Project – Chess GUI with Engine  
Design  
Harley Reeves-Martin

Contents

[2.0.0 Overall System Design 5](#_Toc8748194)

[2.1.0 Description of the modular structure of the system 9](#_Toc8748195)

[2.2.0 Definition of Data Requirements (Data Dictionary) 10](#_Toc8748198)

[2.3.0 Validation Required 12](#_Toc8748203)

[2.4.0 File Organisation and Processing 13](#_Toc8748204)

[2.5.0 Algorithms for Data Transformation 14](#_Toc8748205)

[2.6.0 Class Definitions and Details of Object Methods 21](#_Toc8748210)

[2.7.0 User Interface for Planned Data Capture and Entry 23](#_Toc8748211)

[2.8.0 Annotated User Interface for Planned Valid Outputs 25](#_Toc8748212)

[2.9.0 Measures for Security and Integrity of the Data 26](#_Toc8748213)

[2.10.0 Measures for System Security 27](#_Toc8748215)

# References

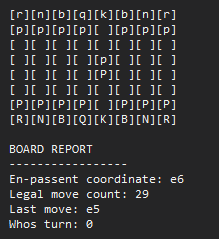
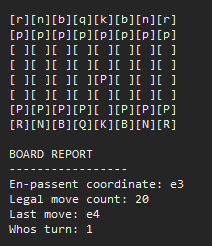
**There are no sources in the current document.**

* 1. Overall System Design

In order to make the development of my program as straightforward as possible, I have decomposed the overall problem into smaller problems that I will be able to deal with individually. After completing each section, I will test to ensure that all components function as intended, and only then move on to the next section.

Stage 1:

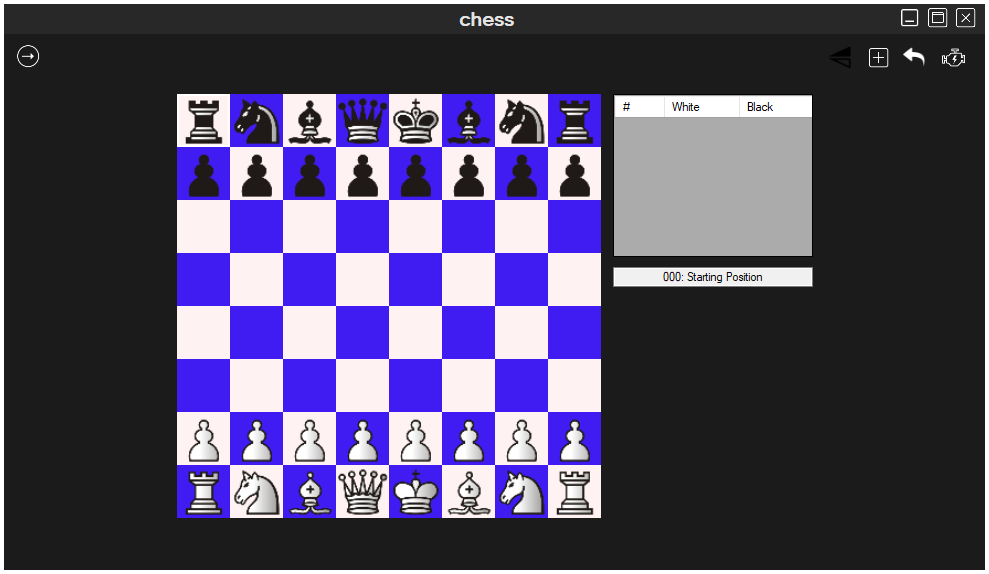
Firstly, I had to develop the classes required for the game to work logically. This involved me creating the following classes: tile, board, piece (abstract with extensions for each type of piece), and an aggregated class - game. In mapping out the methods required to develop the board class, I discovered that I would also have to create a class for a type of board that is not necessarily a legal position, in order to test whether a position (and hence move) is legal or not, I called this class “GhostBoard” since it effectively was the ghost of an already existing board.



I will use this implementation as an opportunity to test my program through playing the game locally through the debug window, I will be able to ensure that all of the methods in the main engine classes function as intended, and run unit tests to ensure that the game logic is programmed correctly (the screenshots above are of a prototype version of my program running in the debug window of Visual Studio 2019).

Stage 2:

The next stage of the design of my solution was developing a graphical user-interface for the user to interact with program through. I believe that the most efficient and manageable solution will involve designing a set of “GUI” classes, that would each be capable of showing a component of a chess game as a control on a windows form. I expect to have to create the following classes: board, pieces, held-piece, move-list, and clock.



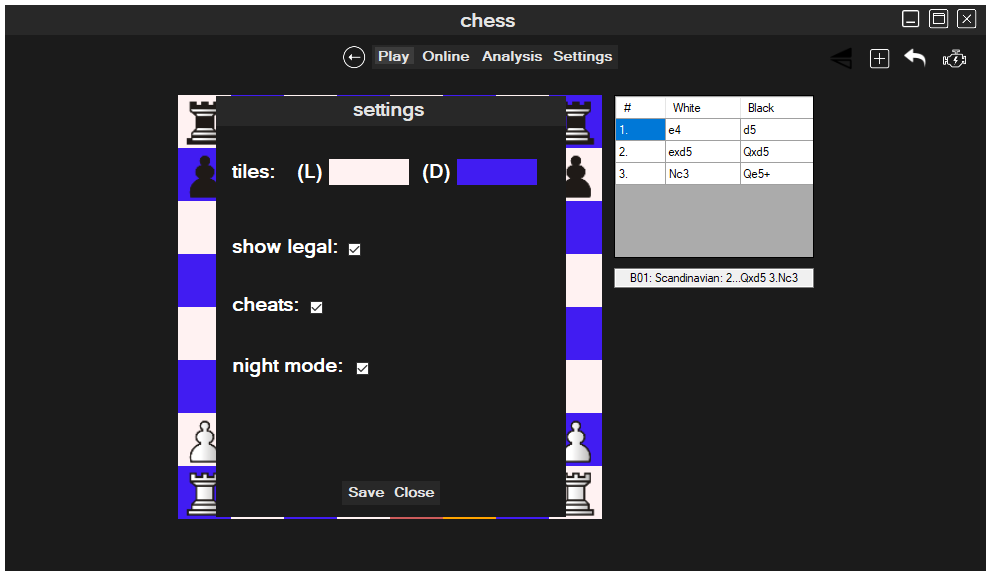
(This is a prototype of the GUI components added to a windows form).

One of the previously stated objectives of my solution is to make the program as interactive, and intuitive as possible. Therefore, when designing each of the GUI classes, I will ensure that there are methods that will support these objectives, I expect this to include: the ability to change the colours of the tiles, showing legal moves or not, showing the selected piece, showing check indicators etc.

Below I have shown two prototypes of how I intend the GUI and settings components to look in my solution.



To allow the user to change preferences for the GUI easily, I will include these options in the settings page of the program.



Stage 3:

The next stage in the development of my program is developing the engine that will analyse a position and heuristically determine the best move from that position. I plan to either use the Monte-Carlo tree search algorithm, or the Minimax algorithm. Both algorithms will provide essentially the same functionality, however the Monte-Carlo tree search will likely prove to be more efficient, since computer resources are limited in the development of my program[[1]](#footnote-1).

Both search algorithms will require me to implement evaluation functions to evaluate the board. Furthermore, the game and GUI classes will both need to be extended to support the engine. The game must allow the user to set-up a match against an engine and the GUI must remain responsive whilst the engine is running, whilst also not letting the user interfere with the position whilst the engine is calculating a position.

Finally, I will need to implement functionality that will allow the user to play games on a local area network. I intend to use the TCP/IP stack in implementing this functionality, operating through sockets in the local network. Once again, the GUI and game classes will need extension to support this feature.

* 1. Description of the modular structure of the system





* 1. Definition of Data Requirements (Data Dictionary)

Access Specifiers:   
I have indicated the access level of each data item using the following symbols:  
(\_): private, (+): protected, otherwise public.  
Additionally, a data-item followed by parenthesis indicates that it is an array, empty parenthesis meaning that the size is undefined.

|  |  |  |  |
| --- | --- | --- | --- |
| Class | Data Item | Data Type | Validation Details |
| cTile | \_isLightSquare | Boolean | No validation required. |
| cTile | \_coordinate | Byte | The coordinate cannot be greater than 255 in order to prevent an overflow exception. |
| cTile | \_piece | cPiece | No validation required. |
| cBoard | \_tiles(63) | cTile | No validation required. |
| cBoard | \_whitePseudoLegalMoves() | sMove | Pseudo-legal moves are validated using column exclusion Boolean functions in each of the pieces generatePseudoLegalMoves method. |
| cBoard | \_blackPseudoLegalMoves() | sMove | See above. |
| cBoard | \_legalMoves() | sMove | The calculateLegalMoves method (contained within the board class) validates the legal moves. The calculateLegalMoves method uses a GhostBoard in order to verify that the moving piece is not pinned. |
| cBoard | \_whoseTurn | Alliance | No validation required. |
| cBoard | \_state | GameState | Whenever an action happens on the board, the calculateBoardState method validates that the current state of the board is valid and updates it if necessary. |
| cBoard | \_moveList | LinkedList(Of String)[[2]](#footnote-2) | No validation required. |
| cBoard | \_enPassentCoord | Byte | The enPassentCoord cannot be greater than 255 in order to prevent an overflow exception. Furthermore, |
| cBoard | \_halfMoveTimer | Integer | No validation required. |
| cBoard | \_plyCounter | Integer | No validation required. |
| cAttackMap | \_board | cBoard | For an attack map to be valid, the aggregated board must also be legal, where there is a move to be made. |
| cAttackMap | \_attackers(63) | List(Of cPiece) | The piece must exist on the board to be added to the attackers’ collection. |
| cAttackMap | \_attackers(63) | List(Of cPiece) | See above. |
| cGhostBoard | \_move | sMove | No validation required. |
| cGhostBoard | \_board | cBoard | No validation required. |
| cGhostBoard | \_isLegal | Boolean | No validation required. |
| cPiece (Abstract) | \_title | Chessman (enumerated type) | Valid if the title matches the type of the current instance of the piece. |
| cPiece (Abstract) | \_coordinate | Byte | Only valid for values less than or equal to 63. |
| cPiece (Abstract) | \_alliance | Alliance (enumerated type) | No validation required. |
| cPiece (Abstract) | \_value | Integer | Negative values are not valid. |
| cKing, cRook | \_hasMoved | Boolean | No validation required. |
| cGameGUI | \_boardGUI | cBoardGUI | No validation required. |
| cGameGUI | \_wrapper | cBoardWrapper | No validation required. |
| cBoardGUI | \_lastMove | sMove | Is always the last item in the list of moves contained within the referenced board. |

2.3.0 Validation Required

The primary validation required in my solution will be the validation of moves played by the user. Since the user can input any move through the GUI, I must ensure that the move is legal before it is played, I have done this with a set of Boolean functions which will determine whether the move is legal, if it is legal then the move will be played, otherwise the move will not be played. To reduce the need to test whether a move is legal, the GUI will only allow the user to select a piece of the alliance that matches the boards whose turn member, this will reduce the number of possible invalid inputs, and therefore make it less likely for the user to make a mistake.

I will also require validation when allowing the user to import FEN or PGN files. Firstly, the open file dialog that prompts the user to select the file to open will utilize a filter, meaning only files marked “.PGN”, or “.FEN” will be shown to the user. This will reduce the chance that the user selects a file that does not match the required format. Unfortunately, there is no effective way of verifying the validity of a PGN or FEN file, therefore, when allowing the user to select these files, the file is assumed to be valid. If an exception is thrown, it will be caught, and the user will be informed that the file was unable to be loaded without providing further information as to why the file could not be loaded.

Furthermore, in creating and signing into accounts, I must validate the user’s inputs to ensure that the program will behave as expected. For example, when creating an account, I must ensure that the username chosen by the user will not interfere with the delimiter selected to be used in the account table file. Furthermore, to ensure that the program is relatively secure, I will make the user create a password that is at least 6 characters in length, and they will need to enter it twice to reduce the chance of a mis-typed password being used. When logging into an account, the user will be asked to enter the username and password of the account they want to log into, this will then be compared against the account table to verify that the account of the given username exists, and then the password hashes are compared. If the user’s input is not valid, then they are not granted access to the account.

Lastly, validation is required in storing and allowing the user to edit the configuration file for the application. The configuration file will provide the user with option such as the option to change the colour of tiles, show legal moves, etc. To generate the settings file, the user will use the settings form, which will be self-validating since only values understood by my program will be selectable through the interface, this self-validation will be achieved by using check-boxes, combo-boxes and other controls that only allow a pre-defined set of inputs. If the user is to edit the configuration file from outside of the application, then I will need to generate a new valid configuration file for the user. This will be done by the following method: once the application is launched, the program will locate the settings file, if the file does not exist or the cannot be read into the application without an exception being thrown, then the existing file will be deleted and a new file will be created with default values for the settings. The user will then be able to edit the settings through the application and the changes will be mirrored in the configuration file.

2.4.0 File Organisation and Processing

My project makes use of multiple file storing and processing techniques.

Firstly, I have used a plaintext file to store configuration files generated by the user. This has meant I have had to create and algorithm to both read and write the file, whilst allowing the user to edit the settings from the application. The order of items stored in the file is non-specific, however it is essential that the file is read and written in the same order. I have used the text-file storage method in this case in order to make alterations to the file easy to make from outside of the program, and it is unimportant to protect the data with some sort of encoding or encryption since the data is unimportant.

Secondly, I have used the text-file storage method in order to store a list of opening names along with the move list that leads to that opening. The file used is the “ecocodes9.txt” file provided by Martin Blume of Arena Chess GUI. The file contains lines in the format of:

1. an opening parenthesis ( { )
2. the opening ECO code (e.g. A00)
3. three spaces
4. the opening name or names where there is more than one
5. a closing parenthesis ( } )
6. The moves of the actual opening in coordinate notation or Algebraic notation

Where each line is an entry to the array of ECO codes. I have wrote an algorithm in the custom move list class that parses the file and determines the opening name for any given opening sequence that is stored in the current instance of board in the game. This function will return a string from parsing this file which is the name of the recognized opening on the board, if the opening is not contained within the file then a default string is displayed instead.

Thirdly, I have once again used plaintext files in order to store portable game notation files generated by the application when the user decides to save a game. The formatting of the file followed the portable game notation conventions as described in the Wikipedia page describing portable game notation here[[3]](#footnote-3).

Additionally, I have used binary files to store a transposition table to be used by my chess engine. My engine must both be able to read and write from the transposition table. I utilized a hash table to store the transposition object in my application, therefore I was able to just use the in-built method in Visual Basic that will serialize the hash table to be stored in a binary file.

Furthermore, I have used delimiter separated values (Extension: “.dsv”) files to store an account table, which is a database consisting of a collection of data in the format: ‘username,password\_hash,account\_location’. The delimiter used is the comma since it should never be used in the username, or account file-path (account\_location), and also the hashing algorithm will never output a string that contains a comma.

The file-path of refers to a binary file which stores a serialized account object. The serialization is handled by the serializable interface provided by visual basic.

2.5.0 Algorithms for Data Transformation

The following section will lay out all relevant algorithms used in my solution to complete the main goal of creating a working chess game. Algorithms will be marked if they are pre-existing.

1. Generate Pseudo-Legal Moves:

Process:

1. A board object is passed into the method, and the piece type is identified.
2. The pieces legal moves are generated depending on the type of piece it is: if it is a pawn then two moves forward are checked for legality, and so are diagonal captures, if it is a bishop then diagonal moves are checked, if it is a queen then the pseudo-legal moves is the union of the moves of a bishop and rook as if they were both in the same position.
3. The generated array of pseudo legal moves are returned.

Pseudo-Code:

(cPiece) overridable function generate\_legal\_moves() as move()

(cPawn) override function generate\_legal\_moves() as move(){  
 check forward once, add if possible;  
 check forward twice, add if possible;  
 check diagonal captures, add if possible;  
 return pseudo\_legal;  
 }

(cQueen) overridable function generate\_legal\_move() as move(){  
 implement bishop move generation algorithm.add;  
 implement rook move generation algorithm.add;  
 return pseudo\_legal;  
 } etc.

1. Generate Legal Moves:

Description:

This method is a member of the board class and will look at the current instance of the class, and then calculate a list of legal moves in the current position. A definition of a legal move can be found here[[4]](#footnote-4).

Process:

1. Loop through all of the pieces in the current players’ piece collection, add the collection of pseudo legal moves for each piece.
2. With the new collection of pseudo-legal moves, ensure each move is legal using a ghost board, ensuring the move does not leave the moving player in check etc. Add all legal moves to the return array.
3. Return the legal moves.

Pseudo-Code:

private function calculateLegalMoves() as sMove(){

moveset as sMove()  
 moveset = this.whoseturn == white ? this.whitepseudolegal : this.blackpseudolegal  
 newMoves as sMove() = null   
 foreach move in moveset {  
 x as ghostboard : x.board = deepclone(this)  
 if (x.islegal)  
 newMoves.add(move)  
 }  
 return newMoves  
 }

1. Generate Best Move\*:

Description:

This is a method of the engine class that will take the current position on a board, and generate the best heuristically determined move to any given depth. The method is co-recursive and relies on the existence of two other methods, “Min” and “Max”. Due to the problem being solved here being an intractable problem, to prevent excessive computing times my application will limit the maximum depth allowed to be 5.

\*This algorithm is based off of the minimax algorithm[[5]](#footnote-5).

Process:

1. The algorithm searches the game tree to a specific depth recursively.
2. All leaf nodes are analysed using a set of evaluation functions.
3. For a “Min” node, minimum value is passed up to the node prior, and for a “Max” node the maximum value is passed up to the node prior, these two neighbour nodes are then compared, and the process is repeated, until the first depth of nodes has a set of evaluation values.
4. If the engine is calculating a move for white, then the node with the maximum value is selected, else the node with the minimum value is selected.

Pseudo-Code:

Function min\_max(depth as int, board as cBoard){  
 highest = -inf;  
 lowest = +inf;  
 current as double;  
 bestMove as smove = new smove;

foreach (move as smove in board.legal\_moves)  
 if whoseturn = black then{  
 current = max(depth-1, board.make\_move(move));  
 compare current to highest, update best move;   
 else  
 current = min(depth-1, board.make\_move(move));  
 compare current to lowest, update best move;  
 }

return bestMove;

(3) Generate Best Move\* (cont.):

The two following functions are required by the min\_max function.

Function Min(depth as int, board as cBoard){  
 dim min as long = +inf  
 if (depth == 0)  
 Return evaluation(board);  
 else  
 foreach (move as sMove in board.legal\_moves)  
 val as double = max(depth -1, b.make\_move(move));  
 if val >= min then min = val;  
 endif  
 return min;  
 }  
   
 Function max(depth as int, board as cBoard){  
 max as long = -inf;  
 if (depth == 0)  
 return evaluation(board);  
 else  
 foreach (move as sMove in board.legal\_moves)  
 val as long = min(depth -1, b.make\_move(move));  
 if val <= max then max = val;  
 endif  
 return max;  
 }

1. Evaluate position:

This algorithm is required by the Minimax algorithm (\*3), it is a heuristic function that returns a value which indicates which player is winning, positive decimals indicate that white is winning, negative decimals indicate that black is winning.

Process:

1. Looks at a position, determines absolute values for material and scope for each alliance.
2. Takes the difference of the white evaluation and the black evaluation, giving an overall evaluation.

Pseudo code:

Function calculate\_evaluation(board as cboard) as double{  
 eval white material : eval white scope  
 eval black material : eval black scope  
 return (white material + white scope) – (black material + black scope)  
 }

1. Make move:

This method is essential to my solution, it will allow the user to make moves and update all necessary members of the board class when a move is made. This includes the en-passent tile, move timer, etc.

Process:

1. Make the move given as a parameter.
2. If the move is a special move (i.e. castle, en-passent capture), handle as required.
3. Update all members of the board.

Pseudo code:

Sub make\_move(move as sMove){  
 make move;  
 if (special\_move)  
 handle as required;  
 update\_board\_members;

}

1. Load/Save settings file:

This algorithm must be able to both read and write to a configuration (.ini) file. If the configuration file cannot be found once it is loaded, then the algorithm must be able to generate a new one. Any changes made to the settings in the application must also be reflected in the settings file.

Process:

1. Load the file. If it does not exist, generate a new default file and load that.
2. Allow the user to edit settings via the settings form, if any changes are made then update the file.
3. Save the file once settings are confirmed.

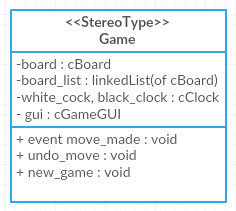
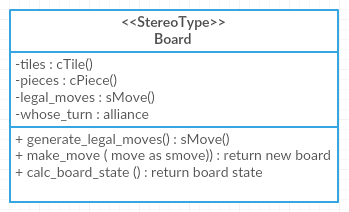
Pseudo code:

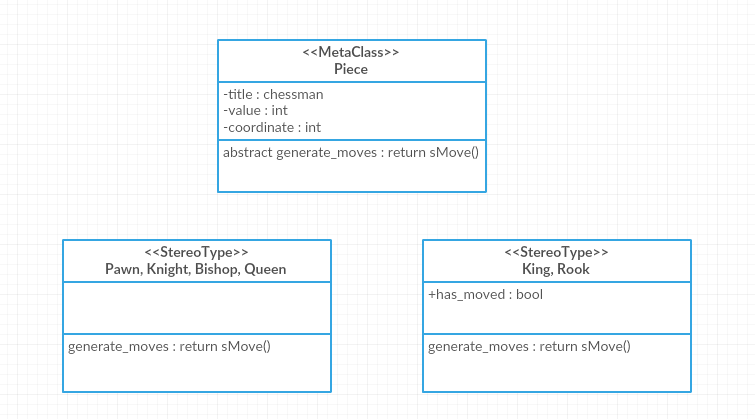
Sub Load\_Settings()  
 me.settings = file.load(settingspath)  
 end sub  
 Sub Save\_settings()

File.save(me.settings)  
 end sub

2.6.0 Class Definitions and Details of Object Methods

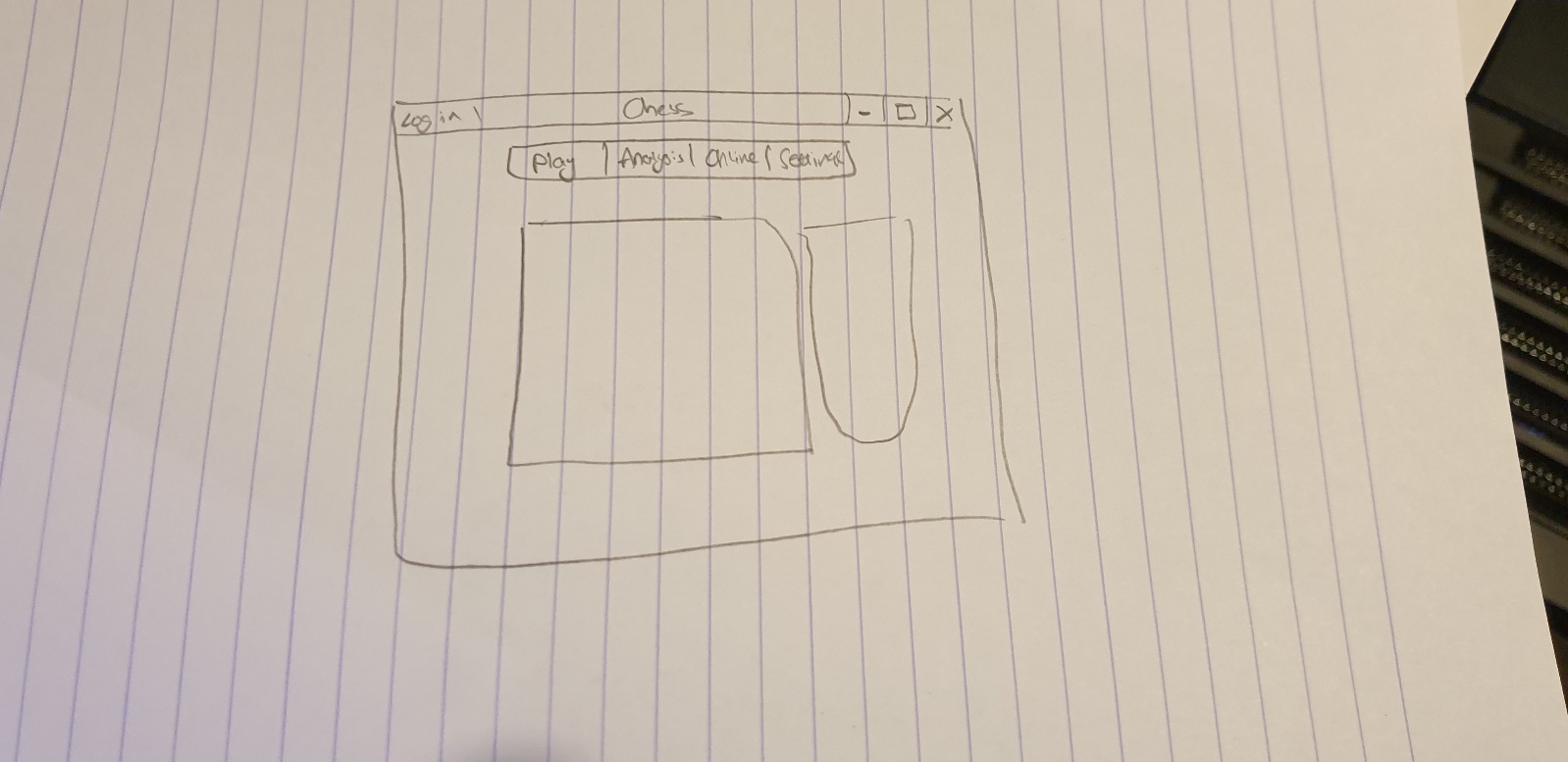
Here I have designed UML diagrams of the important classes used within my program.



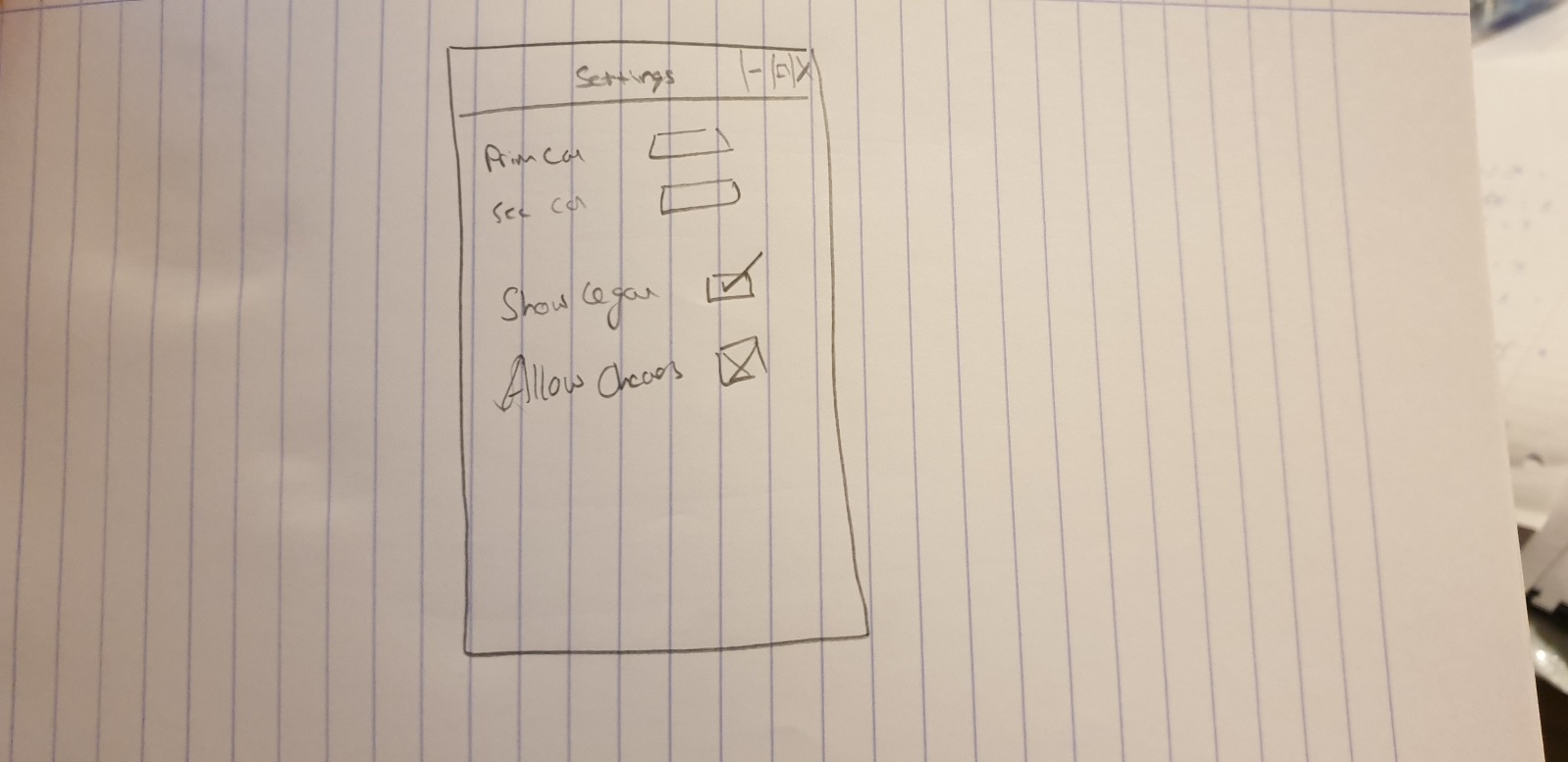


2.7.0 User Interface for Planned Data Capture and Entry

Below are annotated sketches I used to design the graphical user interface of my application.



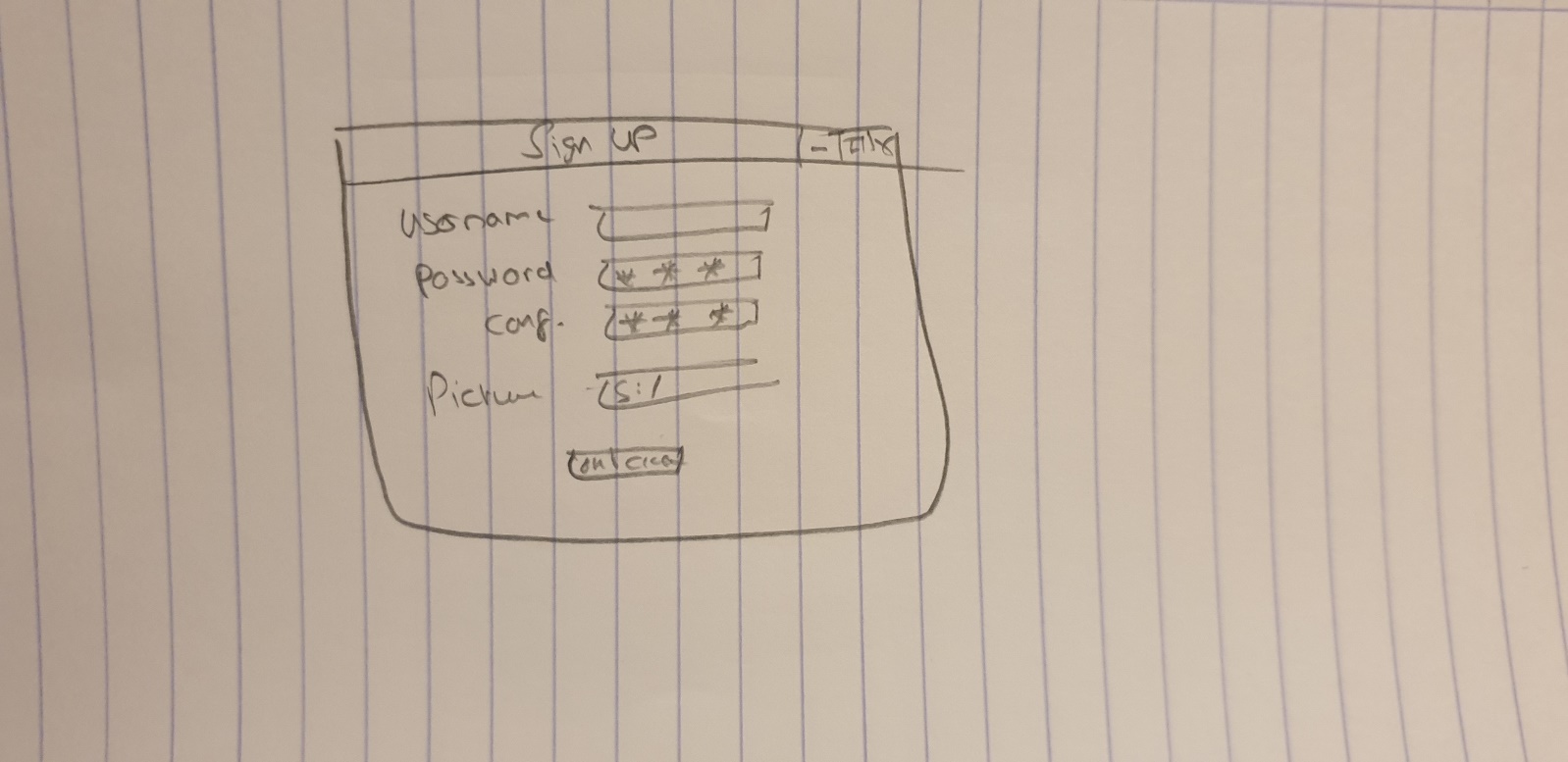
This is a sketch of my application’s main menu; it consists of buttons allowing the user to navigate the program. The main game GUI containing both the board and move list GUI will be displayed here, so that the user can use the main function of the program from the very start. Here data-entry is mostly safe with the exception of moves, the validity of moves therefore must be tested within the board GUI class.



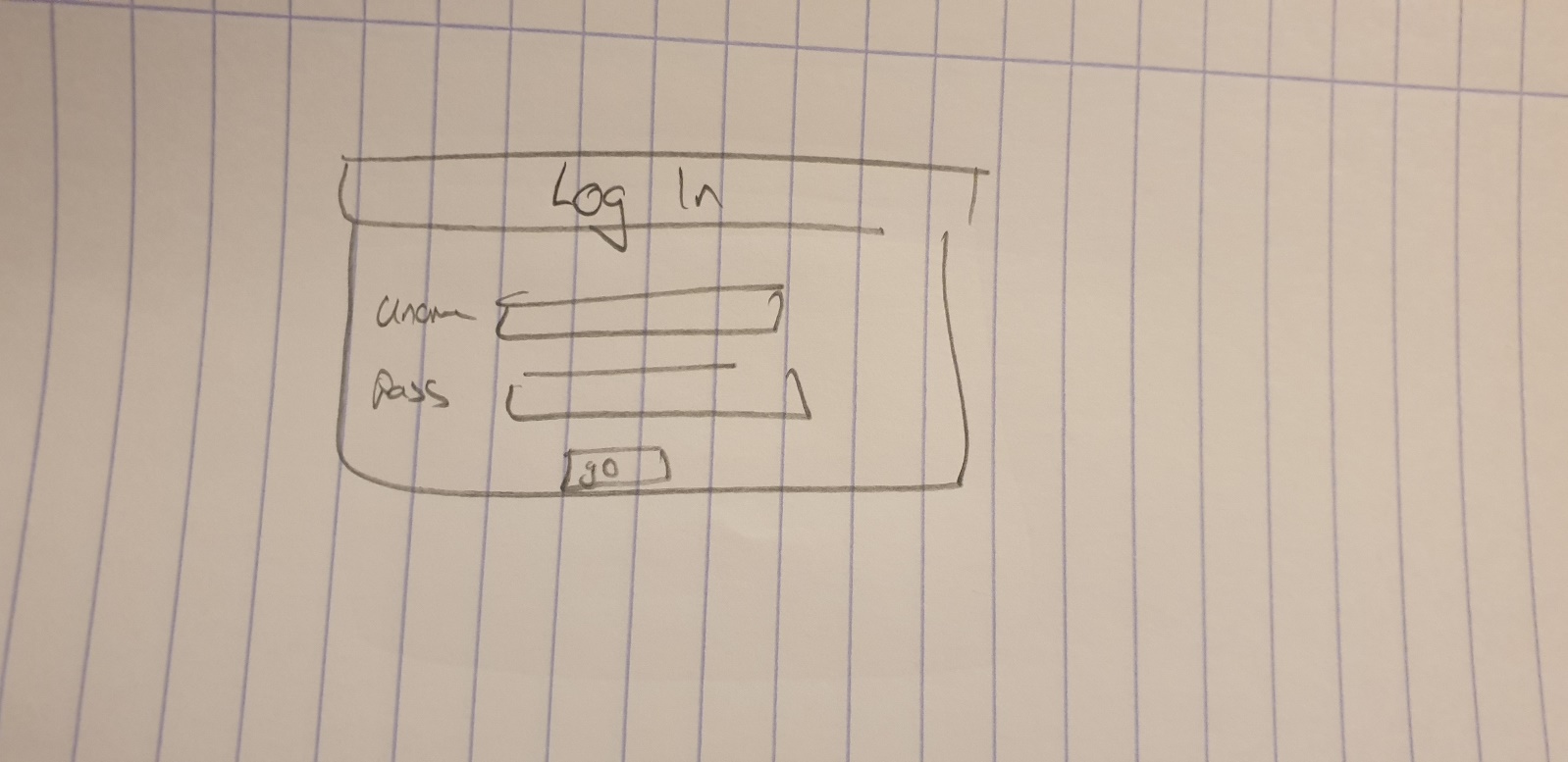
This is a sketch of the settings form in my application. Data-entry is entirely safe in the form since only check-boxes and select colour dialogs are used.

In particular, the settings form will allow the user to: select a colour for the light squares, select a colour for the dark squares, set the Boolean to show legal moves, set the Boolean to allow cheats.

If the user would like to save the changes made, then they must click the save button at the bottom of the form, otherwise any other exit/cancel button will discard the changes made to the settings.

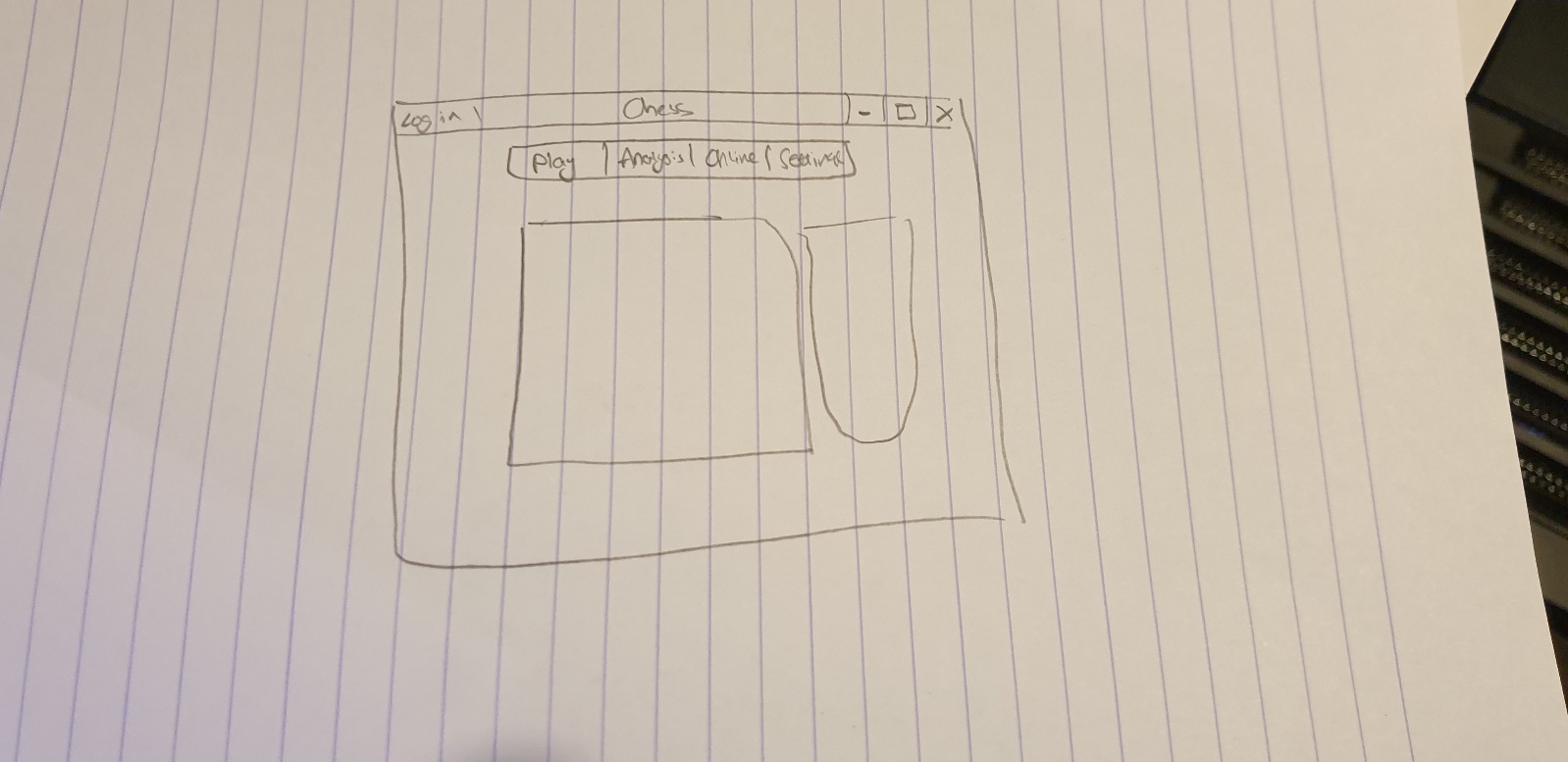
This is a form used to allow the user to create an account in the application. The user is prompted to enter a username, password, password-confirmation and picture location to be associated with their account.

Once the user clicks the Ok button, if the username and password are valid then the account is created and the form is closed, otherwise the account is not created, and the user is given another chance to eitehr exit the form, or create an account.



This is a form that allows the user to attempt to log onto an account. There is no validation when the user is entering the account details, however once the attempt to log on by clicking the Ok button, the inputs are validated in the account classes logon method. If the input was invalid, then the user is given another chance to logon, or close the form.

2.8.0 Annotated User Interface for Planned Valid Outputs

The main output in my program is the graphical user interface for the board and move list GUI. When a move is made through the GUI, the resulting changes to the board object are reflected by the board GUI and by the move list GUI also. The interface will be programmed to scale according to the form size. Furthermore, the user can interact with the move list GUI, and choose to view positions prior to the current position on the board by selecting a previous move. This allows the user to essentially convert the board GUI to show any position, whilst the current position is still stored in the board object.

2.9.0 Measures for Security and Integrity of the Data

Data Security:

To ensure the security of the data stored by my application, I will create regular backups of any user files, and resource files to a server and personal computer. Furthermore, I will advise the user to make backups of any files that they deem to be important.

In the event of any accident such as file corruption, or a natural disaster leading to the loss of data, I will be able to restore important user files since they will be backed up on a server off-site of the original source.

I will also reduce the opportunity for users to maliciously alter files from outside of the program. To do so I will write files in non-plaintext format where necessary, making it more difficult for the user to either do damage to the system or extract private information.

When creating an account, users are advised to make usernames and passwords completely random.

Data Integrity:

Throughout my program I have enabled option strict in VB.NET. This leads to my code being entirely type-safe and so type mismatching will not occur. Furthermore, all data entry points in my program that are not self-validating (which I have done by using controls such as checkboxes and combo-boxes), are validated by Boolean functions that only allow the input to be processed if it is deemed to be valid. If the input is not valid then appropriate error messages are shown, and the user’s input is ignored.

2.10.0 Measures for System Security

In my application the user is able to create a personal account, verified using a username and password. Due to the username and password being sensitive information, it is essential that it is stored properly when saved locally onto a file or database.

Firstly, when saving account files, I have done so using binary files, this obfuscation makes it harder for any malicious user to extract data simply by looking at the files, and therefore protects the user’s data.

Secondly, instead of storing a user’s password in the account lookup table, I have chosen to store the password after it has been passed through a hash function. This makes the original password undecipherable and protects the users account. Due to time constraints, I was unable to implement salting, therefore the method of security I have used is relatively open to dictionary attacks, however if users keep passwords random, the system should be totally secure.

There are no further data protection issues with my project.

1. As discussed in this paper. Kato, Hikari & Fazekas, Szilárd & Takaya, Mayumi & Yamamura, Akihiro. (2015). Comparative Study of Monte-Carlo Tree Search and Alpha-Beta Pruning in Amazons. 10.1007/978-3-319-24315-3\_14. [↑](#footnote-ref-1)
2. A linked list is a generic class (template class), which when declared must have a type passed as an argument. In this case, the linked list is of type string, indicating that every item of the linked list should be a string. [↑](#footnote-ref-2)
3. https://en.wikipedia.org/wiki/Portable\_Game\_Notation#Usage [↑](#footnote-ref-3)
4. <https://www.fide.com/FIDE/handbook/LawsOfChess.pdf> (Article 3, onwards). [↑](#footnote-ref-4)
5. The algorithm is described here: <https://en.wikipedia.org/wiki/Minimax> [↑](#footnote-ref-5)