dim yGap As Integer

yGap = (Me.\_tiles(0).Location.Y + Me.\_tiles(0).Height) - Me.\_tiles(8).Location.Y

Dim x, y As Integer

Dim [start], [end], [step] As Integer

If Me.\_whiteBottom Then

start = 63 : [end] = 0 : [step] = -1

Else

start = 0 : [end] = 63 : [step] = +1

End If

For i = [start] To [end] Step [step]

With \_tiles(i)

.Location = New Point(x, y)

x += Me.\_tiles(0).Width

If x = 8 \* Me.\_tiles(0).Width Then

y += Me.\_tiles(0).Width

x = 0

End If

End With

Next

Me.update\_graphics()

Me.Update()

Me.\_whiteBottom = Not Me.\_whiteBottom

Debug.WriteLine(CStr("Board flipped, whiteBottom: " & Me.\_whiteBottom.ToString))

End Sub

''' <summary>

''' Will temporarily display the referenced board on the GUI until any event causes the current position to be loaded again (i.e. a move being played,

''' mouse down event etc).

''' </summary>

''' <param name="BOARD">Reference to the board that you would like to display.</param>

''' <remarks></remarks>

Private Sub update\_ghost\_graphics(BOARD As cBoard)

Me.\_lastMove = BOARD.get\_move\_list.Last.Value

'update the back colour and resets the background image

For i = 0 To 63

If \_selectedPiece IsNot Nothing AndAlso \_selectedPiece.get\_coordinate = i Then

Me.\_tiles(i).BackColor = Color.Orange

Else

If BOARD.get\_tiles(i).get\_is\_lightsquare Then

Me.\_tiles(i).BackColor = \_lightSquare

Else

Me.\_tiles(i).BackColor = \_darkSquare

End If

End If

Me.\_tiles(i).BackgroundImage = Nothing

Next

'update the image on each tile

For i = 0 To 63

If BOARD.get\_tiles(i).is\_occupied Then

Dim x As New Bitmap(calc\_image(BOARD.get\_tiles(i).get\_piece.get\_alliance, BOARD.get\_tiles(i).get\_piece.get\_title), New Size(Me.\_tiles(0).Height - 5, Me.\_tiles(0).Height - 5))

Me.\_tiles(i).Image = x

Else

Me.\_tiles(i).Image = Nothing

End If

Next

'update the possible moves

If Me.\_selectedPiece IsNot Nothing AndAlso Me.\_selectedPiece.calc\_pseudo(BOARD) IsNot Nothing Then

For Each move As sMove In \_selectedPiece.calc\_pseudo(BOARD)

If BOARD.is\_legal\_move(move) Then

If Me.\_showLegal Then

If BOARD.get\_tiles(move.dest).is\_occupied Then

Me.\_tiles(move.dest).BackgroundImageLayout = ImageLayout.Stretch

Me.\_tiles(move.dest).BackgroundImage = My.Resources.attackedOccupied

Else

Me.\_tiles(move.dest).BackgroundImageLayout = ImageLayout.Center

Me.\_tiles(move.dest).BackgroundImage = New Bitmap(My.Resources.attacked, New Size(CInt(Me.\_tiles(move.dest).Size.Height / 3), CInt(Me.\_tiles(move.dest).Size.Width / 3)))

End If

Else

Me.\_tiles(move.dest).BackgroundImage = Nothing

End If

End If

Next

End If

'update check indicators

If BOARD.is\_in\_check(Alliance.White) Then

Me.\_tiles(BOARD.find\_king(Alliance.White).get\_coordinate).BackColor = Color.IndianRed

End If

If BOARD.is\_in\_check(Alliance.Black) Then

Me.\_tiles(BOARD.find\_king(Alliance.Black).get\_coordinate).BackColor = Color.IndianRed

End If

'update move indicators

If Me.\_lastMove <> Nothing Then

Me.\_tiles(\_lastMove.ogCoord).BackColor = Me.\_ogCol

Me.\_tiles(\_lastMove.dest).BackColor = Me.\_destCol

End If

Me.\_showingGhostBoard = True

End Sub

Public Sub show\_ghost\_position(moveIndex As Integer)

Dim toShow As cBoard = Me.\_game.boardList(moveIndex)

If toShow Is Me.\_game.boardList.Last.Value Then

update\_graphics()

Else

update\_ghost\_graphics(toShow)

End If

End Sub

End Class

<Serializable()>

Public Class cBoardWrapper

Private \_target As cGameGUI

Private \_rankMarkers() As Label

Private \_fileMarkers() As Label

Private \_whiteBottom As Boolean

Public Function getRankMarkers() As Label()

Return Me.\_rankMarkers

End Function

Public Function getFileMarkers() As Label()

Return Me.\_fileMarkers

End Function

Public Sub New(ByRef TARGET As cGameGUI)

Me.\_target = TARGET

Me.AddControls()

Me.\_whiteBottom = True

End Sub

Private Sub set\_colour(c As Color)

For Each rank As Label In Me.\_rankMarkers

rank.ForeColor = c

Next

For Each file As Label In Me.\_fileMarkers

file.ForeColor = c

Next

End Sub

Private Sub AddControls()

For i = 0 To 7

ReDim \_rankMarkers(i)

ReDim \_fileMarkers(i)

Next

For i = 0 To 7

\_rankMarkers(i) = New Label

\_fileMarkers(i) = New Label

Me.\_target.Controls.Add(\_rankMarkers(i))

Me.\_target.Controls.Add(\_fileMarkers(i))

\_rankMarkers(i).SendToBack()

\_fileMarkers(i).SendToBack()

With \_rankMarkers(i)

.Font = New Font("Microsoft Sans Serif", 10)

.TextAlign = ContentAlignment.MiddleLeft

.ForeColor = Color.White

.AutoSize = False

.Size = New Size(20, Me.\_target.getBoardGUI.calc\_size)

.Location = New Point(Me.\_target.getBoardGUI.Location.X - 25, Me.\_target.getBoardGUI.Location.Y + (Me.\_target.getBoardGUI.calc\_size \* i))

.Text = CStr(8 - i)

End With

With \_fileMarkers(i)

.Font = New Font("Microsoft Sans Serif", 10)

.TextAlign = ContentAlignment.MiddleCenter

.ForeColor = Color.White

.AutoSize = False

.Size = New Size(Me.\_target.getBoardGUI.calc\_size, 25)

.Location = New Point(Me.\_target.getBoardGUI.Location.X + (Me.\_target.getBoardGUI.calc\_size \* i), Me.\_target.getBoardGUI.Location.Y + (Me.\_target.getBoardGUI.calc\_size \* 8))

.Text = Chr(97 + i)

End With

Next

End Sub

Public Sub FlipRankMarkers()

Me.\_whiteBottom = Not (Me.\_whiteBottom)

Dim locations(7) As Point

For i = 0 To 7

locations(i) = Me.\_rankMarkers(i).Location

Next

For i = 7 To 0 Step -1

Me.\_rankMarkers(Math.Abs(i - 7)).Location = locations(i)

Next

End Sub

Public Sub Resize()

For i = 0 To 7

With \_rankMarkers(i)

.Size = New Size(25, Me.\_target.getBoardGUI.calc\_size)

.Location = New Point(Me.\_target.getBoardGUI.Location.X - 25, Me.\_target.getBoardGUI.Location.Y + (Me.\_target.getBoardGUI.calc\_size \* i))

End With

With \_fileMarkers(i)

.Size = New Size(Me.\_target.getBoardGUI.calc\_size, 25)

.Location = New Point(Me.\_target.getBoardGUI.Location.X + (Me.\_target.getBoardGUI.calc\_size \* i), Me.\_target.getBoardGUI.Location.Y + (Me.\_target.getBoardGUI.calc\_size \* 8))

End With

Next

End Sub

Public Sub Destroy()

For Each lbl As Label In Me.\_rankMarkers

Me.\_target.Controls.Remove(lbl)

lbl.Dispose()

Next

For Each lbl As Label In Me.\_fileMarkers

Me.\_target.Controls.Remove(lbl)

lbl.Dispose()

Next

End Sub

End Class

<Serializable()>

Public Class cClockGUI

Inherits Panel

Private \_clock As cClock

Private \_label As Label

Public Sub New(c As cClock)

Me.\_clock = c

Me.init\_componenets()

End Sub

Private Sub init\_componenets()

Me.\_label = New Label : Me.Controls.Add(Me.\_label)

Me.\_label.Font = New Font("Microsoft Sans Serif", FontStyle.Bold)

Me.\_label.AutoSize = True

End Sub

Public Sub update\_graphics()

If Me.\_clock.getCounting Then Me.BackColor = Color.Red Else Me.BackColor = Color.Green

Me.\_label.Text = Me.get\_time\_string

End Sub

Private Function get\_time\_string() As String

Dim minutes As Integer : Dim seconds As Integer

minutes = CInt(CLng(Me.\_clock.getTime) \ 60)

seconds = CInt(Int(Me.\_clock.getTime Mod 60))

Dim mString As String : Dim sString As String

If minutes < 10 Then

mString = "0" & minutes.ToString

ElseIf minutes = 0 Then

mString = "00"

Else

mString = minutes.ToString

End If

If seconds < 10 Then

sString = "0" & seconds.ToString

ElseIf seconds = 0 Then

sString = "00"

Else

sString = seconds.ToString

End If

Return mString & ":" & sString

End Function

End Class

'designed to be able to show moves from a standard chess game of two players. Also supports variants such as crazyhouse etc.

'also includes events that allow previous positions from a game to be mirrored on the board gui through aggregation.

Public Class cCustomMoveList

Inherits Panel

Private \_gameGUI As cGameGUI

Private \_openingText As Label

'headers

Private \_h1 As Label

Private \_h2 As Label

Private \_h3 As Label

Private \_bgCol As Color 'bg

Private \_primCol As Color 'headers, opening text, count

Private \_secCol As Color 'entries

Private \_numberLabels As Label()

Private \_noOfRows As Integer

Private \_entryPanel As Panel 'used for the scrolling function

Private \_entries As Label()

Private \_entryCount As Integer

Private \_maxHeight As Integer

'scroll stuff

Private WithEvents \_scrollBar As Panel

Private \_maxNoOfRows As Byte

Private \_hoverCol As Color = Color.FromArgb(80, 80, 80)

Private \_selectedLbl As Label

Private \_selectedCol As Color = Color.FromArgb(70, 70, 70)

#Region "Accessors and Mutators"

Public Function getEntryCount() As Integer

Return Me.\_entryCount

End Function

Public Sub setBackgroundColour(BG As Color)

Me.\_bgCol = BG

End Sub

Public Sub setPrimaryColour(PRIM As Color)

Me.\_primCol = PRIM

End Sub

Public Sub setSecondaryColour(SEC As Color)

Me.\_secCol = SEC

End Sub

Public Function getBackgroundColour() As Color

Return Me.\_bgCol

End Function

Public Function getPrimaryColour() As Color

Return Me.\_primCol

End Function

Public Function getSecondaryColour() As Color

Return Me.\_secCol

End Function

#End Region

Public Sub New(GAME\_GUI As cGameGUI, BG\_COL As Color, PRIM\_COL As Color, SEC\_COL As Color)

Me.\_gameGUI = GAME\_GUI

Me.\_entryCount = 0 : Me.\_noOfRows = 0

Me.Size = New Size(200, 40)

Me.\_bgCol = BG\_COL : Me.\_primCol = PRIM\_COL : Me.\_secCol = SEC\_COL

Me.InitHeaders()

Me.InitOpeningText()

Me.InitEntryPanel()

Me.InitScroll()

Me.\_maxHeight = CInt(GAME\_GUI.getBoardGUI.Height / 2)

Me.\_maxNoOfRows = CByte((Me.\_maxHeight) \ 20)

End Sub

#Region "Init methods"

Private Sub InitHeaders()

Dim lSet = Sub(ByRef lbl As Label, width As Integer, TEXT As String)

lbl = New Label : Me.Controls.Add(lbl)

lbl.AutoSize = False

lbl.Size = New Size(width, 20)

lbl.BackColor = Me.\_primCol

lbl.Font = New Font("Microsoft Sans Serif", 8)

lbl.ForeColor = Me.\_primCol

lbl.BackColor = Me.\_bgCol

lbl.Text = TEXT

lbl.TextAlign = ContentAlignment.MiddleCenter

lbl.BorderStyle = BorderStyle.FixedSingle

End Sub

lSet(Me.\_h1, 40, "#")

lSet(Me.\_h2, 77, "White")

lSet(Me.\_h3, 77, "Black")

Me.\_h1.Location = New Point(0, 20)

Me.\_h2.Location = New Point(40, 20)

Me.\_h3.Location = New Point(117, 20)

End Sub

Private Sub InitOpeningText()

Me.\_openingText = New Label

Me.Controls.Add(Me.\_openingText)

With Me.\_openingText

.AutoSize = False

.Size = New Size(200, 20)

.Location = New Point(0, 0)

.Font = New Font("Microsoft Sans Serif", 8)

.ForeColor = Me.\_primCol

.BackColor = Me.\_bgCol

.TextAlign = ContentAlignment.MiddleLeft

.Text = "Z00: Starting Position"

.BorderStyle = BorderStyle.FixedSingle

End With

End Sub

Private Sub InitEntryPanel()

Me.\_entryPanel = New Panel

Me.Controls.Add(Me.\_entryPanel)

With Me.\_entryPanel

.Location = New Point(0, 40)

.Size = New Size(194, 0)

End With

End Sub

Private Sub InitScroll()

Me.\_scrollBar = New Panel

Me.Controls.Add(Me.\_scrollBar)

With Me.\_scrollBar

.Location = New Point(194, 20)

.Size = New Size(10, 20)

.BackColor = Color.FromArgb(80, 80, 80)

End With

End Sub

#End Region

Public Sub AddMove(MOVE\_STRING As String, WHOS\_TURN As Alliance)

ReDim Preserve Me.\_entries(Me.\_entryCount)

Me.\_entries(Me.\_entryCount) = New Label

'add handlers

AddHandler Me.\_entries(Me.\_entryCount).MouseEnter, AddressOf Me.e\_label\_mouseenter

AddHandler Me.\_entries(Me.\_entryCount).MouseLeave, AddressOf Me.e\_label\_mouseleave

AddHandler Me.\_entries(Me.getEntryCount).Click, AddressOf Me.e\_label\_click

Me.\_entryPanel.Controls.Add(Me.\_entries(Me.\_entries.Length - 1))

If WHOS\_TURN = Alliance.White Then 'add the # if starting a new row

ReDim Preserve Me.\_numberLabels(Me.\_noOfRows)

Me.\_numberLabels(Me.\_noOfRows) = New Label

Me.\_entryPanel.Controls.Add(Me.\_numberLabels(Me.\_noOfRows))

With Me.\_numberLabels(Me.\_noOfRows)

.Text = (Me.\_noOfRows + 1).ToString

.TextAlign = ContentAlignment.MiddleLeft

.Size = New Size(77, 20)

.Font = New Font("Microsoft Sans Serif", 8)

.Location = New Point(0, Me.\_noOfRows \* 20)

.ForeColor = Me.\_primCol

.BackColor = Me.\_secCol

End With

Me.\_noOfRows += 1

End If

Dim xOffset As Integer : If WHOS\_TURN = Alliance.White Then xOffset = 40 Else xOffset = 117

With Me.\_entries(Me.\_entries.Length - 1)

.Text = MOVE\_STRING

.TextAlign = ContentAlignment.MiddleLeft

.Size = New Size(77, 20)

.Font = New Font("Microsoft Sans Serif", 8)

.Location = New Point(xOffset, (Me.\_noOfRows - 1) \* 20)

.ForeColor = Me.\_primCol

.BackColor = Me.\_secCol

End With

Me.\_entryCount += 1

Me.ResizeAll()

Me.\_selectedLbl = Me.\_entries(UBound(Me.\_entries))

Me.update\_label\_graphics()

End Sub

Private Sub ResizePanels()

Me.\_entryPanel.Size = New Size(Me.\_entryPanel.Size.Width, (Me.\_noOfRows \* 20))

'If Me.\_noOfRows <= Me.\_maxNoOfRows Then

With (Me)

If Me.\_noOfRows < Me.\_maxNoOfRows Then

.Size = New Size(.Width, 40 + (Me.\_noOfRows \* 20))

Else

.Size = New Size(.Width, 40 + (Me.\_maxNoOfRows \* 20))

End If

End With

'End If

'scroll is updated regardless

End Sub

Public Sub ClearAndUpdate(BOARD As cBoard)

For Each lbl As Label In Me.\_entries

Me.Controls.Remove(lbl)

lbl.Dispose()

Next

For Each lbl As Label In Me.\_numberLabels

Me.Controls.Remove(lbl)

lbl.Dispose()

Next

Me.\_noOfRows = 0

Dim whosTurn As Alliance = Alliance.White

Me.\_entryCount = 0

Me.\_entries = Nothing

For Each strMove As String In BOARD.get\_move\_list\_string

AddMove(strMove, whosTurn)

whosTurn = Not (whosTurn)

Next

Me.UpdateOpeningName()

End Sub

Public Sub UpdateOpeningName()

Dim moveList As String = ""

Dim halfPlys As Integer = 0

Dim no As Integer = 1

Dim whitesMove As Boolean = True

For Each move As String In Me.\_gameGUI.getBoardGUI.get\_board.get\_move\_list\_string

If whitesMove Then

moveList &= no & "." & move

whitesMove = Not (whitesMove)

halfPlys += 1

moveList &= " "

Else

moveList &= move

no += 1

whitesMove = Not (whitesMove)

halfPlys += 1

moveList &= " "

End If

Next

'remove the extra space off of the end of the string

moveList = Strings.RTrim(moveList)

FileOpen(1, System.AppDomain.CurrentDomain.BaseDirectory & "\ecocodes9.txt", OpenMode.Input)

LineInput(1)

LineInput(1) ''skips the two credit lines in the file

For i = 0 To 10402

Dim line As String = LineInput(1)

Dim ECO As String = line(1) & line(2) & line(3)

Dim openingName As String = ""

Dim ctr As Integer = 7

Dim c As Char = line(ctr)

Do Until c = "}"

openingName &= c

ctr += 1

c = line(ctr)

Loop

Dim parsedMoveList As String = Strings.Right(line, line.Length - 1 - ctr)

If parsedMoveList = moveList Then

Me.\_openingText.Text = ECO & ": " & openingName

FileClose(1)

Exit For

End If

Next

FileClose(1)

'' < 12ms elapsed

End Sub

Public Sub ResizeAll()

'used in conditions of updating the scroll bar and entry

Me.\_maxHeight = CInt(Me.\_gameGUI.getBoardGUI.Height / 2)

Me.\_maxNoOfRows = CByte((Me.\_maxHeight) \ 20)

Me.ResizePanels()

Me.UpdateScroll()

Me.UpdateEntryLocation()

Me.UpdateScrollLocation()

End Sub

Private Sub UpdateEntryLocation()

If Me.\_noOfRows <= Me.\_maxNoOfRows Then

Me.\_entryPanel.Location = New Point(0, 40)

Else

Dim oflow As Integer = Me.\_noOfRows - Me.\_maxNoOfRows

Me.\_entryPanel.Location = New Point(0, 40 - (oflow \* 20))

End If

End Sub

Private Sub UpdateScrollLocation()

If Me.\_noOfRows <= Me.\_maxNoOfRows Then

Me.\_scrollBar.Location = New Point(194, 20)

Else

With (Me.\_scrollBar)

.Location = New Point(194, Me.Height - .Height)

End With

End If

End Sub

Private Sub UpdateScroll()

Dim height As Integer = 20 + (Me.\_noOfRows \* 20)

If Me.\_noOfRows <= Me.\_maxNoOfRows Then

Me.\_scrollBar.Size = New Size(10, height)

Me.\_scrollBar.Location = New Point(Me.\_scrollBar.Location.X, 20)

Else

'MsgBox(\_maxNoOfEntries) ''USED TO TEST

'MsgBox(Me.\_noOfRows - Me.\_maxNoOfEntries)

Dim excessRows As Integer = Me.\_noOfRows - Me.\_maxNoOfRows

Me.\_scrollBar.Size = New Size(Me.\_scrollBar.Width, 20 + (Me.\_maxNoOfRows \* 20) - (excessRows \* 20))

Me.\_scrollBar.Location = New Point(Me.\_scrollBar.Location.X, 20)

End If

End Sub

#Region "Scroll Event Handling"

'SCROLL EVENTS

Private \_held As Boolean 'used to store whether the mouse was pressed on the scroll wheel

' not using e.Button property since it does not indicate whether the mouse was hovering the bar when clicked

Private \_distFromTop As Integer

Private Sub E\_Mouse\_Down(sender As Object, e As MouseEventArgs) Handles \_scrollBar.MouseDown

Me.\_held = True

Me.\_lastLoc = PointToClient(MousePosition)

Me.\_distFromTop = Me.\_lastLoc.Y - Me.\_scrollBar.Location.Y

End Sub

Private \_lastLoc As Point

Private Sub E\_Mouse\_Move(sender As Object, e As MouseEventArgs) Handles \_scrollBar.MouseMove

Dim newY As Integer = PointToClient(MousePosition).Y - Me.\_distFromTop

If Me.\_held Then

If newY >= 20 AndAlso newY + Me.\_scrollBar.Height <= (Me.Height) Then

Me.\_scrollBar.Location = New Point(Me.\_scrollBar.Location.X, PointToClient(MousePosition).Y - Me.\_distFromTop)

Me.\_entryPanel.Location = New Point(Me.\_entryPanel.Location.X, 60 - Me.\_scrollBar.Location.Y)

End If

End If

End Sub

Private Sub E\_Mouse\_Up(sender As Object, e As MouseEventArgs) Handles \_scrollBar.MouseUp

Me.\_held = False

Me.\_lastLoc = Nothing

End Sub

Private Sub E\_Mouse\_Enter(sender As Object, e As EventArgs) Handles \_scrollBar.MouseEnter

With (Me.\_scrollBar)

.BackColor = Color.FromArgb(80, 80, 80)

End With

End Sub

Private Sub E\_Mouse\_Leave(sender As Object, e As EventArgs) Handles \_scrollBar.MouseLeave

With (Me.\_scrollBar)

.BackColor = Color.FromArgb(60, 60, 60)

End With

End Sub

#End Region

#Region "Entry Label Event Handling"

Private Sub e\_label\_mouseenter(sender As Object, e As EventArgs)

Dim lbl As Label = CTypeDynamic(Of Label)(sender)

With (lbl)

If CTypeDynamic(Of Label)(sender) IsNot Me.\_selectedLbl Then .BackColor = Me.\_hoverCol

.Cursor = Cursors.Hand

End With

End Sub

Private Sub e\_label\_mouseleave(sender As Object, e As EventArgs)

With CTypeDynamic(Of Label)(sender)

If CTypeDynamic(Of Label)(sender) IsNot Me.\_selectedLbl Then .BackColor = Me.\_secCol

.Cursor = Cursors.Arrow

End With

End Sub

Private Sub e\_label\_click(sender As Object, e As EventArgs)

Dim senderLbl As Label = CTypeDynamic(Of Label)(sender) : Me.\_selectedLbl = senderLbl

senderLbl.BackColor = Me.\_selectedCol

Dim index As Integer = 0

For i = 0 To UBound(Me.\_entries)

If Me.\_entries(i) Is senderLbl Then

index = i

Exit For

End If

Next

Me.\_gameGUI.getBoardGUI.show\_ghost\_position(index + 1) 'used to represent the board from the chosen position on the gui rather than the up

' to date board

Me.update\_label\_graphics()

End Sub

#End Region

#Region "Key Events"

Public Sub e\_previewkeydown(sender As Object, e As PreviewKeyDownEventArgs)

If Me.\_entries IsNot Nothing Then

Dim kd As Keys = e.KeyData

If Me.\_selectedLbl Is Nothing Then Me.\_selectedLbl = Me.\_entries(UBound(Me.\_entries))

Dim findIndex = Function()

Dim i As Integer = 0

For Each lbl As Label In Me.\_entries

If lbl Is Me.\_selectedLbl Then

Return i

End If

i += 1

Next

Return UBound(Me.\_entries)

End Function

Dim index As Integer = findIndex()

Select Case kd

Case Keys.Left

If index <> 0 Then

index -= 1

End If

Case Keys.Right

If index <> UBound(Me.\_entries) Then

index += 1

End If

Case Keys.Up

index = UBound(Me.\_entries)

Case Keys.Down

index = 0

End Select

Me.\_selectedLbl = Me.\_entries(index)

Me.\_gameGUI.getBoardGUI.show\_ghost\_position(index + 1)

Me.update\_label\_graphics()

End If

End Sub

#End Region

Public Sub update\_label\_graphics()

For Each lbl As Label In Me.\_entries

lbl.BackColor = Me.\_secCol

Next

Me.\_selectedLbl.BackColor = Me.\_selectedCol

End Sub

End Class

<Serializable()>

Public Class cGameGUI

Inherits Panel

Private \_game As cGame

Private \_boardGUI As cBoardGUI

Private \_whiteClockGUI As cClockGUI

Private \_blackClockGUI As cClockGUI

Public WithEvents \_moveListGUI As cCustomMoveList

Private \_wrapper As cBoardWrapper

Public Function getMoveList() As cCustomMoveList

Return Me.\_moveListGUI

End Function

Public Function getBoardGUI() As cBoardGUI

Return Me.\_boardGUI

End Function

Public Function getBoardWrapper() As cBoardWrapper

Return Me.\_wrapper

End Function

Public Sub New(TARGET As Form, GAME As cGame, SETTINGS As sSettings)

TARGET.Controls.Add(Me)

Me.\_game = GAME

Me.Location = New Point(5, 90)

Me.Size = New Size(TARGET.Size.Width - 10, (TARGET.Height - 90 - 30))

Me.\_boardGUI = New cBoardGUI(Me.\_game, Me.\_game.get\_board, CInt(Me.Height) - 30, SETTINGS, Me)

Me.Controls.Add(Me.\_boardGUI)

Me.\_moveListGUI = New cCustomMoveList(Me, Color.FromArgb(27, 27, 27), Color.FromArgb(-1842205), Color.FromArgb(40, 40, 40))

Me.Controls.Add(Me.\_moveListGUI)

Me.\_boardGUI.Location = New Point(CInt((Me.Width - (Me.\_boardGUI.Width + Me.\_moveListGUI.Width + 10)) / 2), 0)

Me.\_moveListGUI.Location = New Point(Me.\_boardGUI.Location.X + Me.\_boardGUI.Width + 10, 0)

Me.\_wrapper = New cBoardWrapper(Me) 'autmatically adds to game gui in constructor.

End Sub

Public Sub ResizeComponents(NEW\_FORM\_SIZE As Size)

Me.Size = New Size(NEW\_FORM\_SIZE.Width - 10, (NEW\_FORM\_SIZE.Height - 90 - 30))

Dim ls As Color = Me.\_boardGUI.get\_light\_colour

Dim ds As Color = Me.\_boardGUI.get\_dark\_colour

Me.\_boardGUI.Dispose()

Me.\_boardGUI = New cBoardGUI(Me.\_game, Me.\_game.get\_board, CInt(Me.Height) - 30, InterfaceForm.settings, Me)

Me.Controls.Add(Me.\_boardGUI)

'centered

'Me.\_boardGUI.Location = New Point(20, 0)

'Me.\_moveListGUI.Location = New Point(20 + Me.\_boardGUI.Width + 10, 0)

Me.\_moveListGUI.ResizeAll()

Me.\_boardGUI.Location = New Point(CInt((Me.Width - (Me.\_boardGUI.Width + Me.\_moveListGUI.Width + 10)) / 2), 0)

Me.\_moveListGUI.Location = New Point(Me.\_boardGUI.Location.X + Me.\_boardGUI.Width + 10, 0)

Me.\_wrapper.Resize()

End Sub

End Class

Public Class cPieceGraphics

Inherits PictureBox

Private \_bitmap As Bitmap

Private \_heldPieceSize As Integer

Public Function getScaleFactor() As Integer

Return Me.\_heldPieceSize

End Function

Public Sub setScaleFactor(VALUE As Integer)

Me.\_heldPieceSize = VALUE

End Sub

Public Sub New(HELD\_PIECE\_SIZE As Integer)

Me.Region = New Region

Me.\_heldPieceSize = HELD\_PIECE\_SIZE

End Sub

Public Sub UpdateImage(BITMAP As Bitmap)

Me.Location = New Point(0, 0)

Dim b As New Bitmap(Me.\_heldPieceSize, Me.\_heldPieceSize)

Dim g As Graphics = Graphics.FromImage(b)

g.DrawImage(BITMAP, 0, 0, \_heldPieceSize, \_heldPieceSize)

Me.\_bitmap = b

Me.Size = New Size(Me.\_heldPieceSize, Me.\_heldPieceSize)

Me.SizeMode = PictureBoxSizeMode.StretchImage

Me.Image = Me.\_bitmap

Me.Region = Me.getUpdatedRegion

End Sub

Private Function getUpdatedRegion() As Region

Dim pixel As RectangleF = New Rectangle(New Point(0, 0), New Size(CInt(Me.\_bitmap.Height / Me.Height), CInt(Me.\_bitmap.Width / Me.Width)))

Dim r As New Region

r.MakeEmpty()

With (\_bitmap)

For i = 0 To .Width - 1

For j = 0 To .Height - 1

If \_bitmap.GetPixel(i, j).A > 1 Then

pixel.Location = New Point(i, j)

r.Union(pixel)

End If

Next

Next

End With

Return r

End Function

End Class

Public Class cTileBorder

Inherits Panel

Private \_tileSize As Integer

Private \_borderColor As Color

Public Sub New(TILE\_SIZE As Integer, COLOR As Color)

Me.Size = New Size(TILE\_SIZE, TILE\_SIZE)

Me.Location = New Point(0, 0)

Me.\_tileSize = tile\_size

Me.BackColor = COLOR

Me.Region = getUpdatedRegion()

End Sub

Public Function getUpdatedRegion() As Region

Dim tHorizontalBar As RectangleF = New Rectangle(New Point(0, 0), New Size(\_tileSize, 2))

Dim bHorizontalBar As RectangleF = New Rectangle(New Point(0, \_tileSize - 2), New Size(\_tileSize, 2))

Dim lVertBar As RectangleF = New Rectangle(New Point(0, 0), New Size(2, \_tileSize))

Dim rVertBar As RectangleF = New Rectangle(New Point(\_tileSize - 2, 0), New Size(2, \_tileSize))

Dim r As New Region

r.MakeEmpty()

r.Union(tHorizontalBar)

r.Union(bHorizontalBar)

r.Union(lVertBar)

r.Union(rVertBar)

Return r

End Function

End Class

Public Enum interface\_state

single\_player = 0

online

analysis

settings

End Enum

Option Strict On

Public Class InterfaceForm

Public chess As cGame

Public WithEvents gui\_panel As cGameGUI

Public min\_size As Size = Me.Size

Public settings As sSettings

Public state As interface\_state

Public WithEvents main\_account As cAccount

Public WithEvents secondary\_account As cAccount 'the account object used by player2 if a local game is being played, otherwise is set to null.

Public main\_account\_window As cAccountWindow

Public secondary\_account\_window As cAccountWindow

Private media As media\_controller

Public Function get\_game\_gui() As cGameGUI

Return Me.gui\_panel

End Function

#Region "UI"

#Region "Events"

''general ui interaction''

Private Sub FormDoubleClick(sender As Object, e As EventArgs) Handles TitleStripPanel.DoubleClick, MyBase.DoubleClick, TitleLabel.DoubleClick, gui\_panel.DoubleClick

If Me.WindowState = FormWindowState.Normal Then

Me.WindowState = FormWindowState.Maximized

Me.nwseResizeButton.Hide()

Me.hrzResizeButton.Hide()

Me.vrtResizeButton.Hide()

Else

Me.WindowState = FormWindowState.Normal

Me.nwseResizeButton.Show()

Me.hrzResizeButton.Show()

Me.vrtResizeButton.Show()

End If

End Sub

''drag-drop functionality''

Private \_xStart As Integer

Private \_yStart As Integer

Private Sub FormMouseDown(sender As Object, e As MouseEventArgs) Handles TitleStripPanel.MouseDown, MyBase.MouseDown, TitleLabel.MouseDown, gui\_panel.MouseDown

\_xStart = e.X : \_yStart = e.Y

End Sub

Private Sub FormDragDrop(sender As Object, e As MouseEventArgs) Handles TitleStripPanel.MouseMove, MyBase.MouseMove, TitleLabel.MouseMove, gui\_panel.MouseMove

If e.Button = MouseButtons.Left Then

Me.Location = New Point(Me.Left + e.X - \_xStart, Me.Top + e.Y - \_yStart)

End If

End Sub

''title-strip panel''

Private Sub MinMaxClose(sender As Object, e As EventArgs) Handles MinimizeButton.Click, MaximizeButton.Click, CloseButton.Click

If sender Is MinimizeButton Then

Me.WindowState = FormWindowState.Minimized

ElseIf sender Is MaximizeButton Then

If Me.WindowState = FormWindowState.Normal Then Me.WindowState = FormWindowState.Maximized Else Me.WindowState = FormWindowState.Normal

ElseIf sender Is CloseButton Then

Me.Close()

End If

End Sub

''menu-strip panel / poas''

Private Sub ButtonMouseEnter(sender As Object, e As EventArgs) Handles PlayButton.MouseEnter, OnlineButton.MouseEnter, AnalysisButton.MouseEnter, SettingsButton.MouseEnter

CType(sender, Label).BackColor = Color.FromArgb(60, 60, 60)

End Sub

Private Sub ButtonMouseLeave(sender As Object, e As EventArgs) Handles PlayButton.MouseLeave, OnlineButton.MouseLeave, AnalysisButton.MouseLeave, SettingsButton.MouseLeave

CType(sender, Label).BackColor = MenuStripPanel.BackColor

End Sub

Private Sub MenuCollapseButtonClick(sender As Object, e As EventArgs) Handles MenuCollapseButton.Click

If MenuStripPanel.Visible Then

MenuStripPanel.Hide()

MenuCollapseButton.Location = New Point(CInt(MenuCollapseButton.Width / 2), MenuCollapseButton.Location.Y)

MenuCollapseButton.BackgroundImage = My.Resources.arrow\_right

Else

MenuStripPanel.Show()

MenuCollapseButton.Location = New Point(CInt(MenuStripPanel.Location.X - MenuCollapseButton.Width - 6), MenuCollapseButton.Location.Y)

MenuCollapseButton.BackgroundImage = My.Resources.arrow\_left

End If

End Sub

#Region "Resizing"

'NWSE Resize

Private \_ogFormSize As Size

Private \_nwseXStart As Integer

Private \_nwseYStart As Integer

Private \_resizeInProgress As Boolean

Private Sub nwseMouseDown(sender As Object, e As MouseEventArgs) Handles nwseResizeButton.MouseDown

If e.Button = MouseButtons.Left Then

\_ogFormSize = Me.Size

\_nwseXStart = e.X : \_nwseYStart = e.Y

\_resizeInProgress = True

End If

End Sub

Private Sub nwseResize(sender As Object, e As MouseEventArgs) Handles nwseResizeButton.MouseMove

If e.Button = MouseButtons.Left Then

Dim newSize As Size = New Size(\_ogFormSize.Width + e.X - \_nwseXStart, \_ogFormSize.Height + e.Y - \_nwseYStart)

If newSize.Width >= min\_size.Width AndAlso newSize.Height >= min\_size.Height Then

Me.Size = newSize

End If

End If

End Sub

Private Sub nwseMouseUp(sender As Object, e As MouseEventArgs) Handles nwseResizeButton.MouseUp

nwseResizeButton.Location = New Point(Me.Width - nwseResizeButton.Width, Me.Height - nwseResizeButton.Height)

\_resizeInProgress = False

e\_form\_resize(Me, New EventArgs)

End Sub

'vrt resize

Private \_vrtYStart As Integer

Private Sub vrtMouseDown(sender As Object, e As MouseEventArgs) Handles vrtResizeButton.MouseDown

If e.Button = MouseButtons.Left Then

\_ogFormSize = Me.Size

\_vrtYStart = e.Y

\_resizeInProgress = True

End If

End Sub

Private Sub vrtResize(sender As Object, e As MouseEventArgs) Handles vrtResizeButton.MouseMove

If e.Button = MouseButtons.Left Then

Dim newSize As Size = New Size(\_ogFormSize.Width, \_ogFormSize.Height + e.Y - \_vrtYStart)

If newSize.Width >= min\_size.Width AndAlso newSize.Height >= min\_size.Height Then Me.Size = newSize

End If

End Sub

Private Sub vrtMouseUp(sender As Object, e As MouseEventArgs) Handles vrtResizeButton.MouseUp

vrtResizeButton.Location = New Point(0, Me.Height - vrtResizeButton.Height)

\_resizeInProgress = False

e\_form\_resize(Me, New EventArgs)

End Sub

'hrz resize

Private \_hrzXStart As Integer

Private Sub hrzMouseDown(sender As Object, e As MouseEventArgs) Handles hrzResizeButton.MouseDown

If e.Button = MouseButtons.Left Then

\_ogFormSize = Me.Size

\_hrzXStart = e.X

\_resizeInProgress = True

End If

End Sub

Private Sub hrzResize(sender As Object, e As MouseEventArgs) Handles hrzResizeButton.MouseMove

If e.Button = MouseButtons.Left Then

Dim newSize As Size = New Size(\_ogFormSize.Width + e.X - Me.\_hrzXStart, \_ogFormSize.Height)

If newSize.Width >= min\_size.Width AndAlso newSize.Height >= min\_size.Height Then Me.Size = newSize

End If

End Sub

Private Sub hrzMouseUp(sender As Object, e As MouseEventArgs) Handles hrzResizeButton.MouseUp

hrzResizeButton.Location = New Point(Me.Width - hrzResizeButton.Width, 0)

\_resizeInProgress = False

e\_form\_resize(Me, New EventArgs)

End Sub

#End Region

#End Region

#End Region

#Region "Board, move-list, chat etc"

Private Sub e\_form\_load(sender As Object, e As EventArgs) Handles MyBase.Load

settings.load()

Me.state = interface\_state.single\_player

chess = New cGame(Me, settings)

gui\_panel = chess.get\_gui

Me.min\_size = Me.Size

settings.enact(Me, gui\_panel)

media = New media\_controller(Me.select\_button, Me.start\_stop\_button, Me.skip\_button, Me.prev\_button)

cAccount.initialize\_account\_management()

End Sub

Private Sub e\_form\_resize(sender As Object, e As EventArgs) Handles MyBase.SizeChanged

If chess IsNot Nothing AndAlso Not \_resizeInProgress AndAlso Me.WindowState <> WindowState.Minimized Then

chess.get\_gui.ResizeComponents(Me.Size)

Me.TitleStripPanel.BringToFront()

Me.nwseResizeButton.Location = New Point(Me.Width - Me.nwseResizeButton.Width, Me.Height - Me.nwseResizeButton.Height)

Me.vrtResizeButton.Size = New Size(Me.Width - nwseResizeButton.Width, vrtResizeButton.Height)

Me.vrtResizeButton.Location = New Point(0, Me.Height - Me.vrtResizeButton.Height)

Me.hrzResizeButton.Size = New Size(hrzResizeButton.Width, Me.Height - nwseResizeButton.Height)

Me.hrzResizeButton.Location = New Point(Me.Width - Me.hrzResizeButton.Width, 0)

If Me.main\_account\_window IsNot Nothing Then

Me.main\_account\_window.Size = New Size(Me.gui\_panel.getBoardGUI.Location.X - 10, CInt(Me.gui\_panel.getBoardGUI.Height / 8))

Me.main\_account\_window.Location = New Point(5, Me.gui\_panel.getBoardGUI.Location.Y)

End If

End If

End Sub

Private Sub e\_newgame\_click(sender As Object, e As EventArgs) Handles NewGameButton.Click

Dim x As DialogResult = MessageBox.Show("Are you sure you want to start a new game?", "?", MessageBoxButtons.YesNo)

If x = DialogResult.Yes Then

Me.Controls.Remove(chess.\_gameGUI)

chess = Nothing

chess = New cGame(Me, Me.settings)

Me.gui\_panel = Me.chess.get\_gui

Me.settings.enact(Me, Me.gui\_panel)

End If

End Sub

Private Sub e\_undo\_click(sender As Object, e As EventArgs) Handles UndoMoveButton.Click

Dim x As DialogResult = MessageBox.Show("Are you sure you want to undo your move?", "?", MessageBoxButtons.YesNo)

If x = DialogResult.Yes Then

Me.chess.undo\_move()

Me.gui\_panel = Me.chess.get\_gui

End If

End Sub

Private Sub e\_geteval\_click(sender As Object, e As MouseEventArgs) Handles ShowEngineEvalButton.MouseDown

If e.Button = Windows.Forms.MouseButtons.Left Then

MsgBox(New sEval(Me.chess.get\_board).toString)

Else

MsgBox(Minimax.calc\_best\_move(2, Me.chess.get\_board).ToString(Me.chess.get\_board))

End If

End Sub

Private Sub e\_flipboard\_click(sender As Object, e As EventArgs) Handles FlipBoardButton.Click

Me.chess.\_gameGUI.getBoardGUI.flip\_board()

End Sub

Private Sub e\_settings\_click(sender As Object, e As EventArgs) Handles SettingsButton.Click

SettingsForm.ShowDialog()

End Sub

#End Region

Private Sub e\_form\_previewkeydown(sender As Object, e As PreviewKeyDownEventArgs) Handles MyBase.PreviewKeyDown

Dim moveListAcceptedKeys() As Keys = {Keys.Left, Keys.Down, Keys.Up, Keys.Right}

If moveListAcceptedKeys.Contains(e.KeyCode) Then

Me.chess.get\_gui.getMoveList.e\_previewkeydown(sender, e)

End If

End Sub

Private Sub e\_login\_click(sender As Object, e As EventArgs) Handles login\_out\_label.Click

LoginForm.ShowDialog()

End Sub

Private Sub e\_signup\_click(sender As Object, e As EventArgs) Handles sign\_up\_label.Click

SignupForm.ShowDialog()

End Sub

Public Sub e\_login\_successful(sender As cAccount)

Me.main\_account = sender

Me.main\_account\_window = New cAccountWindow(sender)

Me.gui\_panel.Controls.Add(Me.main\_account\_window)

Me.main\_account\_window.Size = New Size(Me.gui\_panel.getBoardGUI.Location.X - 10, 45)

Me.main\_account\_window.Location = New Point(5, Me.gui\_panel.getBoardGUI.Location.Y)

Me.login\_out\_label.Hide()

Me.sign\_up\_label.Hide()

End Sub

End Class

Public Class SettingsForm

Private \_settings As sSettings

Public Sub FormLoad(SENDER As Object, E As EventArgs) Handles MyBase.Load

\_settings.load()

lPanel.BackColor = \_settings.lightSquare

dPanel.BackColor = \_settings.darkSquare

showLegalCB.Checked = \_settings.showLegalMoves

cheatsCB.Checked = \_settings.enableCheats

nightModeCB.Checked = \_settings.nightMode

End Sub

''drag-drop functionality''

Private \_xStart As Integer

Private \_yStart As Integer

Private Sub FormMouseDown(sender As Object, e As MouseEventArgs) Handles TitleStripPanel.MouseDown, MyBase.MouseDown

\_xStart = e.X : \_yStart = e.Y

End Sub

Private Sub FormDragDrop(sender As Object, e As MouseEventArgs) Handles TitleStripPanel.MouseMove, MyBase.MouseMove

If e.Button = MouseButtons.Left Then

Me.Location = New Point(Me.Left + e.X - \_xStart, Me.Top + e.Y - \_yStart)

End If

End Sub

Private Sub SaveButton\_Click(sender As Object, e As EventArgs) Handles SaveButton.Click

\_settings.lightSquare = lPanel.BackColor

\_settings.darkSquare = dPanel.BackColor

\_settings.showLegalMoves = showLegalCB.Checked

\_settings.enableCheats = cheatsCB.Checked

\_settings.nightMode = nightModeCB.Checked

\_settings.enact(InterfaceForm, InterfaceForm.gui\_panel)

\_settings.save()

Me.Close()

End Sub

Private Sub CloseButton\_Click(sender As Object, e As EventArgs) Handles CloseButton.Click

Me.Close()

End Sub

Private Sub lightSquareClick(sender As Object, e As EventArgs) Handles lPanel.Click

Dim x As New ColorDialog

x.CustomColors = New Integer() {}

x.ShowDialog()

lPanel.BackColor = x.Color

Dim intl As Integer() = x.CustomColors

End Sub

Private Sub darkSquareClick(sender As Object, e As EventArgs) Handles dPanel.Click

Dim x As New ColorDialog

x.CustomColors = New Integer() {15866945}

x.ShowDialog()

dPanel.BackColor = x.Color

Dim intl As Integer() = x.CustomColors

End Sub

End Class

Public Structure sSettings

Public lightSquare As Color

Public darkSquare As Color

Public showLegalMoves As Boolean

Public enableCheats As Boolean

Public nightMode As Boolean

''' <summary>

''' Generates a template settings.ini file for the user in the expected file path.

''' </summary>

''' <remarks>Can be used by choice or if a file is not found.</remarks>

Public Shared Sub gen\_default\_file()

'this is only ever called if there is not a file found in the given file path. Therefore, in the following coding, we assume that the file does not already exist.

Try

Dim w As New System.IO.BinaryWriter(System.IO.File.Open(System.AppDomain.CurrentDomain.BaseDirectory & "\settings.ini", IO.FileMode.CreateNew))

w.Write(CByte(255)) : w.Write(CByte(255)) : w.Write(CByte(255)) 'RGB, lightsquare (off-white)

w.Write(CByte(10)) : w.Write(CByte(10)) : w.Write(CByte(10)) 'RGB, darksquare (black)

w.Write(False) 'do not show legal

w.Write(True) 'enable cheats

w.Write(True) 'night mode

w.Close()

Catch ex As Exception

MsgBox("EXCEPTION: " & ex.ToString)

End Try

End Sub

''' <summary>

''' Carries out the neccessary changes on the targets based on the instances current members.

''' </summary>

''' <param name="FORM">The target form to make the changes to.</param>

''' <param name="GAME\_GUI">The target game\_gui to make the changes to.</param>

''' <remarks></remarks>

Public Sub enact(FORM As InterfaceForm, GAME\_GUI As cGameGUI)

With (FORM)

If Not (enableCheats) Then

.UndoMoveButton.Hide()

.ShowEngineEvalButton.Hide()

.FlipBoardButton.Location = New Point(FORM.Width - 82, 42)

.NewGameButton.Location = New Point(FORM.Width - 43, 42)

Else

.UndoMoveButton.Show()

.ShowEngineEvalButton.Show()

.FlipBoardButton.Location = New Point(FORM.Width - 157, 42)

.NewGameButton.Location = New Point(FORM.Width - 118, 42)

End If

If Not (nightMode) Then

.BackColor = Color.FromArgb(238, 233, 233)

.vrtResizeButton.BackColor = Color.FromArgb(238, 233, 233)

.hrzResizeButton.BackColor = Color.FromArgb(238, 233, 233)

Else

.BackColor = Color.FromArgb(27, 27, 27)

.vrtResizeButton.BackColor = Color.FromArgb(27, 27, 27)

.hrzResizeButton.BackColor = Color.FromArgb(27, 27, 27)

End If

If Not (showLegalMoves) Then

GAME\_GUI.getBoardGUI.set\_show\_legal(False)

Else

GAME\_GUI.getBoardGUI.set\_show\_legal(True)

End If

GAME\_GUI.getBoardGUI.set\_light\_colour(Me.lightSquare)

GAME\_GUI.getBoardGUI.set\_dark\_colour(Me.darkSquare)

GAME\_GUI.getBoardGUI.update\_graphics()

End With

End Sub

Public Sub load()

If Not System.IO.File.Exists(System.AppDomain.CurrentDomain.BaseDirectory & "\settings.ini") Then

sSettings.gen\_default\_file()

End If

Dim r As New System.IO.BinaryReader(System.IO.File.Open(System.AppDomain.CurrentDomain.BaseDirectory & "\settings.ini", IO.FileMode.Open))

Dim cR, cG, cB As Byte

cR = r.ReadByte : cG = r.ReadByte : cB = r.ReadByte 'reads lights colour

Me.lightSquare = Color.FromArgb(cR, cG, cB)

cR = r.ReadByte : cG = r.ReadByte : cB = r.ReadByte 'reads dark colour

Me.darkSquare = Color.FromArgb(cR, cG, cB)

Me.showLegalMoves = r.ReadBoolean

Me.enableCheats = r.ReadBoolean

Me.nightMode = r.ReadBoolean

r.Close()

End Sub

Public Sub save()

Dim w As New System.IO.BinaryWriter(System.IO.File.Open(System.AppDomain.CurrentDomain.BaseDirectory & "\settings.ini", IO.FileMode.OpenOrCreate)) 'open or create the .ini file

With Me.lightSquare

w.Write(.R) : w.Write(.G) : w.Write(.B)

End With

With Me.darkSquare

w.Write(.R) : w.Write(.G) : w.Write(.B)

End With

w.Write(showLegalMoves)

w.Write(enableCheats)

w.Write(nightMode)

w.Close()

InterfaceForm.settings.load()

End Sub

End Structure

Imports System.Net.Sockets

Imports System.IO

Public Class Client

Private \_client As TcpClient

Private \_data\_stream As StreamWriter

Public Sub New(host As String, port As Integer)

Me.\_client = New TcpClient(host, port)

\_data\_stream = New StreamWriter(Me.\_client.GetStream)

End Sub

Public Sub Send(data As Stream)

Me.\_data\_stream.Write(data)

Me.\_data\_stream.Flush()

End Sub

End Class

Imports System.IO

Imports System.Net

Imports System.Net.Sockets

Imports System.Threading

Public Class Controller

Public Event on\_recieve(sender As Controller, data As Stream)

Private \_ip As IPAddress '= IPAddress.Parse("192.168.0.22")

Private \_port As Integer

Private \_server As TcpListener

Private \_listener\_thread As Thread

Private \_is\_listening As Boolean

Private \_client As TcpClient

Private \_client\_data As StreamReader

Public Sub New(Optional i As String = "192.168.0.22", Optional p As Integer = 64583)

With (Me)

.\_ip = IPAddress.Parse(i) : .\_port = p

End With

Me.\_is\_listening = True

Me.\_server = New TcpListener(Me.\_ip, Me.\_port)

Me.\_server.Start()

Me.\_listener\_thread = New Thread(New ThreadStart(AddressOf Listening))

Me.\_listener\_thread.Start()

End Sub

Private Sub Listening()

Do While Me.\_is\_listening

If Me.\_server.Pending Then

Me.\_client = Me.\_server.AcceptTcpClient

Me.\_client\_data = New StreamReader(Me.\_client.GetStream)

End If

Try

RaiseEvent on\_recieve(Me, Me.\_client\_data.BaseStream)

Catch ex As Exception

End Try

Thread.Sleep(30) 'reduces system lag, gives up responsiveness on the connection slightly.

Loop

End Sub

End Class