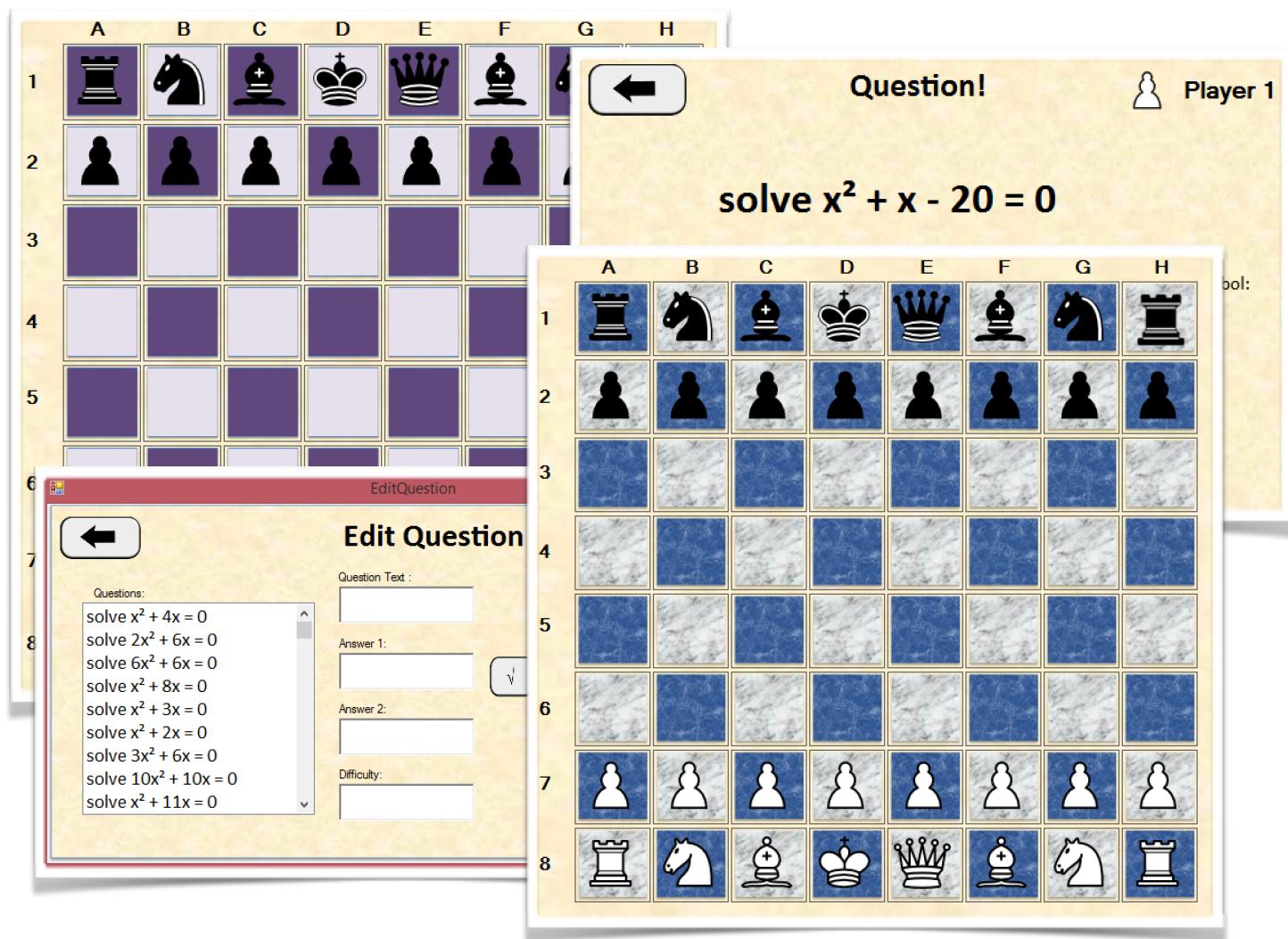


# Comp 4: Chess & Mathematics System

By Harry Graham



Name: **Harry Graham**

Candidate Number: **0394**

Centre Number: **61161**

Centre Name: **Saint Georges C of E School**

# Table of Contents:

<b>Comp4 - Analysis</b>	<b>4</b>
● Background and identification of the problem	4
● Identification of the prospective user(s)	4
● Description of the current system	5
● Identification of end-user needs and acceptable limitations	7
● Object Analysis and Diagram	12
● Objectives for the proposed system	14
● Data source(s) and destination(s)	15
● Data volumes	16
● Realistic appraisal of the feasibility of potential solutions	16
● Justification of chosen solution	19
● Feasibility Study	19
<b>Comp4 - Design</b>	<b>21</b>
● Overall System Design	22
● Description of modular structure of system	25
● Revised Class definitions and details of objects	30
● Revised Class definitions and details of objects (Extended Design)	32
● Description of record structure, file organisation and processing	44
● Validation Required	47
● Identification of appropriate storage media	49
● Identification of processes and suitable algorithms for data transformation	50
● User Interface Design (HCI) Rationale	62
● Planned User Interface Designs	65
● Measures planned for security and integrity of data	77
● Measures planned for system security.	78
● Overall Test Strategy	78
<b>Technical Solution</b>	<b>79</b>
Main Menu	80
BoardUI (GameBoard)	87
PromotePawn	108
AskQuestion	110
AdminLogin	117
AdminPage	119

ViewQuestions	122
AddQuestion	124
DeleteQuestion	128
EditQuestion	131
Object orientated Classes	135
<b>System Testing</b>	<b>183</b>
<b>System Maintenance</b>	<b>226</b>
• System Overview	227
• Procedure and Functions list	236
• Applications used in my project	252
• Annotated Code Reference	255
• Any code not written by myself	255
• Detailed Algorithm Annotation	255
• Variable Lists	275
• Installation Logs	288
<b>Appraisal</b>	<b>292</b>
• SMART Objectives Evaluation	292
• Questionnaires and User Feedback	300
• Analysis and Evaluation of User Feedback	307
• System Extensions	308
• Reflection	310
<b>Appendix</b>	<b>311</b>
• User Manual(s)	311

# Comp4 - Analysis

## ● Background and identification of the problem

I am planning to code a chess game project, for my school's chess club. The chess club runs on thursday afternoons after school. It runs from 3:15 - 4:30 so lasts just over an hour. The club runs in one of the ICT rooms, with tables in the middle of the classroom. Students come after school finishes and they setup the current chess boards that are in use. The students allocate matches and use paper and pen to record scores and times of the games. The timing of which is done by stopwatches or the students mobile devices.

This game will be tailor made and will include specific features based on what the club would like to see. This programmed solution can incorporate many features that online software may not have. This means my solution would be the schools own chess software. Current systems are all physical chess boards and pieces which means pieces are easily lost and there are also only a finite number of boards. A computer solution to this problem would mean the software can run on multiple computers and would be clear and suitable (opposed to various free chess softwares out there which can be clunky and not specific enough).

Another side to this project, provides a solution to AS maths revision. The bridge between GCSE and AS mathematics is a massive jump and from experience, I know how tedious learning can be at this stage. "Head start to AS maths" books were like foreign languages and were hardly effective for learning. Implementing revision into a fully functioning chess game makes the learning more enjoyable whilst maintaining the logical reasoning that is needed to excel at both learning maths and the game of chess. The solution to this problem will be interlinked with chess by having a mode which associates answers to maths questions with the moves of the game (I.e by deducting time from the timer value of each player).

## ● Identification of the prospective user(s)

The end users are ultimately the school's chess club which consists of the students who attend it. They will be the main source of feedback and requirements however Mr C Coetze Runs the chess club and so he will be the main end user as they can then distribute the software onto the schools computers. This person will be the main source of feedback and appraisal, however the students that attend the club are means for just as valuable feedback for the project.

A secondary prospective user(s), would be the maths department - mainly the students. A large percentage of chess club attenders also study mathematics at some level. Mainly GCSE but there are a handful of year 11/year 12's too.

After a brief survey, 84% of the people who attend chess club study mathematics and an average of 76% of mathematics students (years 10-12) can play chess. So a strong correlation is evident between people who play chess and people who study maths. I will also have words with the head of the maths department to get a clearer idea on how I maximise the learning of AS content and implement this into my solution. After speaking with Mrs N Shibli about the transition, she said that many students really struggle with the step up and many do not bother reading the text books they give out to prepare them. So this solution will hopefully cater for these problems also.

- Description of the current system

### Chess:

The chess club runs on Thursday afternoon after school from 3:15 to 4:30. In one of the schools ICT suites which has tables in the middle of the room. This is the only room available but lucky enough there are tables in the room too and not just the computers - which are around the edges of the room.

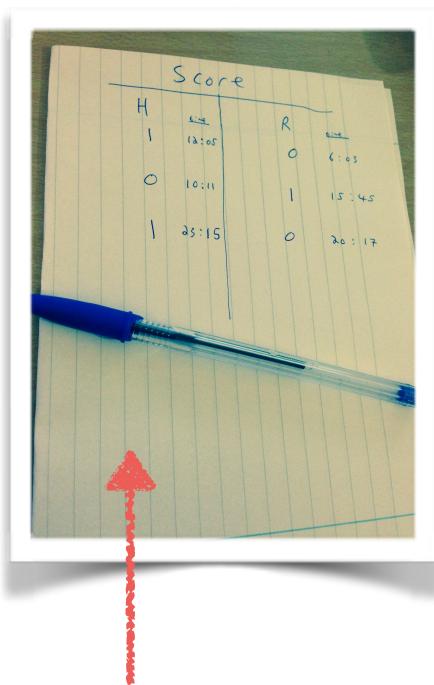
The current system for the club includes only physical chess boards and pieces. Students use the boards alongside their mobiles or stopwatches to time the game (pausing the time when they have made their move).

#### ***The complete process is as follows:***

- 1) The students come in from afternoon registration.
- 2) The students get out the chess boards and set up the game.
- 3) The students then set the stopwatches to a game of time value should they want a timed game.
- 4) They then play the game until one player has won they then record scores on a piece of paper.
- 5) Players rotate or play again and continue until the time is up.

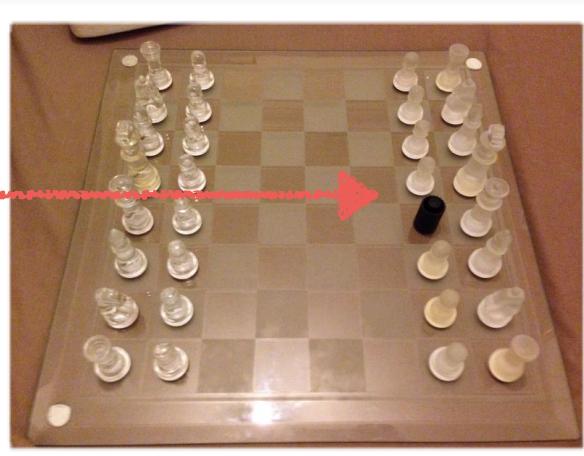
There are lots of problems with this system which has become apparent after speaking to some of the students.

- The club is only open for 1 hour after school, so time is wasted setting up the boards and stopwatches etc which then eats into gameplay time.
- Pieces of the board are often missing and as a result, less boards can be used or an alternative object has to be used for the missing piece.
- There are only a finite number of chess sets so sometimes students have to wait to play a game.
- There are only a finite number of stopwatches so some students need to use their mobile devices to time the game.
- There is too many variables needed for a fully functioning game of chess (ie stopwatches, mobiles, the board, the pieces, pen and paper etc).
- The students having to use their phones as external devices can lead to distractions, un-ethical game play and so forth.



**The current scoring system, pen and paper with recorded times from stopwatches**

**Missing pieces, replaced by objects such as pen lids.**



## Problems with other online solutions:

- Many of the good online chess software solutions are not free
- The free solutions are often clunky and unclear to play with.
- Not all features wanted will be included in the online solutions
- many of these websites are blocked/filtered through the school's network proxy.
- Most software now contains lots of advertisements for other online companies which is disruptive.

By coding a solution to these problems myself, I can create a chess game that eliminates the needs for all the external devices and also the physical boards. By having software on the computers, a game can be setup within seconds which means there is more time to actually play the game instead of wasting time setting up the boards.

Pieces cannot be lost and the software can be on as many computers as the end user wishes - this also means more students can play without having to wait. Features can be included in the solution to make the software tailor made and how the club wants it. Also seeing as the club is run in computer room, this solution would be extremely convenient.

**Many online solutions can be tedious to look for and use...  
Most are blocked by the school's network**

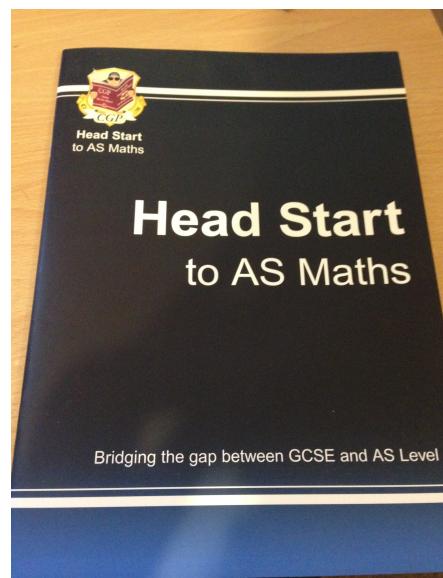
Maths:

The current maths system is all paper based learning. Students in year 11 who have decided to do A-level maths next year are given a "head start to AS maths" textbook which they are strongly advised to look through and prepare from to ease the difficulty transition.

The maths department holds a few sessions before the summer break to start the content early.

There are lots of problems with this system which has become apparent after speaking to the head of maths.

- Students forget everything from the sessions because they have the summer break and do not revise over the summer. So the sessions held have become redundant.
- The text books are quite tedious to learn from as the students are not very independent at this stage and so cannot use the books effectively.



- Most resources are not engaging enough to keep up revision after the pre-summer sessions.
- High end GCSE knowledge (such as factorising quadratics) is often forgotten by the time the students come back after summer so the first term is normally spent going back over GCSE material which is incredibly inefficient.

By coding a chess-maths solution, The students will have an engaging piece of software which they can play at home and also at the schools chess club (during term time). This will keep their high end GCSE knowledge fresh and will also maintain the newer AS content they learned from the pre-summer sessions.

Because chess engages the brain in logical thinking, the maths will be deposited in the brain more efficiently as their brains will be active.

### ● Identification of end-user needs and acceptable limitations

To get a valid view on what the key requirements for this solution are, I interviewed a couple of students who attend the club and see what they thought about the current system:

#### Lucas King (year 9)

“The club is on for an hour after school, do you find that time gets wasted by setting up the boards etc?”

Lucas: Yes, definitely. It can be really frustrating when you really get into a game and we have to pack up.

“What problems are there with the club?”

Lucas: Well, some of the boards have missing pieces so we use pen lids to replace them which looks tacky. Also we only have about 5 sets so sometimes you have to wait to play a game. Time is always pushed which is rubbish when you play chess, I like to take my time but I always have to play 15-minute timed games

“What would you like to see in a computer based chess game?”

Lucas: It would need to have different timed games so we can select how long the game runs. Also it would be good if the software could highlight possible moves, not many do that and its good for when you are learning the game like I was last year. The timing of the game must run and pause when the players have finished there move. we do this with stopwatches so this would need to be on the software too. Maybe different options of gameplay would be good, such as a non timed mode and a mode where you can play against the computer.

“Do you think it would be good to have a maths revision mode as well to help the transition to AS?”

Lucas: It would be a brilliant idea as long as it is a separate mode of gameplay otherwise that restricts the ams to people who study maths. Also make sure the right answers are shown if the players answer is wrong - some revision programs I've seen just tell you that you're wrong!

Kyla Caller (year 10)

“The club is on for an hour after school, do you find that time gets wasted by setting up the boards etc?”

Kyla: Yes time is always wasted setting up the games, especially if there are missing pieces as we have to find alternative pieces! - my favourite has been a wine gum for a pawn!

“What problems are there with the club?”

Kyla: As well as the timing, the boards themselves are not very appealing and we don't have loads of them either so I always hope that not many people come so I can get a game! the way we time the games is really stupid, the stopwatches never stop properly and sometimes you can accidentally reset the timer which ruins the whole game - very frustrating!

“What would you like to see in a computer based chess game?”

Kyla: I think the ability to change the style of the pieces would be cool as the same pieces can get boring to look at. Timed modes must be on there as we only have an hour so we all use timed games also timed games make you think fast! It would be really cool if you could play the same game across two computers!

“Do you think it would be good to have a maths revision mode as well to help the transition to AS?”

Kyla: Yes definitely! I hope to do A level maths so that would be good for when I come to study it! - it would be good if there was some high level GCSE stuff in there too as a little recap swell as testing new things (such as quadratics etc).

Sim Rakkar (year 12)

“You decided to do A-Level maths, did you prepare at all over the summer?”

Sim: I did try to prepare, the school gave us head start text books to try and learn from but I found it really hard to actually learn the new things from it, so I didn't really manage to prepare at all.

“How was the first term of maths, can you notice the step up in difficulty from GCSE?”

Sim: The first term was quite difficult, I was actually debating whether to drop maths or not. The step up from GCSE is massive! I found that my GCSE maths needed to be solid before learning any new maths which was my downfall as I had forgotten some things from last year. After the first term and some perseverance, things got better, but I just feel that initial starting point could have been made a lot easier.

“What advice would you give to year 11's currently considering maths?”

Sim: None of us do it, but revise what you already know during the summer just to make sure you do not struggle with simple stuff when you actually start the course. if you do that then you will instantly find it a lot better than I did. Maths really does need time and effort put into it.

“Im planning on coding a chess game software which incorporates maths revision - do you play chess, what do you think of this idea?”

Sim: I love chess! I play with my granddad at weekends. I think that would be a really good way to revise if students use it during the summer. I can't see everyone using it but I definitely think it would make a big difference to some people. I would definitely use it!

"What sort of maths questions do you think will be most effective?"

Sim: Hmm, I think the more quick fire / short questions would be most effective and would mean a lot of practise gets done too. Things like quadratics and surds would be a big one to use.

From these three interviews with students I can see that the timed mode of the game is a must. Additional features such as different styles of pieces or playing across two computers are things I could look into.

Highlighting squares for possible moves in the chess game was a brilliant idea and so I will be eager to implement this into the solution.

The idea of being able to undo a move is a good idea as it could be very frustrating making the wrong move.

From Sim, I can clearly see how the students view the transition of maths and he's responses were hopeful. It was evident that revision of existing knowledge was a must and quick fire questions seem to be the most effective.

Needs:

- Time game mode with various values (ie 15 mins, 25 mins etc)
- Non-timed mode
- Highlighting of possible moves
- Timer must pause after players turn.
- Scoring section to record scores
- Some GCSE material for the maths mode
- An "undo" button to allow players to retract a move.

To gain further knowledge I spoke to both Mr Coetzee and Mrs Shibli to see what they thought on my ideas for a proposed system:

#### Mr C Coetzee:

"The club is on for an hour after school, do you find that time gets wasted by setting up the boards etc?"

Yes, I would say at least 15 minutes is taken up until everyone is into a game. However that is not including time when there are problems like missing pieces or there are not enough boards to go around one particular week.

"Have you ever considered downloading an online chess software for the computers around the room?"

Yes, but the problem is many software is blocked from online resources by the school's network and proxy server. Also many of the good chess software costs money which is expensive for the school. All the free software - as I said - are blocked, but they also have lots of advertisements which is distracting, students will end up on the web opposed to playing chess.

"Are there any problems with the process of gameplay in the club?"

The students get on quite well with the resources we have. They seem to record scores on paper fairly efficiently but it is a bit of a pain doing it manually. The main problems arise when the stopwatches accidentally reset - causes a lot of arguments amongst the players. Also some of the slower students - who like to take their time - do not get enough time to finish due to the 15 minute expenditure at the beginning. I usually help out by putting a timer on the board (through the projector) which is then shared by games. But if the students want individual timers (which are paused after each move) then they have to use stop watches. I banned mobile phones as stop watches as it caused too many distractions.

"If I were to code a computer chess system, what would be the main features you would like to see?"

Definitely multi-functionality in terms of time and style. It would be great if the students could choose a time span for a game which would then pause after each of their moves. A lot of our chess boards don't look particularly great, so a flexibility in style of the board and pieces would be great if possible. We have some weaker players so anything which aids moving (which of course can be turned off for more experienced players) would be a bonus.

"A secondary part to this solution is the aid of maths revision - GCSE/AS level particularly. Do you think this would benefit some of the chess club members?"

As long as this mode can be turned on or off then yes that is brilliant. Not all post-16 members of the chess club study mathematics, obviously all the younger years do (years 10 and 11 especially). This would definitely be useful to them. If you pass this software onto the maths department, maybe it'll encourage more people to play chess!

Mrs S Shibli:

"What problems do you have with the transition from GCSE to AS in maths?"

Well, despite being a big step up in term of difficulty, I find that students core GCSE knowledge is gone by the time they return after summer and so we spend a fair bit of time consolidating previous knowledge before we even start the AS content. This includes factoring quadratics, rules of indices etc.

"What do you do to try and overcome these problems?"

There is only so much we can do but we try to encourage the students to work through the head start to AS maths book which will in turn keep their GCSE knowledge in their brain but also attempt learning new ideas. We also start them off early with the AS content (with indices and surds) before the summer break in the hope that it will ease the transition but no year is ever the same.

"Do you think a chess game software with a built in mode of mathematics questions, would encourage the students? or at least keep their existing knowledge present?"

Yes definitely! Anything with a game associated will create more of an interaction which is good for learning. I know quite a few students who play chess actually who would be interested in this.

"What should be the main topic of maths to focus on in the chess game?"

Existing GCSE knowledge as mentioned would be vital to keep that in their brains but also a strong impression on indices and surds would be brilliant as the C1 exam requires students to be slick with surds, so an early impression on these would be great. I would have difficulty levels if possible to give a sense of improvement and challenge - key features of both learning and testing.

From Mr C and Mrs S, it is clear that the chess game needs to be flexible with its controls/options and from the maths point of view, Questions need to be focused on surds, quadratics and indices.

The must have's are definitely a fully working 2-player game of chess with flexibility of style and options to toggle on or off, which aid inexperienced players. Then the mathematics section should be a further option which gives short fire questions within the game to help the students revise. These questions should be difficulty based, adding to the functionality of the system.

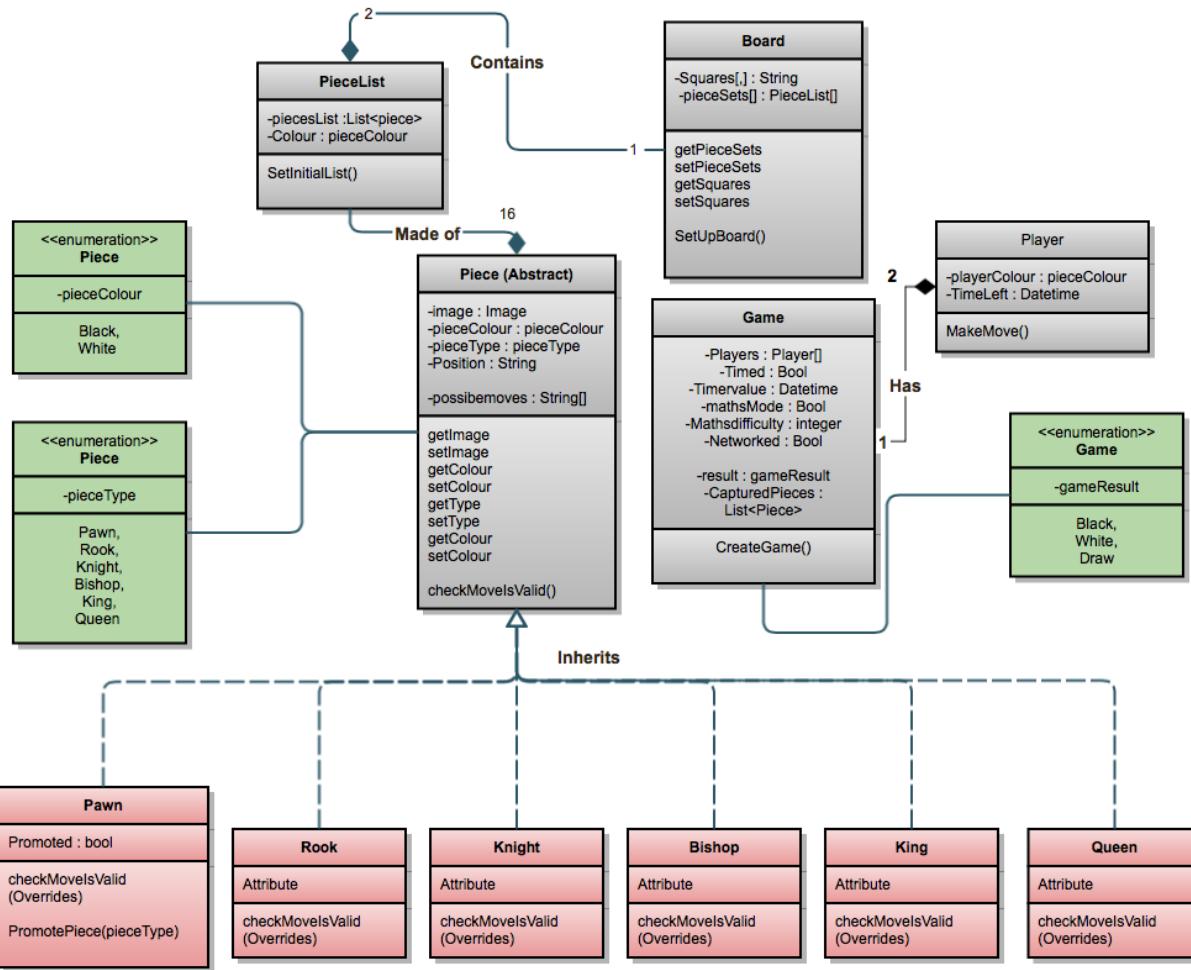
Some limitations would be the computer game mode in which you play against the computer. This would require a heavy artificial intelligence engine which would make the project size too large for the time limit. Also the computer chess AI has been refined over many years so mine would not simply work as efficiently and would probably slow the game down.

The idea of playing across two computers will involve TCP/IP and use of sockets which may well be achievable but this may also turn out to be a limitation which cannot be done due to time constraints.

As for the mathematics section, I may not be able to include maths for ALL years but top end GCSE and AS basic level most certainly.

## ● Object Analysis and Diagram

This is a basic object analysis diagram of the proposed chess system. The chess game will be piece-centric, that is to say, the game will be predicated upon the moving and management of pieces as their individual objects as opposed to a square/board centric approach in which the board is the main manager, with pieces just sitting on the board.



This approached seemed more logical to me and my design thoughts so this way the way I allowed to go. Of course there are many different approaches to designing a chess system but as long as I stick my design path rigorously, the outcome should be just as effective.

With a piece-centric approach, each piece manages itself by means of properties such as position, colour, type etc. Then the main idea is that these pieces sit on a board which just manages the state of the game.

This could have been different as there are many different ways to go about programming chess, some of which can get very mathematical. A ‘bitboard’ was quite a popular method years ago in which an array of 64 bits represented the board (a 1 representing a square is occupied, a 0 representing the square is empty). Other approaches would be the board being the main reference, doing all the hard processing and would simply be scanned and processed accordingly.

When breaking a game of chess down into logical pieces i found that a piece-centric approach seemed to be best suited to the way i was analysing the problem and so this is the direction i took.

In a game of chess, there are four main things: The board, the pieces, the players and the game. These 4 things are linked with each other but they are separate entities and deal with different things.

The board represents an 8x8 set of squares which make up the game board. Each of those squares are either occupied or not occupied - they also have an x-coordinate and a y-coordinate.

On top of the board we have two sets of pieces: black and white pieces. A piece object has a colour, a position, a type, and a set of rules with which it can move.

A player makes moves and moves pieces on the board. 2 players in a game take turns to move pieces, which can be a set of valid movements for that specific piece. Each player has a timer value (if timed game) which is reduced when they make their moves.

A game contains two players and controls the current state of the game. These are such things like “is black in check” and determines whether the game has been won, lost or drawn.

By breaking these four things down we can see that a game contains two players, these two players make moves which involve pieces on a board. The four categories link together whilst controlling separate parts of the system.

These things can then be broken own further into smaller chunks. A piece can be one of 6 types: Pawn, Rook, Knight, Bishop, Queen and King. These objects all do the same thing in the sense that they all have a position, colour, type etc. so they all follow under the piece object. Then each individual type of piece contains individual rules for which they can move (i.e a rook can only move horizontally while a bishop can only move diagonally).

On the board there are two sets of pieces; white and black. therefore a ‘pieceset’ will contain 16 white or black pieces.

A game can be one of many modes which all do the same fundamental thing but with differing properties. (i.e a timed game is a normal game with a timer property). There a game can be split up into ‘timed’, ‘non- timed’, ‘maths’ and ‘non maths’ each have attributes for timer value, maths difficulty etc.

By breaking the game of chess up into these logical pieces, a clear model can be made of how the programmed solution will be based on.

- **Objectives for the proposed system**

### Chess

1. The program should enable a fully functional game of chess to take place between two human players on the computer.
2. In addition to the standard moves of all the chess pieces; support needs to be included for castling.
3. The functionality of the system should cater for the promotion of pawn pieces (a pawn may be promoted when it reaches the other side of the board). This option should clearly be presented to the user in another window.
4. The game must provide two game modes – timed and non-timed. Such modes can be toggled from the game setup section, before the game commences.
5. The game will have an option to change the time variations when using the timed-mode (Increments of 5 minutes up until 60 minutes) that the user can select from.
6. When playing the timed mode, the timers for each player must pause when that player has finished their move. The timer must pause within 2 seconds of that player completing their move.
7. The chess board will highlight squares to show possible move when left clicking on a piece, to aid in the learning of the game. This feature can be enabled or disabled during the in-game menu.
8. There should be appropriate feedback to let the user know who's turn it is, this should be visual and clear so that the turn of the game can be identified without having to click a piece.
9. There should be appropriate feedback of a game result, when a player has been checkmated. A resolution to the game must be made.
10. The Board setup should be dynamic, with Player 1's pieces, whatever colour, at the bottom of the screen (moving upwards).
11. The user interface and chess board must be clear and uncluttered. When the user makes a move, appropriate visual and auditory feedback is required. For example, the user needs to be clear what piece they have selected, where the piece will move to, and the impact of this move.
12. There should be a dynamic list of previous moves shown in unicode notation on the board form, this should be clear what piece moved where in a particular move.
13. There should be a fully functioning “undo” button which allows the user to retract a move that is made for whatever the reason. This button should also therefore delete the move from the list of previous moves shown to the user (objective above).

## Maths

14. The software should have a mathematics mode in which randomised mathematical related questions are presented when a given player makes a move.
15. Each question should be retrieved from a bank of questions and displayed within 2 seconds. These questions should be stored in an easily accessible area which is quick to communicate with (Such as text files).
16. These questions should be retrieved randomly (from the appropriate difficulty of questions) to avoid any sequence to the game and so the games are different every time.
17. Each mathematical question will have an associated test level. Such levels can be selected from the game setup section, before the game commences.
18. The questions asked in the maths mode should be sorted into difficulty which can be selected as a difficulty level before the game starts - the time deducted for a wrong answer should vary with difficulty.
19. A scoring system will keep track of correct and incorrect answers within a game.
20. If a player answers a question wrong, the solution must be shown and time deducted from the current players score. The time deducted should vary with the level of difficulty of the questions. (e.g on level 1 questions, 30 seconds is deducted; on level 2 questions, 20 seconds deducted and on level 3 questions, 10 seconds deducted).
21. The Mathematics question bank should be stored on the school's server, which should then be able to communicate with the client machines.
22. An interface should be available for an admin user to go in an view, add or remove questions from the question bank stored on the server.
23. Access to the question bank alterations should be password protected to prevent students entering the question bank file.
24. Questions should include full use of unicode characters (such as superscripts [ $x^2$ ] and square root signs  $\sqrt{8}$ ) to make the proposed questions and answers clear.
25. When adding or editing a question in the admin section, appropriate buttons should be available to the user for input of special unicode characters that are not present on the standard keyboard. These buttons should append the symbols correctly to the current text box the user cursor is in.

### ● Data source(s) and destination(s)

In regard to the mathematics questions that accompany the game, the questions will either be made by myself or retrieved from an exam board text book. These data will consist of both question and answers and will be stored by means of text files or excel type storage. These will be arranged in categories of topic which can be retrieved straight from the topic if required. Within the topics, questions will be rated in difficulty.

The data will be retrieved in the mathematics mode of the chess game software and will be called upon with every move of the game. a separate form will appear with a randomly selected question based on topic and difficulty and presented to the user. There will be no processing of data, just retrieval when called upon.

### ● Data volumes

With each topic of questions there should be around 90 questions ( around 30 for each level of difficulty) This is based on the statistic that there is an average of around 45 moves per game (calculated from one session of chess club).

There will be 3 main topics, so given a possible extra 2, i expect around  $5 * 90 = 450$  questions to be stored in total.

The fact that some questions will come up more than once is ok as it will embed the answer into the players brain which aids learning.

This is how many questions will be stored but only a certain amount will be called upon randomly.

Judging by the extremities in some games lasting around 65-70 moves, i expect around 70 questions to be called upon during the game.

### ● Realistic appraisal of the feasibility of potential solutions

#### - *The main other solution to this problem would be using Visual Basic as the programming language.*

This solution would involve coding the software in the language visual basic. I would use Microsoft visual studio to implement the solution.

Advantages:

- \* I have programmed in Visual Basic for the past year more so than C# or Java so this means I am theoretically more fluent in this language and will encounter less problems.
- \* This reduces the time taken to implement the solution (by a small amount) as less time will be needed to check up on syntax or code structures for other languages
- \* Visual basic is very good for database work so if I were to implement a small database, it would be effective to use VB here.

Disadvantages:

- \* The chess game will be mostly object orientated to aid efficiently and also to help break down the problem into logical pieces. Visual basic isn't the best regarded language for object orientation due to its less sophisticated nature in comparison to C# or Java. this could lead problems with code structure or confusion due to the shear amount of classes and inheritance.
- \* Visual basic's does not show up some errors that the other languages do, such as uninitialised variables; also you cannot use multiline remarks - which would be helpful for quite a heavily programmed solution.
- \* Other languages are more strict with syntax and language which would be idea for such a project. Strictness in code leads to efficient solutions.

**- *The chess software could be coded using the C# programming language.***

This solution would involve learning the C# programming language and then using this to implement the solution, by creating a piece of software. This choice of solution uses visual studio also.

Advantages:

- \* A much more sophisticated language than visual basic - lends itself to the scale of the project.
- \* Much more strict with syntax and error handling - C# warns you about particular code structures which would not be realised using visual basic.
- \* For object orientation, C# is much more robust.

Disadvantages:

- \* Would have to take some time to learn the language - by translating known programming concepts but learning the variation in syntax.
- \* C# is not necessarily the best language for database work - should i come to need a small database.

**- *Another possible solution is to simply download an open source chess software from the internet and then work on tailoring it to our specific needs.***

This solution would involve selecting an open source software from the internet and adapting its code to meet the needs of the end user(s). The code would need to be able to open in Microsoft visual studio which would be the environment in which the code would be adapted.

Advantages:

- \* This means the main chess engine does not need to be coded, theoretically reducing the time needed to gain a solution.
- \* The user receives the solution is less time, which is optimal

Disadvantages:

- \* Using software from the internet means getting used to the way in which the main engine has been programmed, which takes time and can be tedious if it not been done clearly.
- \* The chances of running into problems are high because I am not aware of how the main engine has be coded, meaning much more debugging will needed.
- \* The tailoring of specific features may be difficult given the current state of the software, whereas given a fresh solution, the main engine can be assigned in a way that will incorporate these features easily.
- \* Online software contains junk and advertisements.
- \* Might not be available to public

**- *The chess software could be HTML based.***

This solution would involve coding the software in HTML using a text editor as the native work environment.

Advantages:

- \* The software would be native to web browsers, which are available on the computers.

Disadvantages:

- \* The game would be restricted to an internet browser which limits the usability of the software as there may be a special case where an internet browser is unavailable for the html to run.
- \* potentially longer to code given the structure of html and limited knowledge of routines etc
- \* My level of knowledge of html is not significant enough for this project which means a lot of time would be spent learning how to code in this manner, whereas in another high level language like C# or java, there are similarities in syntax and structure which I can relate to visual basic.

**- *The chess software could be coded using assembly language.***

This solution would involve coding the software in assembly language. Core routines would be coded from scratch and chess rules would be created via linking together many custom made instructions. Hardware communication would need to be catered for and the work environment would be any IDE that supports assembly language.

Advantages:

- \* Would be a lot faster as less language translation would be needed between the program and the machine.

Disadvantages:

- \* Would take a lot longer to code a solution as standard routines will need to be coded from scratch
- \* I do not feel, with the time given, confident enough to be fluent in assembly language to the extent of being able to write a chess game.
- \* would be difficult as the problem is a non-computing field and so the level of abstraction would be high.

## ● Justification of chosen solution

I have chosen this solution as the benefits of having a programmed chess software outweighs other solutions. Features can be implemented to maximise the efficiency and fun of playing the game. also this solution requires no money, it is free. Having software means the game can be distributed to as many computers as the end user wishes and they are not limited to numbers of physical boards. This solution can be tailored to the end user instead of 'pot luck' with online solutions.

The programming language of choice is C# and this was chosen because it is a very elegant language for object orientation and classes which is how chess game will be coded. C# is better suited to object orientation than visual basic - which is better for database projects. For the solution to be as good as it can be, I wish to use a language which is better suited and better equipped for the style of programming am going to use, hence the choice of this solution.

Also, C# catches more errors than visual basic (such as uninitialised variables, dead code etc) which will be extremely useful if the solution is to be as efficient as possible. Not only this but the comments in C# are slightly better in the fact that you can make use of multiline annotations, which will be useful and important when coding the solution to annotate what everything does.

Not only is C# a better suited language - but i also wanted to learn a different language and apply my programming knowledge to a different environment - which would in turn improve my skills as well as creating a robust system for the end user(s).

A html solution was interesting but I didn't want to encounter lots of problems due to the differences in structure and code. Also I wanted the software solution to be as flexible as possible and at least with the chosen solution, the software can be mounted onto any machine regardless of other software on that computer.

The object orientated design to the solution also appealed to me. the logical breaking down of the project seemed the most effective pathway to take and although it's a demanding one, I know it is a very efficient way to model the software.

## ● Feasibility Study

### Technical feasibility

The technology available is sufficient to implement this system. Using a variety of software to create the system, its technical feasibility is high. The system does not require any specialist equipment or extra devices and will just be created to run on client computer machines.

### Economic feasibility

If the chess club were to invest in an online solution to their problems and have the software do what they want without any advertisements then the school would have to invest and buy a piece of software and its licences to use it. This requires money, therefore, which is not optimum at all as the school budget for other things. So my system is of great economic feasibility as it is free for the

chess club and requires no cost at all. Also in terms of installation, there will only be a couple of storage devices with the installation files on ( 1 or 2 CD's perhaps) as only 1 Cd is needed for installation purposes no matter how many client machines there are, so this is minimal cost in terms of storage. Also the proposed system is planned to be small in memory size so will not take up a significant amount of hdd disk storage on the client machines which is important for other things.

## Legal feasibility

The proposed system does not involve any sensitive data of any type and is not breaching any copyright laws as it is completely tailor made.

## Operational feasibility

The chess club runs after school in a dedicated room so having the system on the computers will not affect classes currently teaching or any conflict of that manor. However, having the system on the computers may lead to students opening it when doing work during ICT/Computer science lessons when using the computers. This may have an impact on lessons. However, this can be overcome by implementing small rules - possibly only having the availability of the software on the students user accounts - only those who attend chess club ( can be registered when joining etc) so that none else runs into this problem and then simply telling the chess club members that they should not use the system outside of chess club hours.

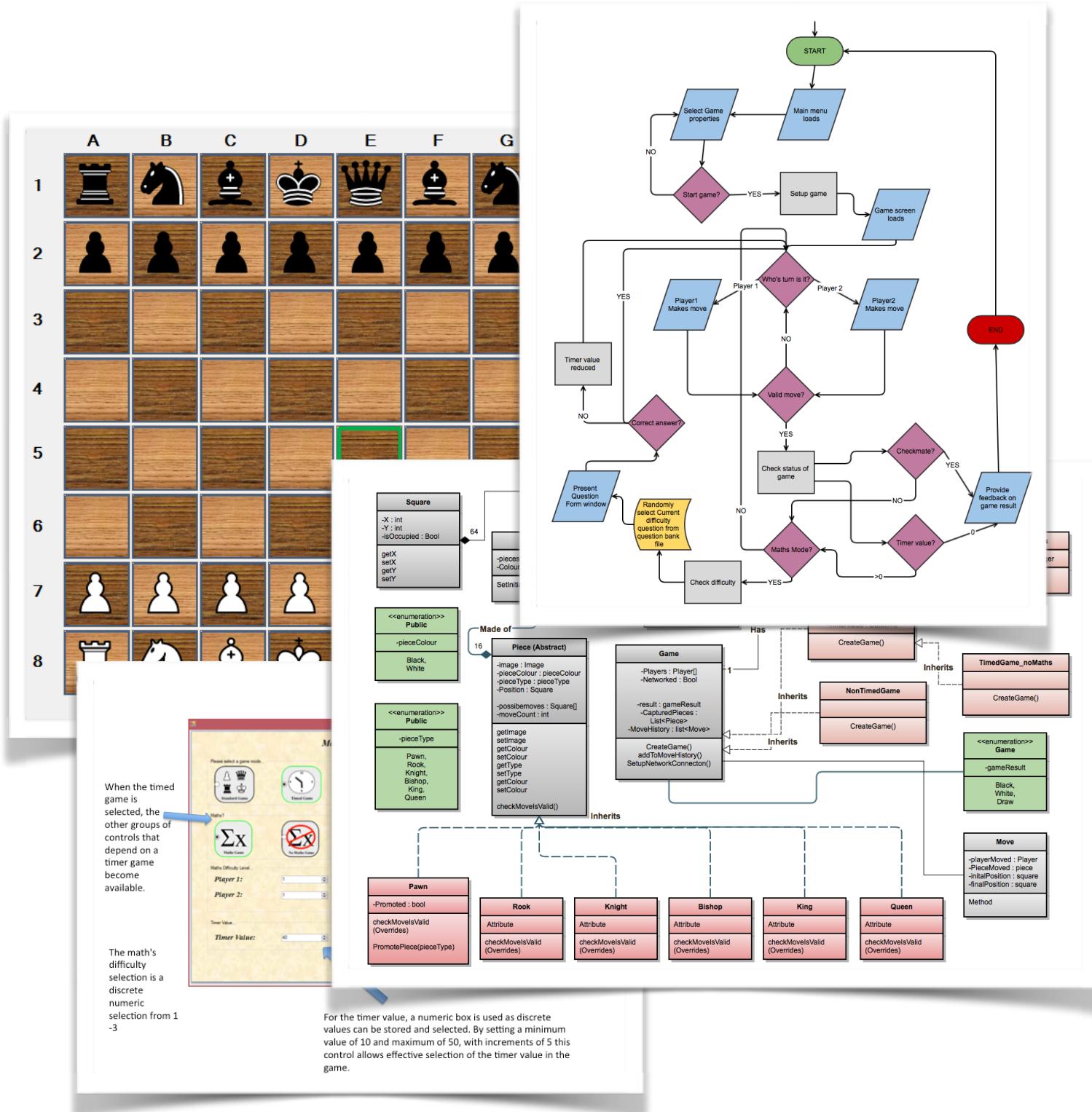
In terms of the room the chess club is held in, there are computer machines around the edges of the room (approx. 20 machines) so there is plenty of machines for the system to be installed on. This then makes space in the Center of the room on the tables for physical boards - should more than 20 people attend one week.

## Schedule feasibility

The system will take a significant time to develop given the circumstances of the project. The user requirements make the system in depth and to meet all of these requirements and make a well developed system will take time. The longest part of the system will be the chess engine and creating a code engine that can play a game of chess. This will also be the largest. So the majority of the time will be dedicated to this part of the software but then subsequently, user forms and the mathematics section will also take a significant time to perfect.

As discussed, given the user requirements, a solution can be developed in as suitable time period but certain features looked into (networking for instance) will be too involved on top of the current proposed system in the given time.

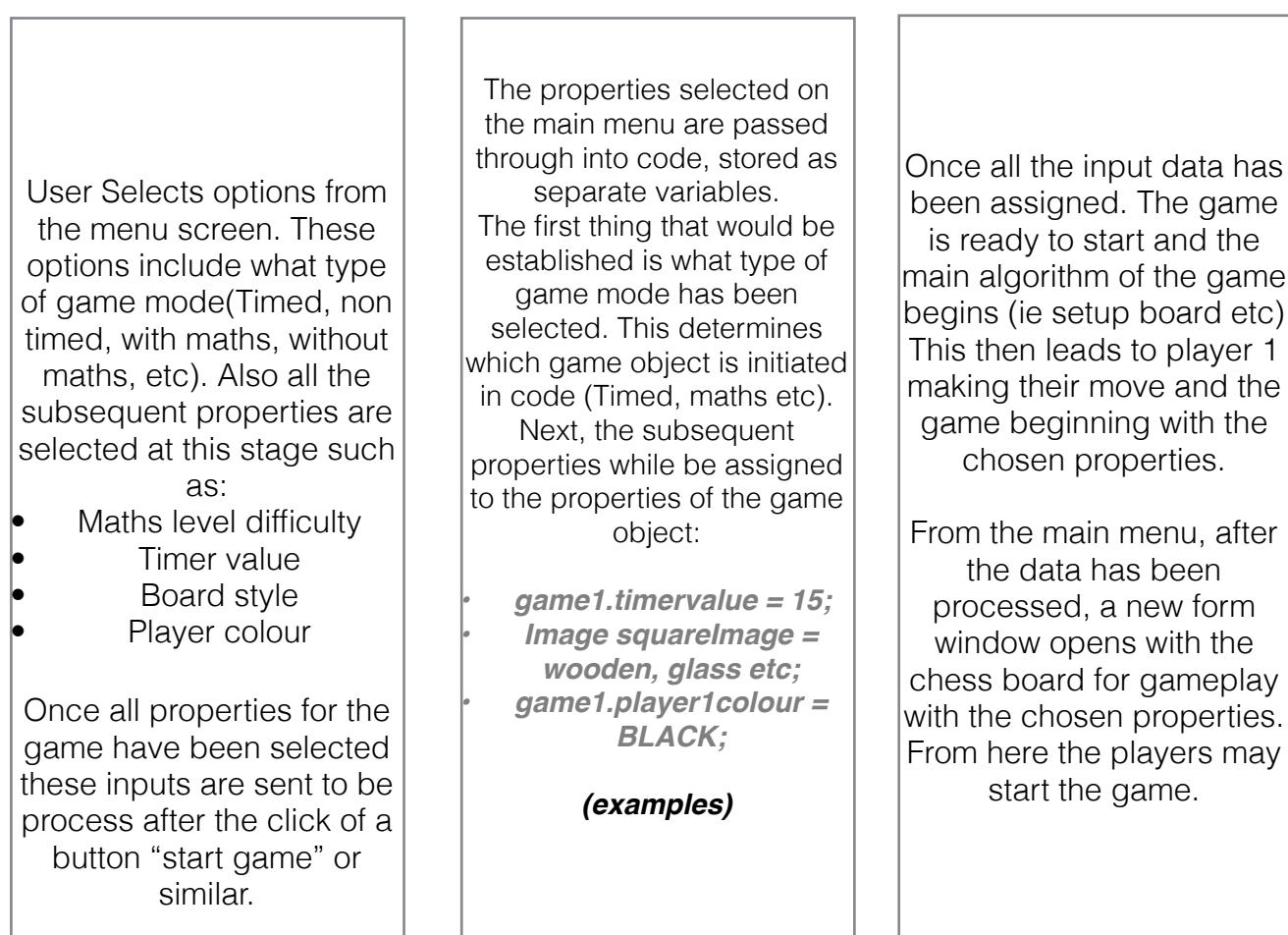
# Comp4 - Design



## Comp4 - Design

### ● Overall System Design

#### Input, process, storage, output - chart (for main menu)



**Input, process, storage, output - chart (gameplay)**

<b>Input</b>	<b>Process</b>	<b>Output</b>
<p>To start the game a start button will need to be clicked.</p> <p>After this initiation, player 1 will need to click a piece and the click a square where the player wishes to move the piece.</p> <p>If it is player 2's turn the he/she will select a piece and move.</p> <p>Some things are able to be changed in game, such as the highlighting of squares. So a possible input will be toggling this option on or off.</p>	<p>As each player begins their move, their individual timer value decreases.</p> <p>As the first click happens, a check is made to see whether the square contains that players colour piece. Also a list of possible moves is generated for that particular piece (Highlighted in green if this option is toggled on in gameplay)</p> <p>As the second click happens a check is made to see whether the square is one of the possible moves - in which case the piece is moved.</p> <p>At the same time at this, lots of algorithms are executed to do things such as check the state of the game, increase move count etc.</p> <p>If the highlighting of moves is toggled then the boolean variable determining this is updated.</p>	<p>Once the input has been processed, the game continues with the next move (i.e player 2 now makes a move). before this however, the board representation is updated to show the new positions etc.</p> <p>If the highlighting of moves option is turned off, then when subsequent moves are made, the squares will not be highlighted.</p> <p>Sounds may also be played as moves are made. Also for actions such as capturing pieces, being put in check etc. All of which create a better interaction with the user.</p>

**Storage**

In game play the main storage involved is the question bank.

If mathematics mode is on, after the player makes the move a question will be presented to the user.

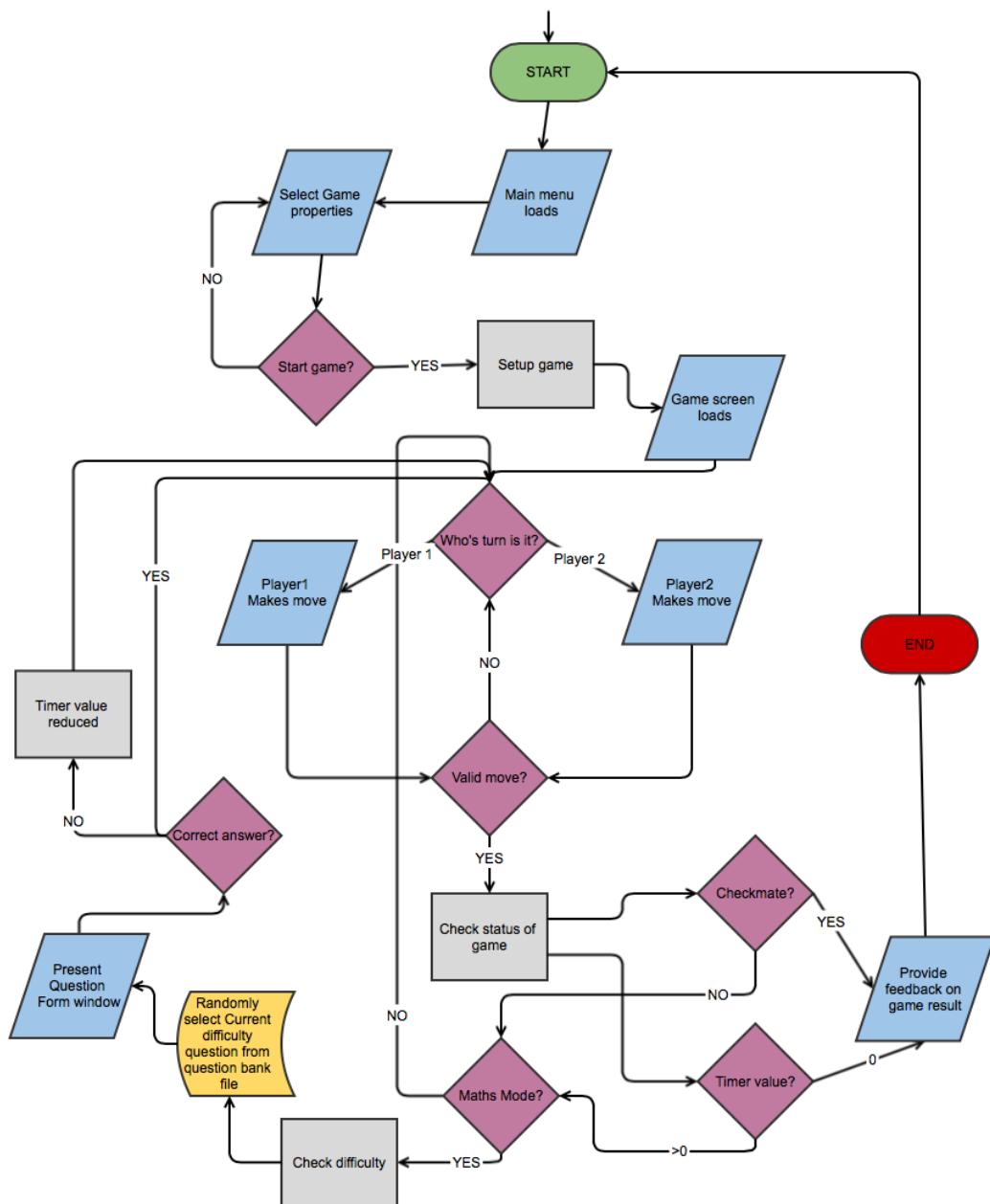
The storage of these maths questions will be in a file on a server. The file will be pushed out to the client machines when the program runs (see file organisation for more detail). The contents of the file will be then subsequently stored in memory however, in secure objects as part of the system.

The question will be asked and the user will be prompted for an answer which they will have to input. This input will be processed by comparing against the answer, also stored in the same place.

Depending on whether they get the answer right or wrong, appropriate feedback will be provided as output letting the user know whether their time will be deducted.

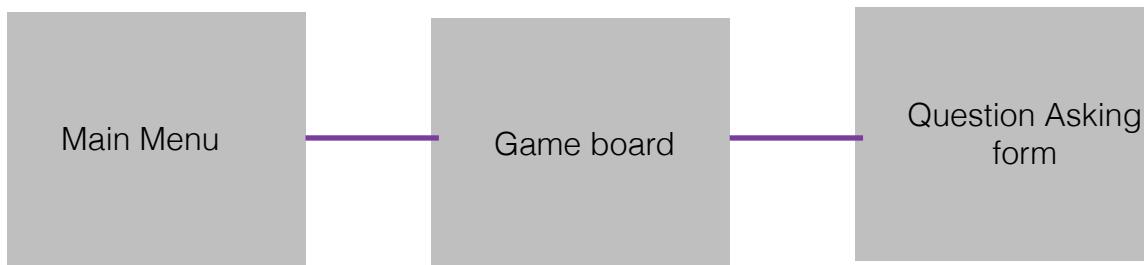
## System Flowchart

Here is a generic system flowchart of how the system will run. Some processes (such as the “setup game” ) or validations (such as “valid move?”) contain much more depth and multiple algorithms but are shown by a single icon on the flow chart. (See *identification of processes and suitable algorithms for data transformation*)



## ● Description of modular structure of system

### **Form/Navigation based structure:**



#### **Main Menu:**

The main menu component is a form that the user is first presented with after loading up the program. This form allows the user to select options for gameplay by a graphical interface containing decision controls such as radio buttons.

The properties that the user can select in this module are properties that determine/adjust gameplay.

#### **These properties include:**

- Which game mode (Timed, Non-Timed, Maths, Regular etc).
- For a timed mode, the value of each players timer ( i.e 15 mins, 20 mins etc).
- For the mathematics mode, the level of difficulty of the questions asked for each player.
- The style of the game board (i.e wooden, plain etc).

Also there may be an option which takes the user to another form in which an admin user can add or remove math's questions, which are stored on the server.

#### **Game Board:**

The game board component is the form that is presented after the main menu is closed and all the properties of the game have been chosen and processed. The form will have the chess board itself, the timer value for each player, current score of maths questions (for mathematics mode) and options such as toggling highlighting of moves.

The style of this form will be dependant of the properties selected in the previous module for board style. The properties of which will be stored as variables in code in this module. (i.e back colour, chess board square images, chess piece images).

This form will be the main form in use by the users. This is the module that will contain the board and the player will move pieces on. This module will be the most dense in terms of background code and algorithms.

#### **Question Asking form:**

This form is a dynamic form, that pops up throughout the game. This module is called upon from the game board module every time a question needs to be presented to the player in the mathematics mode. The module will determine the difficulty of question needed and will present the user with the question will has been selected <sup>28</sup> from the source of questions (anticipated to be a text file or equivalent).

**Code based structure:**

Code structure can be broken down into modules, hence the object orientation approach. See the inheritance and class definition actions for a more detailed overview of each module.

The main ‘components’ of the coded structure are:

**Board:**

The board will be a class which deals with the state of the board. This includes an array of squares on the board and details of whether those squares are occupied or unoccupied. The main properties, therefore, will be a square array (of object square - see inheritance diagrams), array of piece sets (two piece sets will be on the board, black and white).

The main method will be the `setUpBoard` method which uses the list of pieces and the board positions to initiate the board for the game, from here on the pieces will mainly do the workload due to my piece-centric approach.

**Game:**

The game module handles the state of the game. It manages the two players in the game by having a turn property which allows for comparisons to be made to determine who’s turn it is. These players also have scores for maths questions or current state (such as being in check) which is all managed and stored by the game module.

The game module also handles all the game properties described in the form based module description. These properties are assigned to properties in the game class and is used in the code to manipulate gameplay. for example the timer value is assigned to `game1.timervalue`.

**Piece:**

Possibly the ‘heaviest’ module out of the system which represents a piece object and manages the properties of that individual piece.

This module also handles all the validation for the moving of each piece, procedures will be overridden down the inheritance structure, to cater for pieces individual validation.

Each piece will have a position, colour and type. But also to aid the design of the coded structure (as the approach is piece centric) are properties such as number of moves made, (as a pawn, for example, can move two squares on its first move).

The biggest method of piece will be the check validation of moves and also the get possible moves method. These methods require quite heavy processing in comparison to other methods in the system.

**BoardAdmin:**

This module will play a handy role in the function of the system and will be a class that contains static variables which are important to all other modules in the system. This class acts as the administrator of the other modules and allows efficient passing of values between modules. This module will be used to hold temporary values when evaluating actions between two other modules (forms) in the system or it will be used to store important, global variables which are referenced throughout the system to make the coding more logical.

These main components will be the highest on the structure. Other components will link off from these as shown in the analysis object diagram and also the revised diagram in the design.

A module for each type of piece (King etc) will inherit the piece class, but will override validation sub routines as there are different rules for each piece.

Multiple other sub modules will branch off from these three based on the aggregation or inheritance between them. *See the revised object design for more detail.*

### **Form Code:**

The code behind each of the three main form modules, will aid the visual user interface by interacting with the code engine. The main menu form will save the selections as variables which will be passed to the game board form. The game board form will deal with updating the view of the chess board, probably by means of a timer which will update every second to provide a more efficient user interface. Also the whole process of players making their moves will be handled in the code behind this form as it involved click events which is part of the user interface. Players individual timers will be handled in this form code by determining whether it is that players turn and counting down in the background. Finally any in-game toggles will be handled in this module also, as this will affect the output of the user interface.

In the question asking form, the form will interact with the external data storage (the questions). The form will call upon the appropriate question based on difficulty and will be random. The form will then update with the question and provide user input for the answer, which will be compared - in code - with suitable validation

Promote Pawn form

*Another module in the system is a minor form in the system but a separate form nonetheless. This form is a popup that allows the user to choose a piece of which to promote the pawn piece to. This form will include click events on pictures of all the available pieces to promote to and will then pass that information back to the game engine so processing can be done and the piece type property and image can be updated.*

**Mathematics System:**

The mathematics system in the program is predicated upon a quick fire quiz style approach which reinforces students knowledge and aids revision for basic topics in maths.

In terms of how this is implemented, there will be a 'bank' of questions which each have an answer (or two if appropriate) and a difficulty rating(1-3 or easy-intermediate-hard etc). The procedure will be, when a player makes a move or when the player starts the turn, a question will be presented to the user which has been randomly selected from the bank of questions. This question will have a difficulty rating equal to that which the player chose for themselves at the beginning of the game. If the question is answered incorrectly, appropriate feedback is given and the score of the maths questions in the game is updated. Time is deducted for a wrong answer which is granulated depending on the difficulty level of the questions (for example, 5 seconds should be deducted for hard questions, 10 seconds for intermediate and 15 seconds for easy questions).

Using this difficulty separated scheme to the questions allow the students to become comfortable and also gain a sense of improving when they progress to the more difficult questions and also when they continually get questions right. Even if a particular student finds the hard questions easy, they will try to get all questions correct throughout the game - still having a goal and still reinforcing knowledge.

The actual bank of questions however will not be stored on the client machines, they will be stored on a server and the contents of the file will be retrieved at runtime (see file organisation for details why this is the design choice).

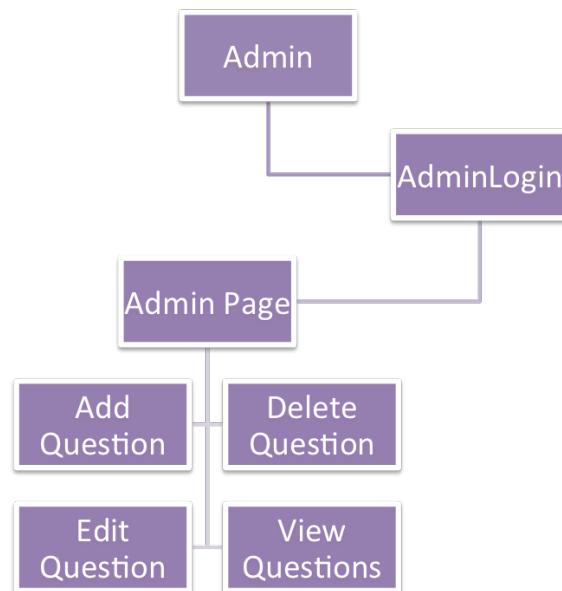
To meet requirements for the system and add flexibility to the system there will be another main module to the system which is available from the main menu which is an "admin" section. This will be password protected so student cannot access this area and will contain a mini system in which the admin user can update, add and delete questions from the question bank. This makes the system dynamic and means that content can be added in the future.

The questions will involve some symbols such as the square root sign and superscripts for powers and exponent questions - These need to be supported which is an objective of the system as having these add to the clarity of the module, which is a key thing in this program. To do this, unicode characters will be used in code and the user should be presented with a little window of buttons which add these symbols to the current text box. This reduces validation needed for input as you are giving the user the standard input symbols needed so this reduces potential errors that may occur.

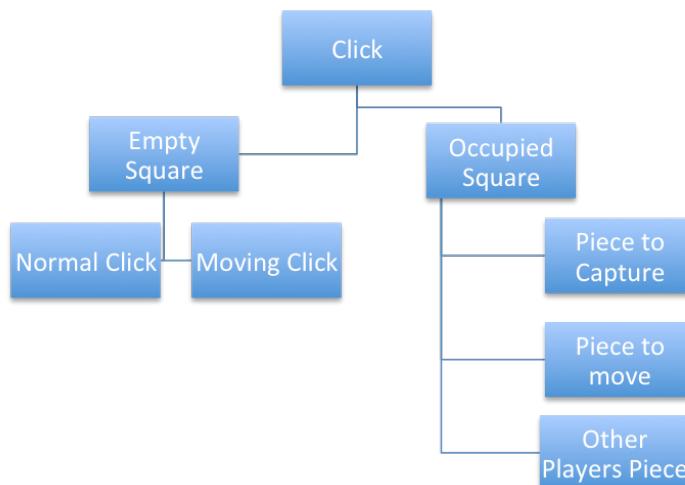
The idea here is that the contents of each line in the file will be the details of each question and in the main system, there will be question objects that have properties to store these details. Then at runtime, these question objects are populated from the file, which is retrieved from the server. This means that the contents of the file can change and whatever change, when the user runs the program, the latest file will be read from and will be available immediately - instead of static assigning of questions in code.

**Board Admin Modular Structure:**

This diagram shows the modular structure of the admin section of the system. The section consists of a login screen which is vital for the security of this area. Then from the admin login page is the main admin page where the admin user can select to do various things involving the question bank file. The admin user may ADD a question, DELETE a question, EDIT a question or simply VIEW the questions.

**Click Event Structure:**

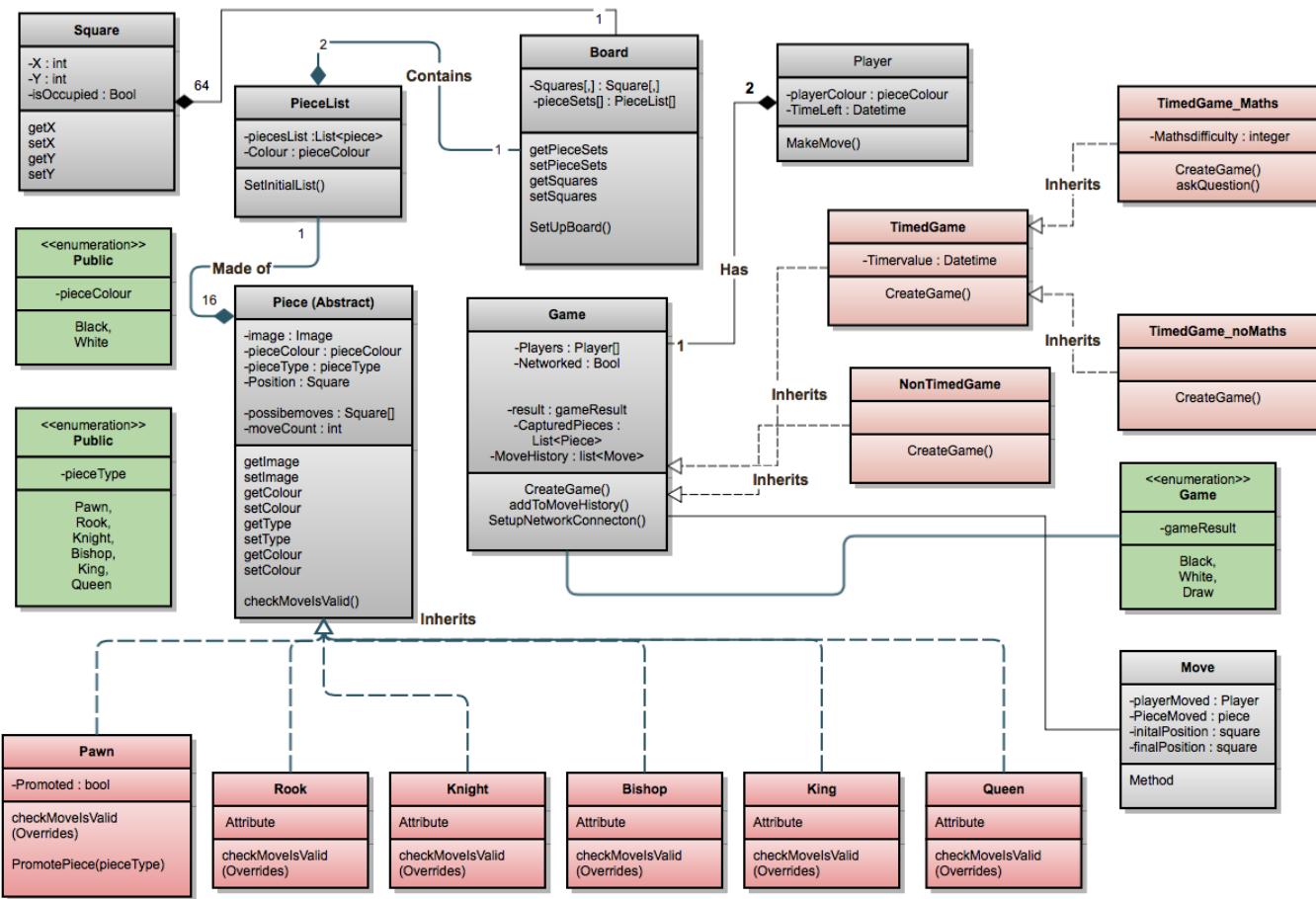
This diagram shows a brief description of the processes that could be happening under a click event. A player making a move is a two click process and depending on the occupant of that clicked square or the current possible moves, a different procedure will be run. For example, a player could have already clicked a piece to move, so this second click would be the square they wish to move to.



## ● Revised Class definitions and details of objects

From the analysis, changes have been made to the object structure after revising the system in more detail. Here is the revised object definition diagrams with inheritance and aggregation:

**The main improvements from the initial analysis design are:**

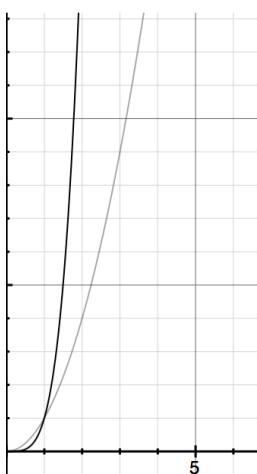


- Using a square class instead of using just an array of coordinates.

When revising the class structure, this seemed like a good move. When using an array of coordinates for the game board, a lot of passing of x and y coordinates will have to be made which would dramatically increase the amount of code and variables being passed through modules.

Giving the square class x and y coordinates, a simple call upon the properties need be made instead of passing the coordinates.

Also, another advantage of using a square class is that i can have an array of square objects which represent the board which is 1) more understandable from my perspective 2) effective with a 'occupied' property which is VITAL in the validation of moving pieces. If this were not the case, and i was only working with coordinates for the board, a scan would have to be made across all pieces on the board to check their position against the squares that lead up to the target square for movement, just to see whether that particular square is occupied.



This would dramatically decrease algorithm efficiency with respect to time as a lot more scans of the board will have to be made which is  $O(n^2)$  complexity ( $8 \times 8$  square board).

As shown left, a graph of  $n^4$  against  $n^2$  showing that the complexity of the algorithm will increase with respect to time, dramatically with the alternative approach - reducing performance. The other approach would also need quite a few repetitions of the algorithm where as the designed approach with square objects means the board only has to be scanned once to check a position on the board - a massive improvement in time complexity.

Also, the possibleMoves procedure will be incredibly more efficient as less checks will have to be made. Previously an array of strings representing the positions would be stored but with the square approach, only a single set of checks is required. Furthermore, the highlighting of squares will be made much easier by using a parallel array between the board squares (object square) and the picture boxes on the form (object picture box).

- Splitting up the game class/module into sub classes which inherit game:*

The game class was currently trying to cater for all possible game modes of the system. So logically it made sense to break this down further. The game class therefore became the generic class and new classes were made which inherited the game class:

- TimedGame
- NonTimedGame

These were also broken down further for the decision of maths mode or no maths mode.

- TimedGame\_Maths
- TimedGame\_NoMaths

The whole idea of the maths mode is that time is deducted for wrong answers to questions so the maths game mode only needed to be inherited from the timed game classes.

Splitting the game module up like this proved to have several benefits in terms of design of the code structure. The properties of gameplay selected at the start of the game on the main menu can be used to determine which game object is created via an implicit assignment. This is much more efficient as before, a timer property would constantly be checked despite the game not being timed at all. which wastes processing time. Instead with a broken down class structure, specific properties like timer value and maths difficulty will only be checked if the game mode involved them.

- *Involving Move lists and move counts:*

After considering some aspects of the game of chess more closely it became apparent move lists and move counts would be extremely useful.

The pawn, for example, can move two squares on its first move but subsequently one square. The way around this rule in code would be to have a move count property on the piece class and then validation can be included to check whether it is the pawns first move or not.

Similarly, the king and the kingside rook can only perform the castling procedure if both the king and rook have not moved yet. So again, a move count property will be useful here to check whether either piece has moved yet.

To have a list of “moves” it therefore follows there should be a Move class, so one was added. Every time a move is made, a move object can be added to a move list of the game with the important details being:

- Player moved
- Piece moved
- Initial Position
- Final Position

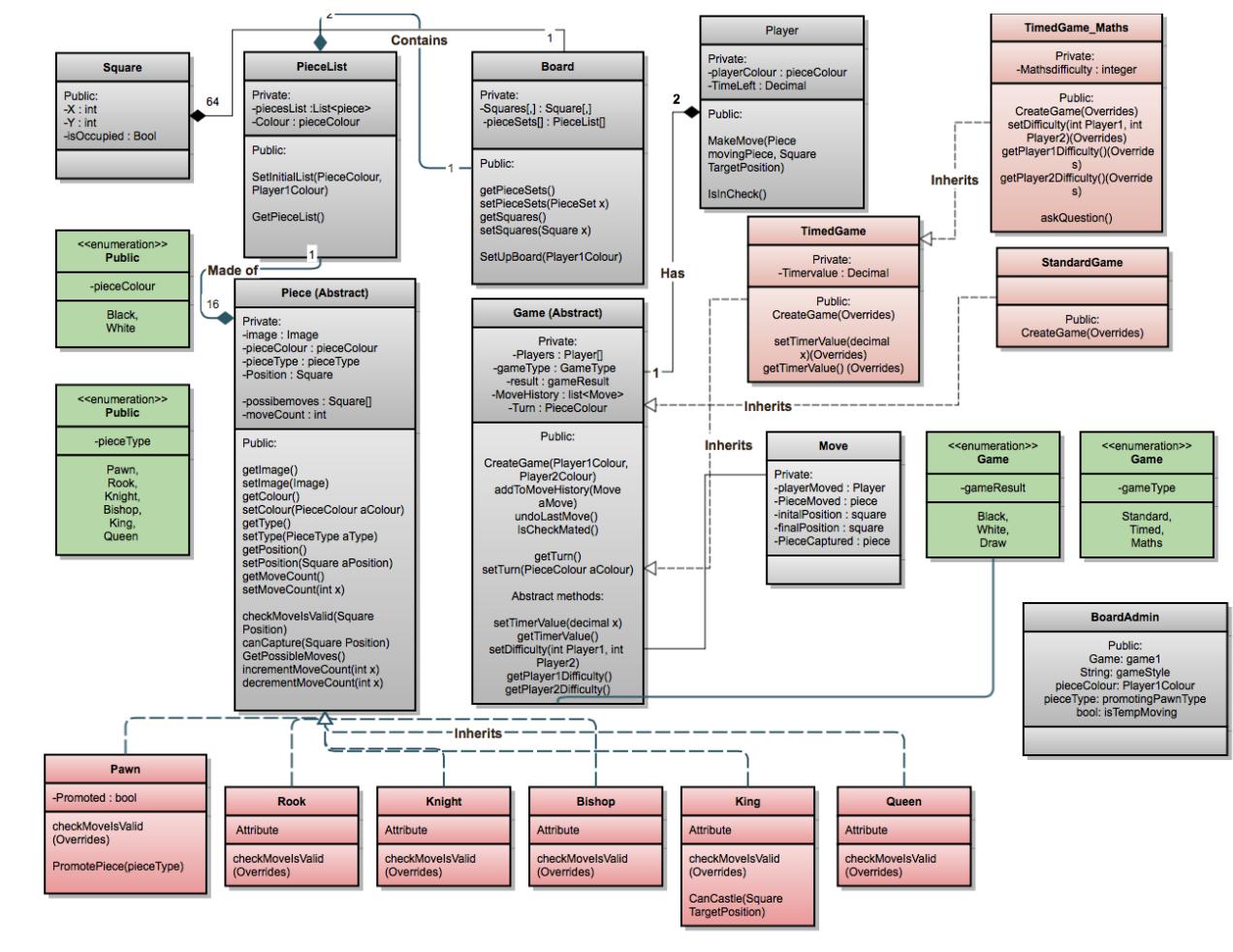
This is a logical and simplistic way to record moves that have been made which will aid the coding of the rest of the system.

- **Revised Class definitions and details of objects (Extended Design)**

The revised model was an important stage in the design process but the model could be refined further by looking into detailed aspects of the objects and how the link and work with each other.

After further design and thinking in more depth about what problems may arise and what potential obstacles may occur, a better – more detailed object design could be made.

In some cases, it was evident that some classes designed initially were redundant which is unnecessary and in other area it seemed logical to have an extra class to manage and aid processing. In general, many methods were thought of and which classes these methods would come under in theory.

**The extended object analysis diagram:****The main improvements from the secondary design are:**

- Creating a BoardAdmin class

When revising how the main modules (forms) in the system would communicate with each other and how important variables would be passed, efficiently, between the forms it became evident that this would be a problem due to the nature of the windows forms modules but also the fact that a lot of passing variables would be involved which I incredibly tedious.

Using a board ‘administrator’ class, this can be a gateway between not only the main menu and the gameBoard form but between all modules in the system and essentially act as a multi module administrator class.

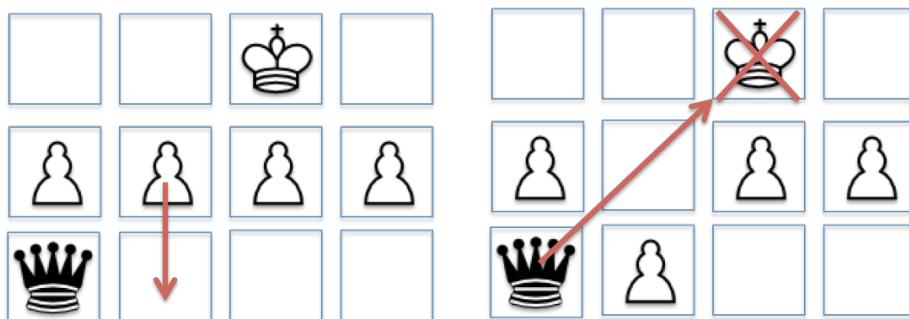
This class will simply store **static** variable which can then simply be referenced and assigned to in the other modules of the system, making the more global variables much more efficient to access while removing any potential problems in the technical solution.

Some of the planned variables for this class to store are those passed over from the main menu: **Game Type**, **Game Style**, **Timer value** etc

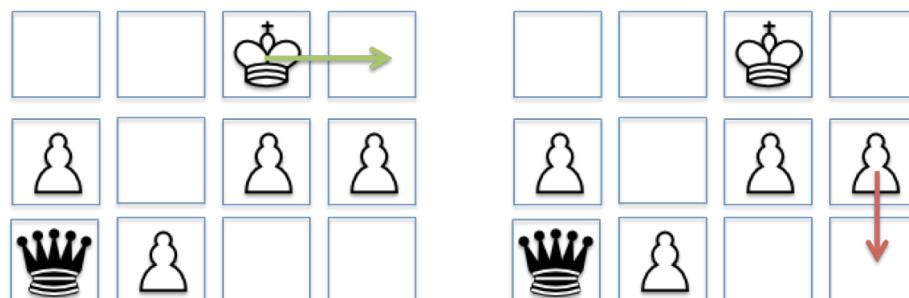
Also some Boolean values relating to how the main chess system will operate (see further down – **Temporary moving**)

- TEMPORARY MOVE APPROACH

This is possibly the biggest change in the design process for the chess system in general and how it will behave. One of the hardest situations to analyze was how the game would cater for a player being in check **as a result of other pieces movement**. By this, I mean – in a game of chess, a piece cannot be moved if the result of that move leads to your king being in check (see diagram below):



Example of an **INVALID** move, as the White's king is then exposed to an attack from the queen – hence, putting yourself in check.



Example of an **VALID** move, as the White's king is moving OUT of check.

Example of an **INVALID** move, as the White's king REMAINS in check.

As shown, it is not only the king's movement that needs to be validated when it comes to check situations; there are in fact many other scenarios where movement of **other pieces** results in check. Also, as shown in the second diagram, when the player is in check, they can't simply move any piece, they must make a move to get out of check.

Although not totally sure how I would overcome a problem like this, after thinking logically the best thing that seemed possible was pieces making “Temporary Moves” after which, an algorithm can run to determine whether the player's king is in check, if TRUE then this move is invalid. Of course, this temporary move will be retracted and will not be seen at all by the user, it will happen as part of the chess engine.

Otherwise, this would be extremely more difficult to implement, a scan of every enemy piece would need to be made first to determine the piece that has put the king in check. Then, a scan of every friendly piece would have to be made – within which, each pieces moves would have to be evaluated mathematically, by comparing x and y coordinates to determine whether the move will ‘block’ the enemy piece – which of course then requires another scan of that pieces possible moves.

The efficiency in terms of time complexity would be decreased dramatically with this approach whereas the idea of a temporary move and then the execution of a “is in check?” algorithm seems the most logical, the most efficient and would reduce the amount of board scans needed.

This temporary moving procedure will be a standard move made by the player class but instead this move is **retracted**. This compliments the initial design of having a move List, which records all moves made. A further method can be added now, which undoes the last move using the properties of the move class, this extends the functionality of the game to incorporate an undo button, which allows moves that have actually been made, to be retracted just like in the temporary move procedure.

This would be extremely powerful as no matter what move is made, the algorithm will determine the state of the board and also this would dramatically reduce possible mathematical/arithmetic errors.

- *Removing redundant, inherited game classes*

Looking more closely at the selection of game mode in the system, it appeared to be pointless having a subclass “**TimedGame**” and then an inherited class

“**TimedGame\_NoMaths**” which behave in exactly the same manor and do not deviate at all, so the creation of this class is a waste. Instead, in the technical solution, if the user should wish to play a timed game with no mathematics involved, a game of type “**TimedGame**” should be declared.

- *Adding further methods to existing classes*

At this stage in the design a better idea of the game is obtained and with this, further modules can be added to the object diagram based on designed procedures.

- Using an extra property in the move class of “**capturedPiece**” allows any pieces that were captured to also be stored in the games history of moves and with this, if a move is to be retracted or evaluated, a scan of the “captured pieces” list does not have to be made – which was the design initially. This reduces the amount of code.
- A method for a king piece would be needed to evaluate whether it can perform the castling procedure, as mentioned before each piece will have a “**movecount**” property which will be used in this procedure as the rook and king cannot have moved prior to this move. This procedure can return true if the move can be made.
- Game state methods such as checking for check and checkmate as well as standard methods for adding and removing from lists etc (see class descriptions for further details)

- Public and private modifiers and also realisation of abstract classes which need to be inherited down the class structure.

### Description of Each Class:

Board
<b>Private:</b> -Squares[,] : Square[,] -pieceSets[] : PieceList[]
<b>Public:</b> getPieceSets() setPieceSets(PieceSet x) getSquares() setSquares(Square x) SetUpBoard(Player1Colour)

The **board** class manages the state and squares of the board. One property is a 2-dimensional array of Square objects which represent the game board. As each square has X and y coordinate properties, this makes the game board a simple reference to squares on the board. This array of squares will be passed around to reference the board and to manipulate its contents.

A second property is a reference to the pieces in play as the pieces “sit” on the board and need reference to it. The **pieceSets** property is an array of 2 elements, a black set of pieces and a white set. This is of type pieceList (see pieceList class). Having the piece sets in the board class aids the setting up of board and pieces.

The main methods are for reference to the board properties: **getPieceSets** and **setPieceSets** allow for reference of the **pieceSets** property of the board.

The **setUpBoard** method initiates the board for gameplay by generating the list of pieces from the pieceList class and assigning the positions of each piece with reference to the squares array. Also setting the x and y coordinates of each squad on the board so it is ready for further processing and also setting the occupied squares as occupied. The parameter of **Player1Colour** is used and is vital to determine which colour pieces are set up at the bottom of the board as the system should have player 1’s pieces moving upwards.

Square
<b>Public:</b> -X : int -Y : int -isOccupied : Bool

The **square** class is a replacement for the previous array of string coordinates. It is what makes up the board array, each square object being a square on the board. Each square object will be a reference to a position on the game board. To do this, the square class has an **X and Y** property. Also for the aid of checking valid moves and also for other things in the coded solution, an ‘**isOccupied**’ property is assigned to a boolean value to determine whether the square is empty, or occupied by a piece. This helps when determining the possible moves of a particular piece or comparing the position of a piece with a position on the board, through a consistent use of square objects all having a place on the board class as a board squares array property.

There is no need for methods with this class and its properties are public, this is to make them easily accessible and to remove any unnecessary get and set methods.

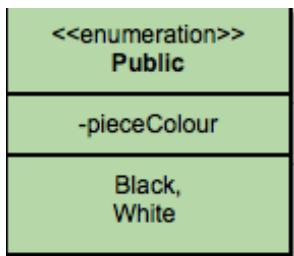


The **pieceList** class does a job of grouping together piece objects into logical lists for black and white pieces. This is beneficial as scanning pieces is reduced down to half the number by their colour which is much more efficient. Also, it eases management of pieces as they are all in one logical place and not individually managed.

The properties, therefore, are a **piecesList** property which is a list type of piece objects. Also a **colour** property to determine the colour of the piece list.

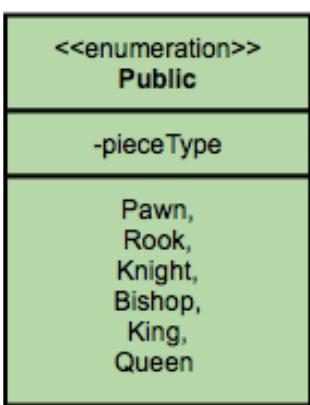
The method **SetInitialList** is what populates the list of pieces, within this procedure is a large set of comparisons and assignments of pieces to a list. The position, colour and type of the pieces are all set in this method before the pieces are added to the list.

The parameters for this method are Piececolour and Player1Colour, this is so that the piece set populates all the same colour pieces in a list and also sets their initial position to the bottom of the board. This method will be called in the board class when the board is setup.



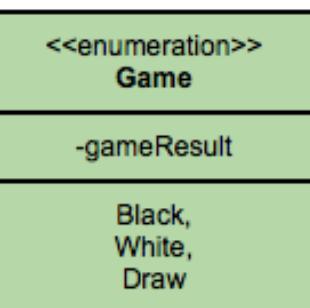
A lot of classes depend on colour comparisons and so it makes sense to have this as a public enumeration with elements **BLACK** and **WHITE**.

This can then be consistent across the whole system and avoids the use of string comparisons or number comparisons, instead just a simple use of this enumeration type.



As each piece has a type, it makes sense to have a type property, but instead of having string references it is much more robust and less error prone to have a public enumeration for **pieceType**.

Although the pieces are split into subclasses (king, pawn etc) it is still useful to have this enum for the piece property to make the comparisons a lot easier.



The **gameResult** enumeration provides a clear determination of the result of the game and can then be used to provide feedback to the user on the game from robust boolean options.

The **Piece** class is by far the biggest class of the system. The pieces have a lot of code behind them and therefore key properties are needed.

The property **image** is what stores the image of the piece, this is assigned in the pieceList class method and is determined by what the user selected in the main menu. This follows the design of using picture boxes on the game board form to represent the squares on the board. The images of which will represent the pieces.

The other properties are very important for checking purposes: **colour**, **type and position**. These properties are also assigned in the pieceList class. The colour property will be the public enumeration type to keep the system consistent. The type will also be a public enumeration to ease the assignment and comparison of piece types throughout the system. Finally the position property is of type square, this square will be a square in the board squares array from the board class.

Piece (Abstract)	
Private:	<ul style="list-style-type: none"> <li>-image : Image</li> <li>-pieceColour : pieceColour</li> <li>-pieceType : pieceType</li> <li>-Position : Square</li> </ul>
Public:	<ul style="list-style-type: none"> <li>-possiblemoves : Square[]</li> <li>-moveCount : int</li> </ul> <ul style="list-style-type: none"> <li>getImage()</li> <li>setImage(Image)</li> <li>getColour()</li> <li>setColour(PieceColour aColour)</li> <li>getType()</li> <li>setType(PieceType aType)</li> <li>getPosition()</li> <li>setPosition(Square aPosition)</li> <li>getMoveCount()</li> <li>setMoveCount(int x)</li> </ul> <ul style="list-style-type: none"> <li>checkMovesValid(Square Position)</li> <li>canCapture(Square Position)</li> <li>GetPossibleMoves()</li> <li>incrementMoveCount(int x)</li> <li>decrementMoveCount(int x)</li> </ul>

The next two properties are for slightly more complex things. The **Possible moves** property is an array of positions (square objects) which represent the possible moves for a particular piece. This will be used in conjunction with the check valid move method to see whether the target move is one of the available moves but also to populate an array of possible moves so that the visual representation can then access these squares easily, to highlight these squares. This again, avoids scanning the whole board more than once, just to highlight the squares.

The **Move count** property keeps a track of how many times the piece has moved and this increments every time a move is performed. This is used (as discussed earlier) to check rules for certain pieces such as the pawn and king.

There are standard get and set methods to access the main properties of a piece. Then quite a large method is the **checkmoveisValid** procedure. This method first checks that the target position is on the board. It then will check the rules against movement for that particular piece (hence this is an overridable routine). This involves checking squares to see if they are occupied and also whether the target position is present in the possible moves array of squares. (*see identification of process and suitable algorithms section for more detail on this algorithm*).

There are also standard get and set methods for the private properties of the class which both retrieve and manipulate them.

The **canCapture** method is a procedure that takes a parameter of type square, which represents the target position being evaluated. This method checks the target position to see whether this square is a valid move (the checkValidMove procedure will be called here) then it checks whether the square is occupied by an enemy piece (by scanning their piece list and checking whether a piece has the target square as its current position) if this is the case then the moving piece may capture this piece.

This method will be useful for evaluating capturing of moves in conjunction with the game.

Finally there are two methods **IncrementMoveCount** and **DecrementMoveCount** which do exactly as they say. They are used to alter the value of the **MoveCount** property based on whether a move has been made or perhaps the move has been retracted (if the move was temporary).

The **Game** class is one of the main modules in the system. The game class doesn't deal with pieces directly, it deals with the state of the game and the players but also has reference to the board, which in turn has reference to the pieces. This is the type of object that will be created in the user interface section. Boards and pieces will not be created explicitly, they will be created internally to the game class which is the suitable structure, the interface is about the game. This is an abstract class, as this is now broken down into further subclasses of game.

The property **Players** is an array of two elements, player1 and player2. These are of object player. This property reference the two player objects which are in the game. This will then lead to a loop of turn taking for both players.

The **result** property is of public enum type and stores the result of the game and will be used to provide feedback after the game has ended.

As well as the result of the game, as the game class is abstract a gameType property is needed to dedifferentiate between the different type of game modes in the system. This will make processing of procedures easier in the technical solution as there is a set property with the type of game in, **GameType**.

**MoveHistory** - A list of moves that have been played should be recorded for useful purpose as explained previously. Move such as check and castling are predicated upon a move being made immediately after another and so a move history would be useful here. These are of type move which is a class on its own, storing the important details of each move. This will be added each time a move is made.

The game class also keeps a log of who's turn it is and this will be done by storing the colour of the player who has the current turn of the game. This property **Turn**, can then be used all over the system for comparisons and validation depending on the turn of the game.

The method **CreateGame** creates the initial state of the game by creating a board object which in turn creates and sets all the pieces on the board etc. The method will use variables from the main menu such as each players colour to set the colour property on the player objects which can then be used to start the game depending on whoever is white.

The **addToMoveHistory** procedure adds the latest move made to the moveHistory list property by creating a move object and filling it with the details of the move.

As explained previously moves will be made temporarily to check the resulting state of the game, these moves however will have to be retracted, like nothing ever happened and to do this we need a procedure **undoLastMove**. This method takes the "top" of the **moveHistory** list and uses the **Move** object properties to reverse the move made.

An expected complex method, **isCheckMated** is one that will be used in conjunction with the **IsInCheck** method in the player class. The principle is, this method will take all the friendly pieces and scan through their possible moves (In which there is various calling of the **isInCheck** method) and keeps track of the null moves. If there are no possible moves (as the king is in check mate and no move will change this state) then this method will return true.

The rest of the methods are **abstract** methods which need to be present as they will be inherited by subclasses of game. These are methods associated with timed game and maths game.

Game (Abstract)	
Private:	
-Players : Player[]	
-gameType : GameType	
-result : gameResult	
-MoveHistory : list<Move>	
-Turn : PieceColour	
Public:	
CreateGame(Player1Colour, Player2Colour)	
addToMoveHistory(Move aMove)	
undoLastMove()	
IsCheckMated()	
getTurn()	
setTurn(PieceColour aColour)	
Abstract methods:	
setTimerValue(decimal x)	
getTimerValue()	
setDifficulty(int Player1, int Player2)	
getPlayer1Difficulty()	
getPlayer2Difficulty()	

The **Player** class represents a player in the game. the player is responsible for moving pieces and so it is under the player class in which the make a move procedure is executed. The player has a **colour** which represent which colour pieces they are using. The other property is **timeLeft** which may or may not apply depending on the game mode. This property could not be dynamic like in the game classes because the players need individual values in the game. This property will update during the game, reducing its value by the time spent by the player making a move.

Player
<b>Private:</b> -playerColour : pieceColour -TimeLeft : Decimal
<b>Public:</b> MakeMove(Piece movingPiece, Square TargetPosition) IsInCheck()

The **makeMove** method is what changes the details of the piece and board when making a move. Being called from a click event, the clicked piece and the target square is passed to this procedure and this updates the piece with its new position. This method also handles the squares on the board too, by setting the occupied property to true or false accordingly. This takes two parameters, the piece that is being moved and the target square that it is being moved to.

The **IsInCheckProcedure** will potentially be one of the most important methods alongside the checkMovelsValid method. This method will be called upon with every move of the game and also when possible moves are being populated. This is because the status of 'check' can arise not only by the movement of the king but by other pieces movement also. This procedure takes the position of the friendly king piece and then scans all the enemy piece's possible moves (using the getPossibleMoves method) Then, if one of those moves is equal to the kings position then the player must be in check.

This is used in conjunction with the **IsCheckMated** method in the game class. When temporary moves are made, this method is run to check whether the result of that move has lead to a 'check' status and if the move turns out to be invalid for this reason, the move can be removed from the list of possible moves available to the player (the array property which lists all the possible square objects for moving).

The **TimedGame** class is subclass which inherits the **game** class. Everything about this class is pretty much the same except this class deals with timer values and so the methods will be overridden slightly to cater for timers.

TimedGame
<b>Private:</b> -Timervalue : Decimal
<b>Public:</b> CreateGame(Overrides) setTimerValue(decimal x)(Overrides) getTimerValue() (Overrides)

The timer value selected from the main menu will be passed through to the game board form which will then be assigned to the **TimerValue** property of this class. Which will then be used in coherence with the timer labels on the form, updating every second. This will probably be part of the "updateView" procedure mentioned earlier, in which the user interface is updated on the tick of a timer, updating the images of the pieces etc.

This class also overrides the abstract methods from the Game class which apply to the timed game. These are get and set Methods for the timer value which is the value retrieved from the main menu form at the beginning of the game.

TimedGame_Maths
Private: -Mathsdifficulty : integer
Public: CreateGame(OVERRIDES) setDifficulty(int Player1, int Player2)(OVERRIDES) getPlayer1Difficulty()(Overrides) getPlayer2Difficulty()(Overrides) askQuestion()

The **TimedGame\_Maths** class is another subclass which inherits the **TimedGame** class. It too, carries all the features of a standard game class but with the added functionality of timers and mathematics mode. Therefore this class inherits the timed game mode to avoid repeating any code to do with timers and this class can focus on the maths questions.

The property **MathsDifficulty** is an integer value which will be gathered from the main menu should the maths game mode be selected. This is what determines the level of difficulty of questions that should be called upon and can be used with the class when retrieving questions from the storage.

The **createGame** method is overridden to include the assignment of the maths difficulty property.

The **askQuestion** method is the procedure that calls upon the third module of the system, the question asking form. This method will execute the form and the contents of that form will be determined by which level difficulty is being requested. Once the question has been retrieved, the ask question form code will then deal with user input and comparisons of answers.

The other methods are for retrieving the values for the difficulty levels of player 1 and player 2 for use when generating the question asking form.

Move
Private: -playerMoved : Player -PieceMoved : piece -initialPosition : square -finalPosition : square -PieceCaptured : piece

The **Move** class is what details every move and the instance of this class will be stored in move history lists which is useful for validation of certain areas of chess.

The class has properties to store the important details of the move.

**PlayerMoved** represents the player who made the move and hence it is of type player.

**PieceMoved** is the piece that was moved by the player, the property is therefore of type piece.

**Initial position** stored the initial square the piece occupied before the move was made and **finalPosition** stores the square that the piece was moved to

A newer property added in the design stage is the **pieceCaptured** property which eliminates the need for an extra property which stores all captured pieces and instead stores them alongside the move that was made when the piece was taken which is much more logical.

The **Pawn** class inherits piece and is the only subclass of piece which involves extra properties and methods.

Pawn
-Promoted : bool
checkMovelsValid (Overrides)
PromotePiece(pieceType)

The pawn has a few special rules, one of which is the promotion of the pawn should it reach the other side. This is a boolean value on the created property **Promoted**. A method **PromotePiece** follows with a parameter of the piece that is wanted. Then it is simply a case of adjusting all the pieces property and changing its type and image to promote the piece.

In the overridden **checkMovelsValid** routine, certain mathematics is behind the moving of the pawn. If it's the pawns first move, it may move two squares forward instead of the usual one. This is checked by the piece's move count property.

A pawn may move one square forward so a the check is that the x coordinate remains the same and the y coordinate increase is only one. if this is the case and the square in front is not occupied (**isOccupied** property) then the pawn may move there. A separate check for pieces to capture will be made and they will be one y value in front of the pawn and  $\pm 1$  x value either side.

Rook
Attribute
checkMovelsValid (Overrides)

The **rook** class inherits Piece and the only difference in the classes is the overridden routine **checkMovelsValid**.

The movement for a rook is horizontal, it may move as many squares as possible in either the x plane or the y plane as long as the movement remains horizontal (there is no diagonal movement).

To code this mathematically, this means the target position of the rook must either be the same x coordinate OR the same Y coordinate. If both the X and Y coordinates are different then the move is invalid.

The rook may also perform a castling procedure and to do this its move count must be 0 and it must be the position 3 x-coordinates away from the king (hence the kingside rook). A check for the king is also performed and under the correct conditions and the two pieces may castle.

A check for the squares between the current position and the final position is done and checking the **isOccupied** property of each square.

Knight
Attribute
checkMovelsValid (Overrides)

The **Knight** class inherits Piece and the only difference in the classes is the overridden routine **checkMovelsValid**.

The movement for a knight is an L shape. It may move 2 squares in any direction, followed by one square to either side.

To code this mathematically, we have to use the fact that the movement is 2 squares in one plane and one square in the other plane. So if the knight has moved 2 coordinates in the Y direction, then it must also have moved 1 coordinate in the X direction. This is done by an absolute value comparison to see whether these rules are kept to for the target position.

A check for the squares between the current position and the final position is done and checking the **isOccupied** property of each square.

Bishop
Attribute
checkMovelsValid (Overrides)

The **Bishop** class inherits Piece and the only difference in the classes is the overridden routine **checkMovelsValid**.

The movement for a bishop is diagonal movement only, there is no horizontal movement. It may move as many squares as possible.

To code this mathematically, we use the fact that the movement of the x and y coordinates increment or decrement proportionally to each other. If the y coordinate changes by 1 then the x coordinate must also change by 1 to keep to a diagonal movement. Then it is just a case of checking statements to cater for all directions.

A check for the squares between the current position and the final position is done and checking the **isOccupied** property of each square.

King
Attribute
checkMovelsValid (Overrides)
CanCastle(Square TargetPosition)

The **King** class inherits Piece and the only difference in the classes is the overridden routine **checkMovelsValid**.

The movement for a king is one single square movement in any direction.

To code this mathematically, we make sure that the change in the X and Y coordinates is maximum of 1 for either which will cater for all directions also.

The king may castle with the kingside rook in which case the move count property is checked and if both haven't moved yet and the squares between the two are unoccupied then the two may castle.

A check for the squares between the current position and the final position is done and checking the **isOccupied** property of each square.

To cater for the castling procedure a CanCastle method will be added to the king class to check whether the move can be made. When populating the possible move for the king, this method will be run to check that the targetPosition is valid for the castling move.<sup>43</sup> This also uses the MoveCount property of the piece class as both the king and castle cannot have moved prior to this move.

The **Queen** class inherits Piece and the only difference in the classes is the overridden routine **checkMovesValid**.

<b>Queen</b>
Attribute
checkMovesValid (Overrides)

The movement for a queen is any direction, any possible amount of squares..

The queen is probably the hardest to code for as you have to cater for all directions and both horizontal and diagonal movement. So the first check is that the movement is either with the same X coordinate, the same Y coordinate (like the rook) OR incrementing/decrementing proportionally (like the bishop). Also the direction needs to be checked to code accordingly. If the movement is none of these then the move is not valid.

A check for the squares between the current position and the final position is done and checking the **isOccupied** property of each square.

The properties of all classes are private and are access via get and set methods as shown. This makes the system robust and effectively structured.

## ● Description of record structure, file organisation and processing

The questions used in the mathematics portion of the project will be stored in a text file (or similar) for ease of access and because a database is not really needed for them.

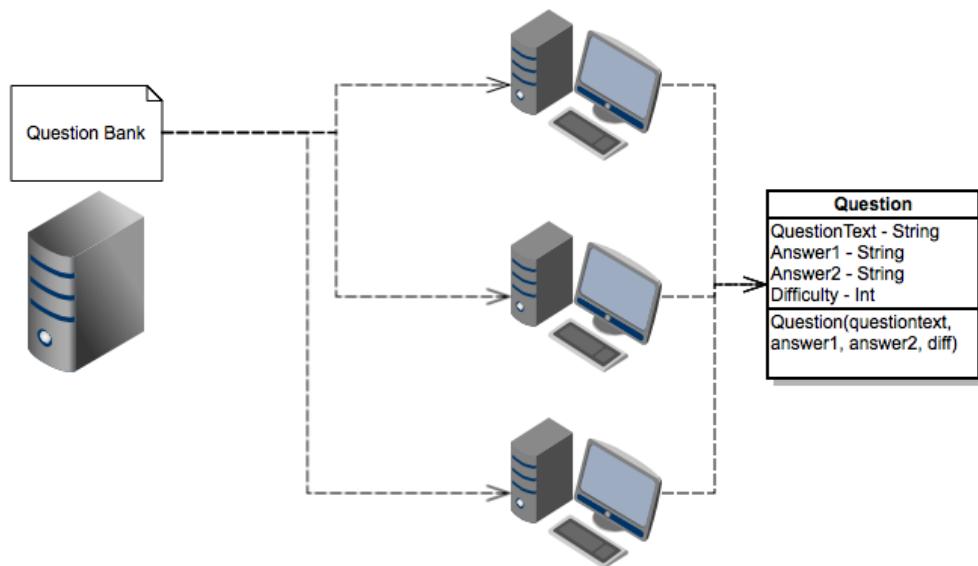
The type of file will be a CSV file. This means that each question and its corresponding details (question text, answers, difficulty etc) can be stored logically in one line of the file and therefore when retrieving the contents of the file, the line can be separated and the question properties can be assigned in a straight forward manor instead of performing various other checks.

The original design plan was to have the file as a binary file so that the contents would be encrypted and unreadable if opened, which is what we want as the students must not have access to the file otherwise they would see the answers to the questions. However this means that the file would have to be stored on every client machine that the software is on. Also if any change has to be made, it would have to be made on all computers individually which is incredibly inefficient. To get round this, it seemed much more logical to use the resources and the environment of the user and have the file stored on the server. This way the file can be pushed out to all the clients.

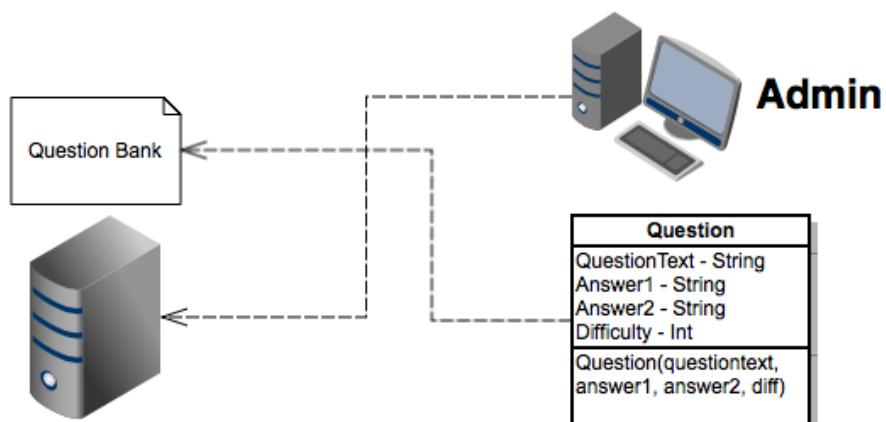
This can be improved though. Instead of having the file pushed out to all the clients, the file should stay on the server as the students or the client machines do not really need to have the file in the local memory. Instead, question objects (with properties questionText etc) will be populated at runtime as discussed earlier from the contents of the file. This approach is much more advantageous for many reasons.

- The students do not come into contact and cannot access the file contents
- The file exists on the server only, no need for client machines to have it in memory

- Question objects dynamically retrieve file info at runtime
- This means the file can be updated and every client will be able to access this updated file without the need for individual change on every client
- The admin user can add/delete questions by logging into **one client only** and make changes to the file stored on the server without having to make the changes to every client.
- This dynamic system means that the admin user can go into **any client** and make changes which will then become global across the whole network



When the admin user logs onto any client machine, the changes made to the questions will be changes to the question objects initially, then a separate procedure will use the list of question objects to rewrite the file and update the text file on the server - this way the process is split up and the user is not directly altering the file.



The whole point of the maths mode is that you can adjust the difficulty of questions asked and they are presented randomly from many categories of questions.

Therefore the first thing in terms of organisation of the data is topic and difficulty. The questions should be organised in topic and the subsequently in difficulty. This way a random topic can be chosen and then from that topic, a random question can be chosen from the applicable difficulty.

The answers of these questions should either be stored separately and referenced by the same number or ID as the question OR the answer to each question should be underneath the question itself to make the process somewhat quicker for comparison.

The process of accessing this data storage comes from the question answering form and the askQuestion method in the game class of the system. When called upon, the program will access these questions and return the randomly selected question onto the form which asks the questions. The question will be presented on the form along with an answer box which gathers user input, the answer may too be gathered swell but hidden on the form until an answer has been submitted.

Looking more into how many questions should be stored, it would be most effective if there were many more questions from each topic than there were moves in the game. This way each game is guaranteed different and also it will take a fair amount of time before the player comes across all questions. Even though repeated questions are not necessarily bad as it will enforce the method into the users head.

Text files can be very small in size so the estimated file size of the total question bank is not expected to be very high. Judging on the basis that a typical question will be around 20-40 characters long and the answers will be around 10-20 characters long, there will be around 90 questions from each topic (35: 35 : 20 in terms of difficulty level).

A way we can estimate this is by creating a text file with some dummy text in to see what file size a typical question would be:

```
test sentence, test sentence, test sentence.
```

Name	test for file sizes
Kind	Rich Text Document
Size	392 bytes
Created	Today 16:54
Modified	Today 16:54
Last opened	Today 16:54

This 6 word text file (including spaces) is 44 characters and the file size is 392 bytes.

**Using the estimations above:**

$$(3) * (60) * (90) = 16200 \text{ characters},$$

Each character amounting to approximately  $(392/44) 8.9\dots$  bytes.

Therefore the question bank will be approximately:

$$\begin{aligned} (16800)(\text{added extra for spaces}) * (8.999) &= 149672\dots \text{ bytes} \\ &= 146\dots \text{kilobytes} \\ &= \underline{\underline{150 \text{ KB (Approx.)}}} \end{aligned}$$

## ● Validation Required

The validation for a chess system is massive. There are constantly checks being made to validate either aid movement of pieces, behaviour of the form, behaviour of the game and other such events. The validation for the system can be broken down into the validation required for the 3 main modules of the system (as shown in the overall system modular design).

### **Main menu:**

The main menu validation is all to do with user input and checking whether all mandatory options have been selected before the game begins.

In the section: a game mode must be selected, each player must have chosen a colour, a board style must be selected, if the game mode is maths then a difficulty must be selected for each player and if a timed game mode, a timer value for the game must also be selected.

A check will need to be performed to see whether all of these necessary options have been selected, as the input approach will be pre-determined options such as list boxes and radio buttons, there will be no need for input conversion.

Some of the checks will be dynamic, for example if the timed game mode is selected then the options for the timer settings should then be available, which they were not previously. This will be a check to see whether the timed game option has been selected - the radio button change event.

The rest will be performed under the action of the start game button or the button that proceeds to the game. Should something not be selected, appropriate feedback will have to be shown to alert the user.

### **Game Board:**

This section has the greatest amount of validation involved out of the form based modules. This form links together with the game engine and provides the user interface.

When setting up the form, the style of the board (the background image of the squares) will be determined by one of the passed properties from the main menu. So for this process to take place, validation will be needed to compare which style has been selected to then subsequent populate the board with the stored image. This also applies to the pieces.

One of the main validation areas on this form is the click events as this is what controls the movement of pieces and represents the moves of each player. When a click is performed, an initial check needs to be made to see if a piece is on the square that has been clicked. This requires a scan through the pieces in the BLACK/WHITE pieceLists to see whether their position is equal to the position clicked on board. If this is true then a movement can be made (shown by a green highlight if this option is turned on - which also requires validation; if this is turned on then squares that represent the piece clicked and possible moves of that piece will have a green border, if this is turned off then the possible moves will not show up in green highlights). If there is no piece on the clicked square then nothing should happen.

During the this first click event, a check needs to be made to see whether the clicked piece matches the colour of who's turn it is, otherwise do nothing because the player cannot move the other player's pieces. After the first check, a secondary set of validation is needed as the second click could represent the square wanted to move to or it could just be another initial click. If the first

click was a piece then there will be set of possible moves, so the secondary click needs to check whether the square clicked second is one of the possible move squares, if true then the move needs to be performed. If not, then the highlighting needs to disappear (if turned on) and nothing should happen.

## **Question Asking Form:**

The validation on this form is mainly to do with user input much like the main menu except with this one a conversion/comparison will need to be made for answers.

The first set of validation checks will be to do with the difficulty selected, depending on this value, a different set of questions will be called upon.

The main validation will be for the user's answer they make with a provided input medium (text box etc) A set of rules will need to be applied to check with the users answers is equal to the answer stored. This will involve comparisons between integer or decimal values and also symbols too as surds involve square root signs. On top of this conversions will need to be made if the input medium is something like a text box which accepts strings, otherwise conversion errors will not be caught.

The answer box input should be limited to prevent overflow errors and null exceptions should be handled by either prompting the user that no answer has been made or simply taking it as an incorrect answer (as they may not know the answer).

## **Code engine:**

The most validation by far is in the chess engine in the coded module. This will constantly performing checks and comparisons to manipulate gameplay.

Probably one of the most prominent validation in the system will be the `checkMovesValid` routine which is designed to be a whole method dedicated to validation of the movement of each piece. The first check that needs to be made is whether the square the piece is trying to move to is on the game board, if this is true then further checks can then be made.

From here the validation algorithms will differ from the type of piece as they all have individual rules for movement (see object design of piece classes for further detail). These will have to involve checking of the target position to see whether that particular piece caters for that movement. This will probably involve mathematics of x and y coordinate planes to determine whether the move is valid.

Not only the validation of the target square itself, but checks will also have to be made to see whether squares leading up to the target square are occupied or not.

Another large validation algorithm will be the one to determine whether the state of the game is check mate. This will involve checking that the king is current IN CHECK and any subsequent movement of the king will cause for it to still be in check. This means checking all the squares around it and seeing if they cause the king to be in check due to enemy pieces, and then returning true of every position available to the king results in a further check. This will probably involve scanning the enemy piece list to see if any of their possible moves is the one which is available to the king, this will have to be done for all available moves to he king for check mate.

The checkmate validation will have to run immediately after the validation for standard check, as nothing else can happen after checkmate.

## **Admin Section:**

The admin section will have lots of input validation to ensure that the user has input the right type of data etc. Also a check will need to be made to see whether all required input has been submitted.

A quite important validation here will be to do with the text file manipulation. Errors will have to be caught and dealt with appropriately. These will include trying to read from the text file and ensuring that no empty lines are read as this would cause an error, writing to the file needs to be corruptly formatted otherwise this will lead to errors showing up later on when reading occurs. The information that the user submits to add a question to the file needs to be added in the correctly format to be consistent in the file and no empty lines should be inserted accidentally.

Catching all these possible errors will make the file handling robust and will ensure that whatever the admin user does, the communication with the file is limited and translated correctly.

Having the contents of the file retrieved at runtime and populating question objects also means that there is not a continuous stream of users trying to access the same file which would cause networking errors. This independence from the server is key to minimising the possible errors that could occur

### **● Identification of appropriate storage media**

The most appropriate medium for storing copies of the software would be CD-ROM. The reason because is that the file size is in the right range for the software (around 700mb max) so there is no waste of data if we used something like a flash drive for the dedicated storage. Also using a CD-ROM means that the data on the CD can not be overwritten and cause any errors in the system data.

The actual system however will be installed on multiple computers in which case will be stored on the computer's hard disk drive. The system will include the executable file. The CD-ROMS are ideal for distributing this software to the computers and once the system is on the computers, no more installation is needed, but the CD's can be used as a backup medium and are there if re-installation is needed.

Alternatively the software, too, can be pushed out via the server in the school on which the question file will be stored. This would be an efficient method of installing on multiple client machines but the CD roms would be useful for further backup purposes.

- Identification of processes and suitable algorithms for data transformation

### **CheckMoveIsValid Algorithm:**

This algorithm plays an important part in the validation of piece movement. The movement of each piece is different due to different rules and this algorithm caters for them all. This checks whether the target position (the position that the piece is trying to move to - the second click on the user interface) is valid for the selected piece.

This algorithm is essentially one entity, but in code it is split up due to the object orientated nature of the system. This algorithm is overridden down the class hierarchy and is different depending on which piece is currently being assessed. Details of all versions are here.

The code first checks that the target position is on the game board and after that the overriding begins. Based on the piece, the target position will be evaluated by comparing x and y coordinates of the target square. This is an enforcement of the rules of chess for the movement of each piece. For example, a king may only move one square at a time (excluding castling procedure) so this can be represented by 1 x or y coordinate change. Using a Cartesian based system, mathematics can be used to evaluate true or false for the movement of the pieces.

This procedure will be called every time a piece on the board is clicked and the possible moves for that piece are populated in the possibleMoves array. To evaluate a square position to be true for a possible move, the checkValidMove method will be run on that square to see whether it is valid for the particular piece. This checks the movement of the piece is correct and also other factors such as whether the piece can capture an enemy piece or other special case moves can be made. But the checkValidMove procedure is the basis of all move validation.

### **Pseudo Code for the algorithm:**

```
public bool checkMoveIsValid(targetSquare, gameBoard) [VIRTUAL]

    if the targetSquare is NOT on the gameBoard then
        return false;

    else
        overriding is performed.
```

```
public bool checkMoveIsValid(targetSquare, gameBoard) [OVERRIDES - PAWN]
```

```
if the targetSquare x-coordinate == currentSquare x-coordinate then
```

```

if the moveCount of the pawn is zero then

    check that the targetSquare is either 1 or 2 y coordinates
    in front of the currentSquare.

    if true then

        check that the squares between the currentSquare and
        the targetSquare
        are not empty

        if one of the squares is occupied

            return false;

        otherwise, return true;

    else

        return false;

else

    Check that the targetSquare is 1 y-coordinate in front of the
    target square

    if true then

        check that the squares between the currentSquare and
        the targetSquare
        are not empty

        if one of the squares is occupied

            return false;

        otherwise, return true;

    else

        return false;

else

    check the two squares that are 1 y-coordinate in front of the
    pawn
    and +/- 1 x-coordinate either side of the pawn.
    If the squares contain an enemy piece,

        return true;

    otherwise,

        return false;

```

**public bool checkMoveIsValid(targetSquare, gameBoard) [OVERRIDES - ROOK]**

If the target square X-coordinate is NOT equal to the current square x-coordinate  
AND  
the target square y-coordinate is NOT equal to the current square y-coordinate

```

        Return false;

    else

        If the target square x-coordinate is equal to the current
        square x-coordinate

            Check all the squares between the current square
            and the target
            square by incrementing/decrementing the y-
            coordinate value

            if square is occupied

                Can the piece occupying the square be
                captured?
                (calls canCapture routine)

                if true

                    return true;

                else

                    return false;

        If the target square y-coordinate is equal to the current
        square y-coordinate

            Check all the squares between the current square and
            the target square by incrementing/decrementing the x-
            coordinate value

            if square is occupied

                Can the piece occupying the square be
                captured?
                (calls canCapture routine)

                if true

                    return true;

                else

                    return false;
    
```

**public bool checkMoveIsValid(targetSquare, gameBoard) [OVERRIDES - QUEEN]**

Check that either:

- The target-X is equal to the current-X
- the target-Y is equal to the current-Y
- the target square is an equal number of x and y increments/
 decrements

if true

```
check the squares in between the current square and the
target square

if square is occupied

    Can the piece occupying the square be
    captured?
    (calls canCapture routine)

    if true

        return true;

    else

        return false;

else

    return false;

public bool checkMoveIsValid(targetSquare, gameBoard) [OVERRIDES - Knight]

Check that the target square is either:
- a change in 2 y co-ordinates and 1 x coordinate
-a change in 2 x co-ordinates and 1 y coordinate

if true

    if square is occupied

        Can the piece occupying the square be
        captured?
        (calls canCapture routine)

        if true

            return true;

        else

            return false;

    else

        return false;

public bool checkMoveIsValid(targetSquare, gameBoard) [OVERRIDES - BISHOP]

Check that the target square is an equal number of x and y
increments/decrements

if true

    check the squares in between the current square and the
    target square
```

```
if square is occupied

    Can the piece occupying the square be
    captured?
    (calls canCapture routine)

    if true

        return true;

    else

        return false;

else

    return false;

public bool checkMoveIsValid(targetSquare, gameBoard) [OVERRIDES - KING]

    if the target square is more than 1 x/y co-ordinate change

        Can the king castle? (performs CanCastle procedure)

    else

        Check that the change in y and the change in x is at most 1
        unit

        if true

            if square is occupied

                Can the king capture this piece?
                (calls canCapture routine)

                if true

                    return true;

                else

                    return false;

            return true;

        else

            return false;
```

When the algorithm "checks squares between the current and target square" this will be done with a FOR loop and will vary for each case depending on the movement.

### FOR EXAMPLE:

For diagonal movement, there will need to be for loops which cater for the move direction (i.e positive x, positive y or positive x, negative y)

The following is an example of diagonal movement in the positive y, positive x direction

IF the target square-X > current square-X AND the target square-Y > current square-Y

```
For( i = current Square - X to target square - X)

    current y = current y + 1;

    if(the square[ i , current y] is occupied)

        Can capture this piece?
        (calls canCapture routine)

        if true

            return true;

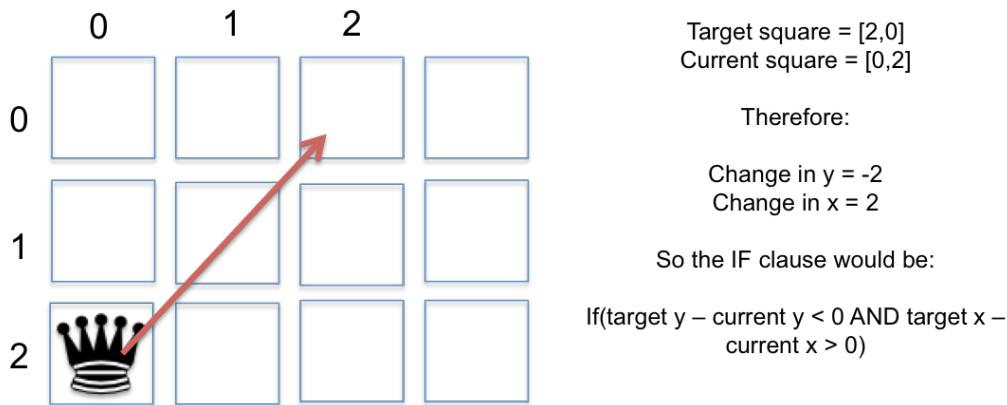
        else

            return false;

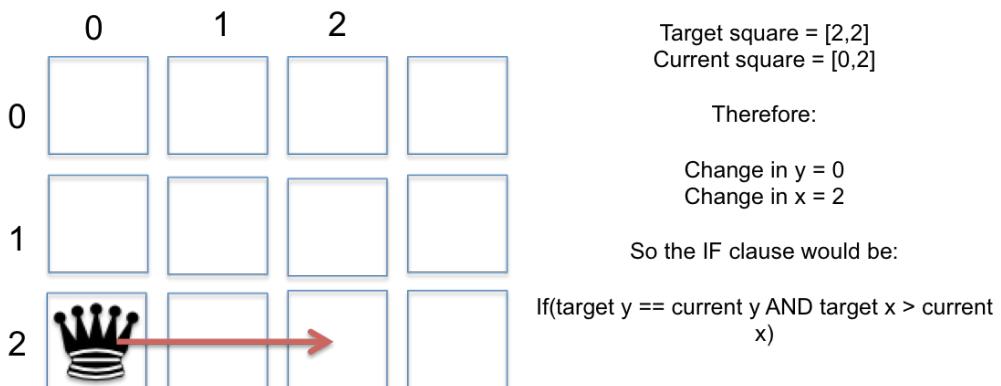
    else

        return true;
```

These loops will change simply by changing the increment/decrement values based on which direction the movement is in.



There will therefore be 4 variations of this, one for each direction of movement.



Again There will therefore be 4 variations of this, one for each direction of movement.

In this procedure, the methods **CanCapture** and **CanCastle** were called upon.

The **canCapture** method evaluates a given position to see whether the piece moving may capture the piece occupying the square.

This will be done by scanning the other players pieces to check whether one of their pieces is occupying the square, if this is true then we may capture the piece. However if not, then the piece occupying the square must be a friendly piece and so we cannot capture it.

Calling this procedure within the **checkValidMove** algorithm is essential to determine whether the given position is a valid move, if the piece occupying the square can be captured.

The **CanCastle** method is a method specific to the king when the king piece is being moved. This also uses the cartesian plane to model whether or not the king can perform the castling procedure. If both the king and the rook have not moved (moveCount = 0) then it is possible, also only if the spaces between the king and rook are unoccupied. The kingside rook could be identified by scanning the list of pieces and finding the rook with an x-coordinate position that is 3 less than the kings.

This method will be called when the king is moving and when the kings possible moves are being populated.

### **IsInCheck Algorithm:**

This algorithm is a key algorithm in the system and although a fairly short one, it is rather important and is used frequently throughout the game (with every move, in fact). The principle is that every enemy piece is checked and each piece has their possible moves evaluated, then if this possibleMoves array contains the position of the friendly king piece then the method returns true to indicate the player is in check. This is incredibly powerful and fairly efficient, proving how object orientation can benefit code in certain situations. Having piece objects, with position properties and methods for populating a list of possible moves is clean and logical. Then only one single scan of the enemy pieces is needed to evaluate the status of the game whereas if I decided on a different design route, with the string array instead of the square object array, many more scans would have

to be made to perform the same procedure (due to comparisons and conversions of both the board and the pieces).

### **Pseudo Code for the algorithm:**

```

Public bool isInCheck()

kingToCheck = piece;

    if current player colour = BLACK

        foreach piece in blackPieceList

            if piece.Type = KING then

                kingToCheck = piece

    if current player colour = WHITE

        foreach piece in whitePieceList

            if piece.Type = KING then

                kingToCheck = piece

    if current player colour = BLACK

        foreach piece in whitePieceList
        {
            if piece.possibleMoves contains kingToCheck.position then

                return true;
        }
    otherwise, return false;

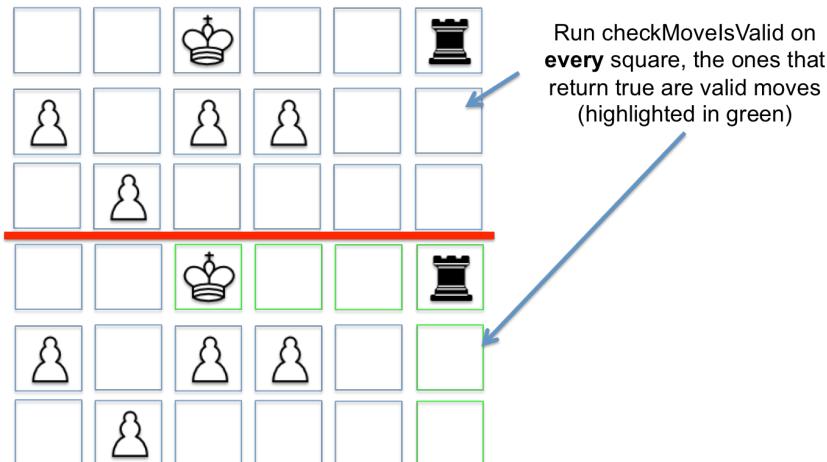
    if current player colour = WHITE

        foreach piece in blackPieceList
        {
            if piece.possibleMoves contains kingToCheck.position then

                return true;
        }
    otherwise, return false;

```

“**blackPieceList**” and “**whitePieceList**” referenced in the algorithm are from the board class and are of type **PieceList** (see object diagram). These are the list population of the white and black pieces on the board. These do not need to be passed as parameters to the algorithm as they are part of the board class and hence also the game class which is stored in the **boardAdmin** class so is available to all other classes (as they are referenced everywhere so this prevents countless passing of arguments).



When populating the possible moves for the black rook, the king is occupying a square that is in the rooks valid moves – This means white is in check.

### IsInCheckMated Algorithm:

This algorithm is another key algorithm that incorporates two other key algorithms; `isInCheck` and `makeTempMove`. In this algorithm, depending on who's turn it is, friendly pieces are scanned and possible moves for each piece are populated. Then for each possible move, the move is made temporarily using the `makeTempMove` procedure. This then makes the move and determines whether making this move results in the 'check' status remaining the same. If this is the case the algorithm sets the possible move to null to represent that this is no longer a possible move as there is no physical move object anymore. This is also to aid the processing in this algorithm. So what happens in this module of code is that it counts up all the null moves after each has been made temporarily and at the end of scanning all the pieces, if the total number of null moves is equal to the maximum value (or 16 null values if the null represents no possible moves for that piece) then this must mean that the player has been checkmated as there are no possible moves from ANY piece that would result in the king NOT being in check.



Once the queen has moved here, this is now check mate. When the possible moves for the king is evaluated, the algorithm `makeTempMove`, moves to all the squares which are available to the king to move, then the `isInCheck` algorithm evaluates whether this move results in check. As all of the king's possible moves results in check – CheckMate! All of the other pieces also have no possible moves as any of their moves will still leave the king in check – which is catered for in the `isInCheck/MakeTempMove` algorithms.

**Pseudo Code for the algorithm:**

```
Public bool isCheckMated()

    int nullCounter = 0

    if game.turn is WHITE then

        foreach piece in blackPieces
        {
            possibleMoves = piece.getPossibleMoves()

            foreach move in possibleMoves

                makeTempMove(move);

                int nullCounter2 = 0
                bool areAllPieceMovesNull = false

                foreach move in possibleMoves

                    if move is null then

                        increment nullCounter2

                    if nullCounter2 equals possibleMoves array length then

                        areAllPieceMovesNull = true
                    else
                        areAllPieceMovesNull = false

                    if areAllPieceMovesNull is true then

                        increment nullCounter

                }

                if nullCounter equals blackPieces List length then

                    return true;
                else

                    return false;

    if game.turn is BLACK then

        REPEAT ABOVE - but scanning white pieces
```

**makeTempMove Algorithm:**

This algorithm came about later in the design stage after revising the current planned model. Having this system, theoretically is very powerful and extremely efficient. This algorithm allows the state of the game to be predicted by making temporary moves and then evaluating the effect of those moves. This could be used for lots of other scenarios and is extremely useful. This reduces the amount of scanning the board and the amount of checking pieces and whatnot and creates a simple, logical procedure in which a future-game state can be simulated.

The algorithm itself uses the built in makeMove method from the player class which takes parameters of the moving piece and target position to make the move by setting the properties of the piece etc. The difference is, this algorithm undoes those moves using the game class's method undoLastMove and performing checks such as isInCheck which then results in possible moves being made null to represent that it is no longer a possible move.

**Pseudo Code for the algorithm:**

```

Public void makeTempMove(Square possibleMove)

    initialise new Move object : tempMove

    tempMove finalposition = possibleMove
    tempMove initialPosition = MovingPiece.position
    tempMove pieceMoved = movingPiece

    if game turn is BLACK then

        foreach piece in whitePieces

            if piece.position equals possibleMove then

                tempMove pieceCaptured = piece

    if game turn is WHITE then

        foreach piece in blackPieces

            if piece.position equals possibleMove then

                tempMove pieceCaptured = piece

    if game turn equals player1.colour then

        player1 makeMove(movingPiece, possibleMove)
        addToMoveHistory(tempMove)

        if player1 isInCheck() is true then
            possibleMove = null

        undoLastMove()

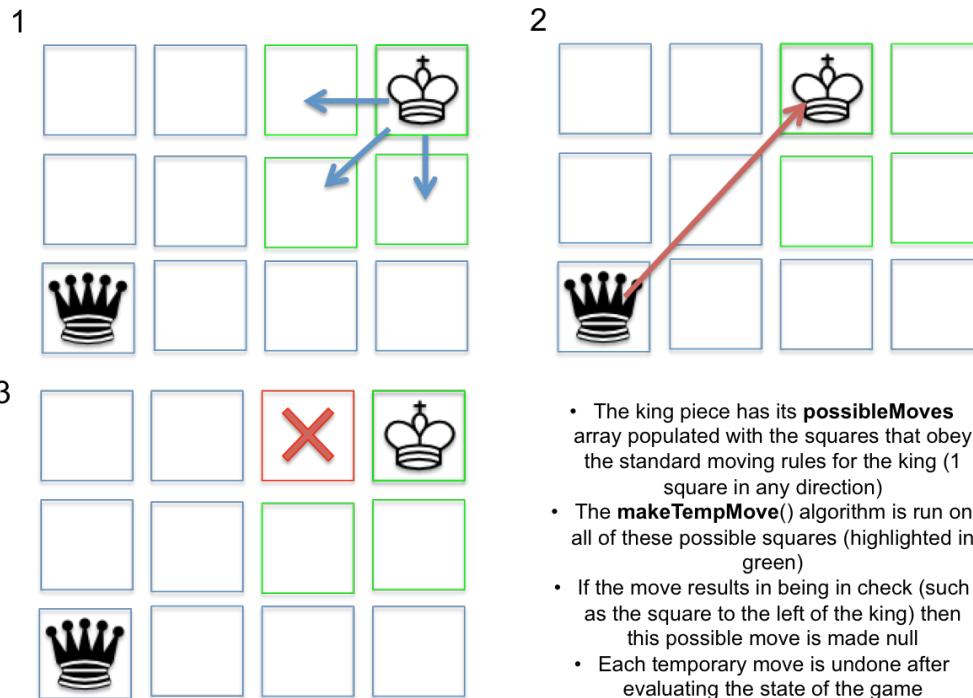
    else
        player2 makeMove(movingPiece, possibleMove)

        addToMoveHistory(tempMove)

        if player1 isInCheck() is true then
            possibleMove = null

        undoLastMove()

```



- The king piece has its **possibleMoves** array populated with the squares that obey the standard moving rules for the king (1 square in any direction)
- The **makeTempMove()** algorithm is run on all of these possible squares (highlighted in green)
- If the move results in being in check (such as the square to the left of the king) then this possible move is made null
- Each temporary move is undone after evaluating the state of the game

The **addToMoveHistory** and **undoLastMove** methods called in this algorithm are methods in the Game class which do as they say. One takes the Move object (populated with properties detailing the move) and adds it to the list of previous moves and the other takes the top-most Move object on the list and uses its properties to reverse the move and remove the move Object from the list.

## ● User Interface Design (HCI) Rationale

The main objective for the user interface designs is clarity. As discussed in the analysis, many other chess softwares are clunky and confusing so the main goal with the GUI designs here is clarity and to do that, components need to be organised efficiently and also need to be simple.

As well as clarity and simplicity, the user needs appropriate visual (and maybe sound) feedback to let them know an action has taken place or confirmation of an action is given. In this context, a main example would be the highlighting of the squares in the game board. This aids the learning of the game by showing all the possible moves for a clicked piece on the board, hence also confirming that the piece was clicked by highlighting it in green also. This confirmation of click event provides a much more satisfactory experience for the user as they know exactly what has taken place.

This idea of confirmation should be consistent throughout the system so perhaps highlighting in green of main menu buttons is also a good idea as this consistency leads to familiarity for the user and again, a more satisfactory experience. Therefore I plan to design my own buttons instead of using the standard controls in the code environment. The reasoning behind this is because by designing my own buttons I can provide much more visually appealing controls but also adding more visibility to the system. The button designs will be related to the action of the button so the user can clearly tell what that button does with little or no text support of this.

To keep the affordance of the controls, however, it will be a good idea to use a combination of radio buttons and my own button designs by linking the two together. This adds extra confirmation by means of the radio button so another layer of clarity is added. The user can clearly tell that the button has been clicked by the highlighting of the button itself but also by the checked radio button.

Another layer to this would be the action of the button when the mouse hovers over it, the button picture could possibly change shade slightly to make it differ from the other consistent buttons and this will provide feedback to say that this button is ready to be clicked, providing more affordance to the control.

Therefore, it is also key that the designed buttons are consistent, with similar designs but only differing in the button content.

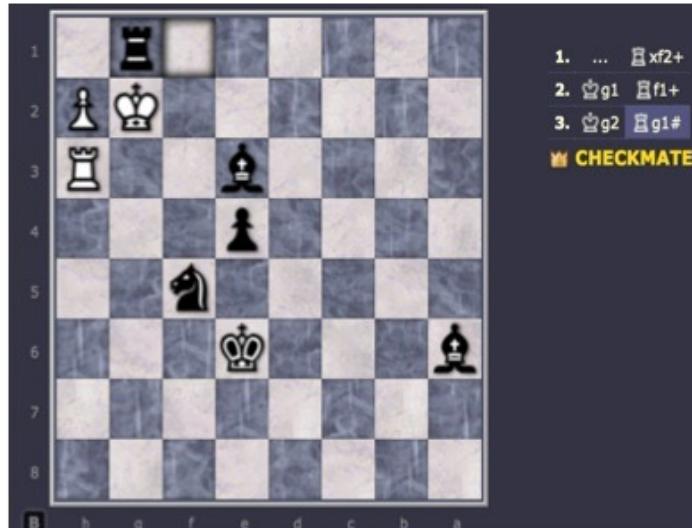
In the main menu, buttons that are not currently available to the user will be dimmed out or not present to clearly state that these are not optional at the current moment. This form will be mainly buttons with little text so the area is uncluttered and easy to understand. The background of which will be a simplistic wooden design to complement the theme of chess.

In the gameBoard form, the area will be clear and will consist of the board itself and only the necessary details such as the timer values for each player and the toggling options (such as the highlighting). The square designs and piece designs chosen by the user will be simple designs to aid the clarity of the game; 3d chess pieces would make the look of the system too complicated - simplicity is the goal here.

Small features will be included on the main game board form to add to the feedback of the game, such as a little chess piece icon with fluctuates between the two player labels, indicating who's turn it is.

**Good/Bad analysis of some existing designs:**

The colours are consistent which make the form look effective – the black and white pieces are plain but effective and make the board clear and attractive. Because the pieces are plain, the board squares can have a slight pattern which is effective.



The move recording is cluttered into the corner but is also unclear. The icons are good feedback but the actual code notation of moves is hard to read and should be a lot easier on the eye.

The checkmate text feedback is also placed right underneath the move recordings which makes that area of the form more cluttered.

The dark background works well as the other components have a light colour outline – this combination works well on the eye.

The colours on this design are in the same spectrum but there are too many variations and makes the form look confusing. The colours should be limited and consistent.



The move list is more spaced out than the previous design so it is easier to read – however the design is confusing as it is hard to tell which move is which as the numbering is confusing.

The chess pieces are not plain and have some extra detail but this works fine as the board squares are simple colours – however with patterned squares this would not be effective and would make the board less easier to look at – so plain pieces is probably better.

This design is effective as the colours are plain and stand out well.

The board squares could have an outline to improve clarity to the form.



The highlighting of possible moves is done in a bad colour – the blue is quite dark and doesn't stand out brilliantly on the board.

Green represents confirmation or “good” so would be ideal for the highlighting but with the green squares this wouldn't look right.

Simple board square colours with simple chess pieces stands out nicely and is clear to look at.

However the board not extremely clear as the squares have white edges.



Having a patterned background takes attention away from the chess board and actually makes the form look cluttered and dense. Also surrounding the actual board with buttons makes the space feel claustrophobic.

However, buttons are consistent with chess board which improves the look of the form.

Having the next pawn as an option just takes up space and adds to clutter

It is clear that when looking at the design and scheme of the form that certain things make up an effective looking board but also what specific details add to clarity of design:

- Having either a light or dark background is effective as the contents of the board stand out - must be an appropriate combination though, light background with dark text or dark background with light text
- Keeping the pieces a simple design means that the actual squares of the board can be either simple colours or patterns also which leads to having an array of options for board style and the more options the user can choose from, the more flexible the design.
- Keeping the area uncluttered adds to clarity and improves how the user feels when using the system. If the system is cramped then the user will feel this when using it. Keeping the forms as clear as possible but making important things organisation and not cramped together will lead to effective viewing for the user.
- The highlighting of possible moves should be a light green colour as green represents: good, confirmation, achievement, possibility, "go" etc and so green is the obvious colour to choose to represent possible moves for a piece. This means that the designs of the board squares cannot be variations of green otherwise the design becomes confusing.

## ● Planned User Interface Designs

### Paper designs:

These are the initial basic hand sketched designs which briefly mock the design of each module in the system.

## Main menu

The buttons will be in conjunction with radio buttons to add to the confirmation that the button has been clicked. (also from a code perspective, it is then easy to evaluate which options have been selected by looking at the radio button properties – their mutually exclusive property helps here.)



The main idea for the main menu module is groups of own made buttons which symbolize certain aspects of the game. These will be the only things on the form to make sure it is clear and simple.

### On the main menu, the main sections will be:

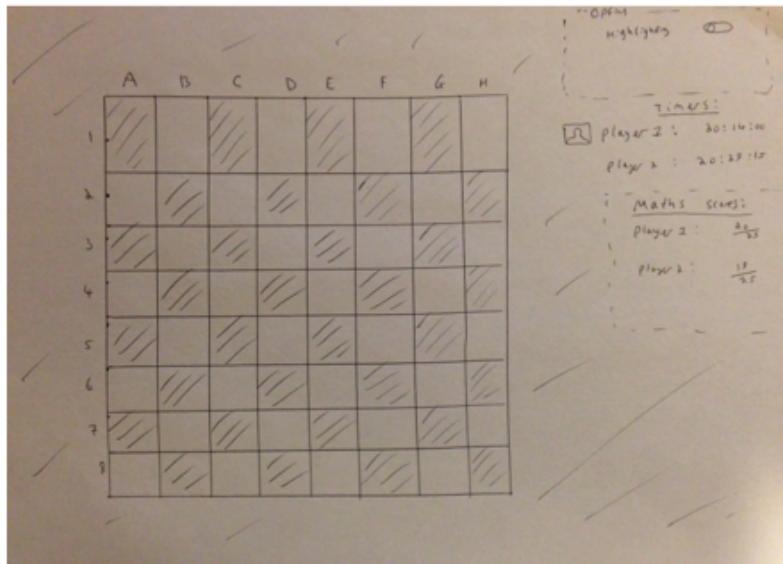
Subsequent Buttons will become available depending on which options have been selected (i.e. the option for a math's game will come after the choice of standard or timed game. The background of the form will be simple too and a light colour to aid clarity – a wooden texture would complement the software theme here, as the default style is wooden.

- Select a game mode (standard or timed)
- Is the game a maths game (yes/no)
- Player 1 piece colour (black/white piece image)
- player maths difficulty (if maths game = yes)
- Play button which proceeds to game

## Game board form

This form will hold the game board which will be made up of picture boxes (which can then have a background and foreground image – background image being the dark or light square design and the foreground being the piece image)

The background of this form will be the same as the other forms, the main focus on this form being the game board.



The board of picture boxes will be indicated by the labels A-H and numbers 1-8 like a realistic chess board and also for reference to the players too. This realism translates to a much more clearer gameplay as it is just like a real life board.

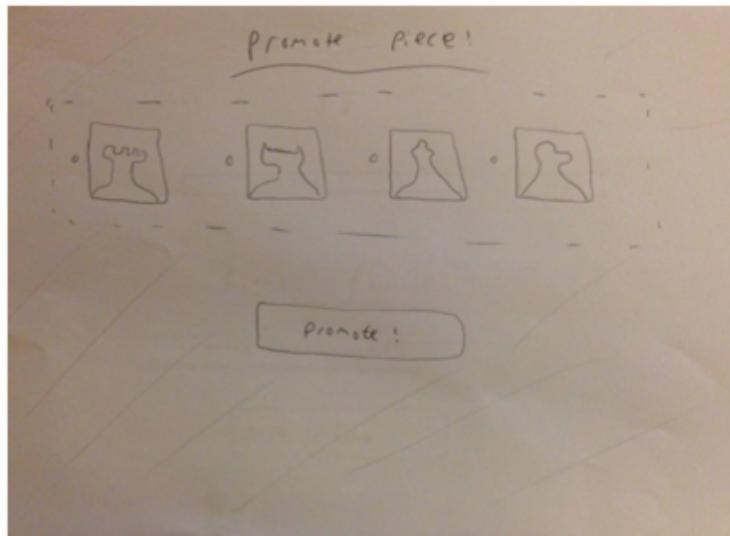
The important details will be placed in the top right of the form, controls to adjust gameplay such as the toggling of highlighting squares or the design of the board (this will either be on the main menu or this form as might be too intrusive on this form).

The players and player timers will be here too in a place next to the game board so it is not in the way, but clear to look at. The indication of turn will be as discussed, the little piece picture showing next to the player labels – a nice little feature which adds to the feedback of the game

## Promoting the pawn module

This is an extremely simple form, but an important one nonetheless. Having the option of promoting the pawn in a separate pop-out module is less intrusive to the game and also logically separates the two modules.

The design of the chess pieces will be mimicked here to remain consistent with game play.



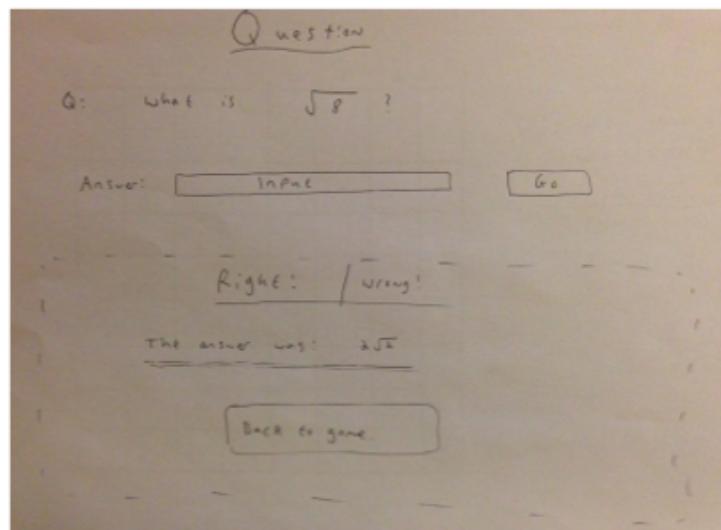
Again, the buttons will be the same design as before and also combined with radio buttons again to remain consistent and to indicate a choice has been made.

The background of this form will be the same as the main menu form and also the game board form too. This keeps the system consistent and looks effective too. This form is another example of why the background should also be simplistic, as the choice of piece is the main feature and so that should stand out the most to the user – not the background.

# Question asking form

The background of this module will be consistent with the rest of the system, so will the fonts too.

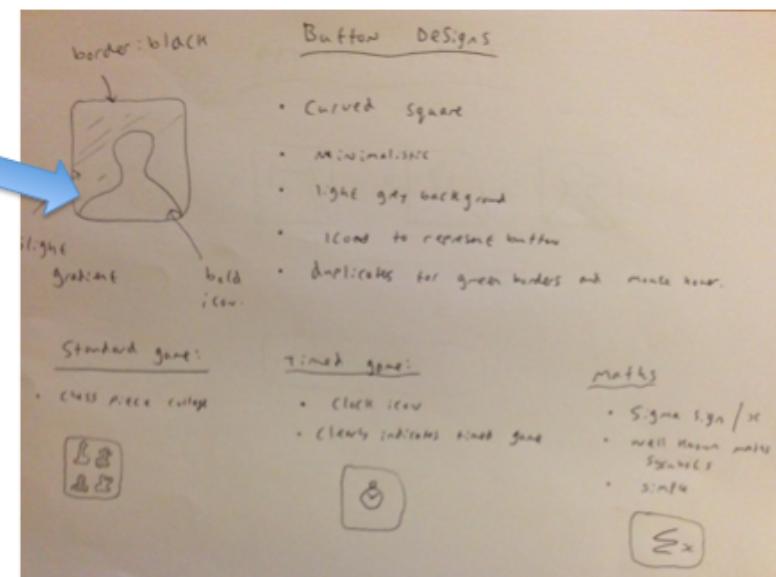
There will be three main parts of this form design: The question retrieved which is asked to the user, the answer reveal and the button to exit the form and return to the game – this logical separation seemed the best way to make the process as simple as possible.



The part where the answer is revealed will be hidden from the user until the answer has been submitted, then this part can be enabled and the answer to the question retrieved from the bank of questions.

# Button designs

The button designs are very simple, which will in turn aid clarity and style. The idea is that the background will be a consistent colour whilst the primary icon on the button will change to represent the action that the button is for. For example, a button with a timer on, represents a timed game.



The buttons will have a black outline to make them stand out amongst the form module, this adds to the affordance of the controls and indicates that they are buttons to be pressed. Also, a separate version will be saved with a green border which will be used to provide feedback that the button has been selected.

The graphics on the buttons will be simplistic too, to be consistent with the rest of the design, actual chess piece images will be used for example, to represent something like a standard game. The buttons will be curved squares because they appear more like buttons than standard pointed squares.

# View Questions

This form will have a list box of items which contain the question text of each question in the file. These will be gathered by looping through each question object currently in memory – as the system loads the contents of the file at runtime. These boxes will be READONLY as this is only for viewing.

The extra options on each of the subsequent admin pages will be a “back” button to aid navigation and to remove the need for closing the form manually. Also a standard icon to represent admin in the top corner for design purposes and consistency.

The selected question will determine what is shown in the other boxes. The other properties of the question object selected will be read and used to populate the answer and difficulty box in the form, so the user can clearly see each question and its properties.

# Add Question

This form will have boxes for each property of a question which allows the user to add a question effectively and easily. To aid this, a set of buttons are available to enter symbols which are not available on the keyboard – this makes it easy and reduces validation needed.

The back button and admin icon will be present to give the system consistency. The buttons will append the symbols to whichever textbox has the current focus.

The add button can be clicked once the data is input and validation will be required to ensure the user has entered the necessary information – a second answer is not required as some questions only have one answer.

# Admin Login

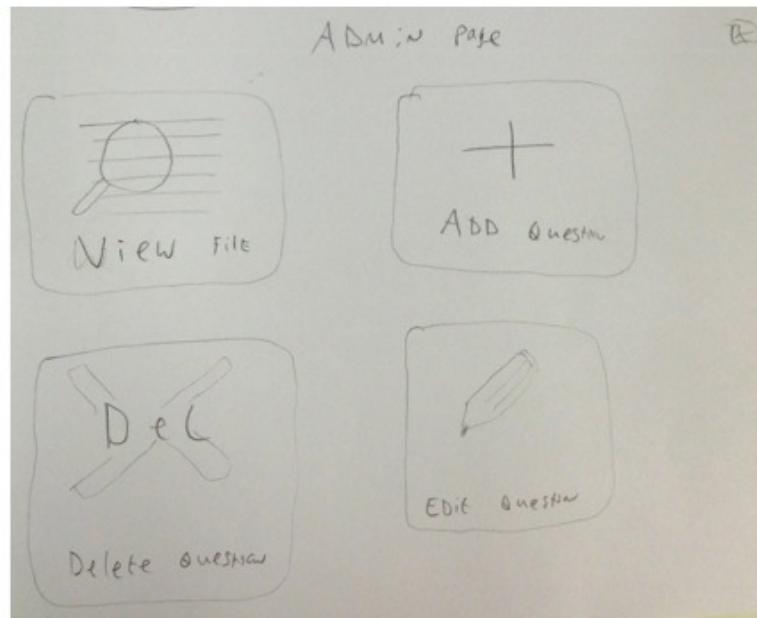
This form will be the first form given to the user after entering the admin section of the system. This will consist of a username and password input from the user which will be validated for incorrect input.

The password box will be encoded with characters that hide the input like in most login systems, this increases security of the admin section as this is accessible from any client machine.

After the enter button is clicked, if the username/password combination is correct, this form will close and the admin page will be opened.

# Admin Page

This form is the base for all the other branches in the admin section. So for clarity, this form will be simply four large buttons illustrating a gateway to the corresponding section.

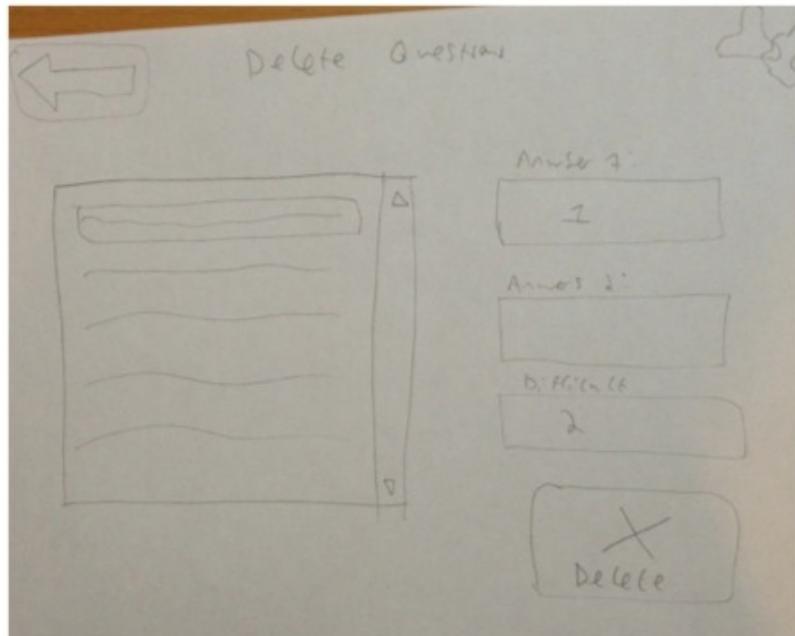


The button icons will represent what the pathway leads to; a magnifying glass for "viewing" the question file, an addition sign for adding a question, a cross for deleting a question and a pencil for editing a question.

For extra user interaction on this form, the cursor of the mouse could change to a hand when its position is over the buttons to indicate it can be clicked.

# Delete Question

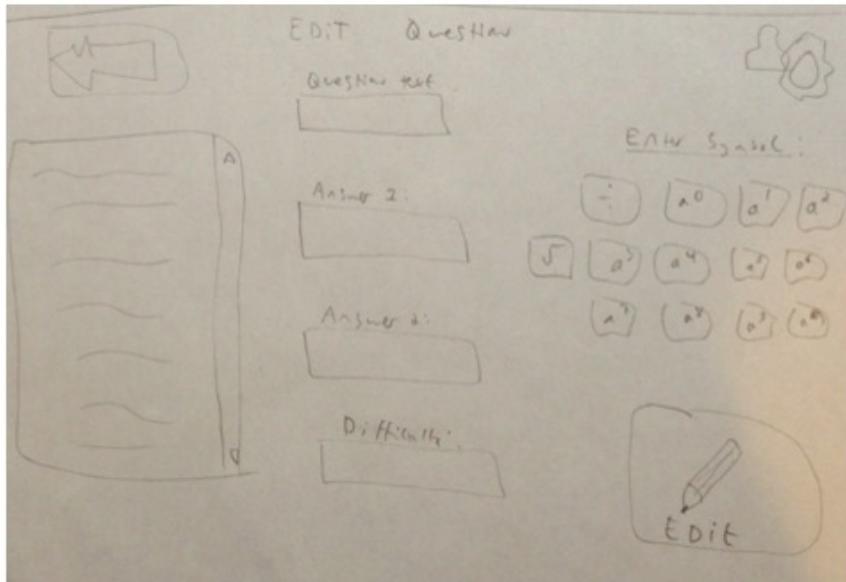
This form will be very similar to the view questions form in the sense that the user can view all the questions and their properties in a read-only format, the only difference is, the selected question can be deleted by clicking the delete button.



The delete button will perform the deletion of the question – probably by updating the list of question objects in the system and then using those objects to rewrite the text file (as a single line in a text file cannot be deleted)

# Edit Question

This form will be very similar to the add question form because the properties of the question can be changed and also the user is presented with a range of symbols for input. The user may select a question from the list and change the properties.

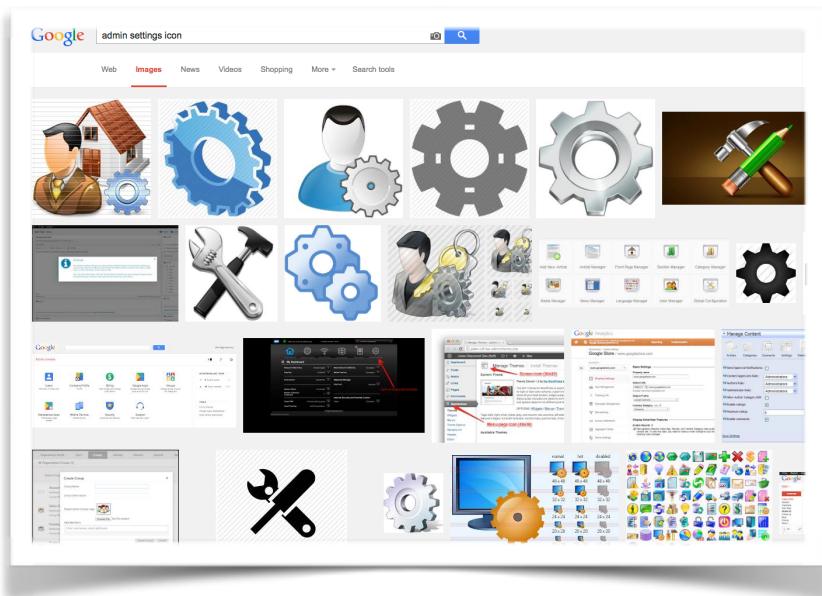


The edit button will perform the update by changing the question object properties and then subsequently updating the text file.

These Design will use calibri font for the symbol buttons as calibri is considered one of the most readable fonts. So for maximum clarity and increase the visibility of the controls. These and other similar buttons will involve this font. The unicode symbols can be quite small so a large font size would also be appropriate.

The buttons on each of the forms will be the same as the button on the main admin page (add question etc) to make the system consistent and not introducing any other foreign controls to the user. Also the headers of each form will be the same with varying titles. The admin logo will be the same and the back button will be present so the user becomes familiar with navigation quickly.

As well as the font being larger, the buttons that perform the actions are also large bandstand out so the user can be certain that they have to click this control to perform the action wanted.



For the admin icon, a generic icon that is recognisable would be the most effective. As shown left, a standardised icon to represent settings/admin is tools or a cog. This is recognised by many people and has become the standard go-to icon because so many people instinctively know what this represents.

With this in mind, a cog in the icon would be a good idea and mixing this with the chess theme would be a nice touch, but simplicity is also important:



Using the same chess pieces as used in the system will provide extra consistency. The combination of the chess piece and cog symbol clearly represents settings for the system as it is related (the chess piece) and there is a standard icon (the cog).

## Button icons:

**Standard game** - to represent a game is just normal with no timers or no maths, chess pieces would be a good representation of this so the button for standard game should be something along the lines of 4 chess pieces on the button, 2 black and 2 white to indicate a standard game.

**Timed game** - the most obvious way to represent a timed game is with a clock icon on the button so this button should have a clock on it to represent a timed game.

**Player 1 colour choices** - These should be a clear indicator of choosing black or white - a clear way to do this is with a picture of a chess piece in the corresponding colour. (i.e. a pawn in black and white).

**Maths/no Maths** - to indicate this, a common mathematics symbol should be used such as PI or SIGMA, this way the button is recognised as something to do with maths. As with the networked buttons, a red cross can be used to indicate a game mode with no maths.

**Promoting pawn buttons** - These should be simple representations of the piece the button will promote the pawn to, so either a knight, bishop, rook or queen. The style of the piece on the button should correspond to the the gameplay style.

**Undo Button** - This should be instantly recognisable that this button will undo a move made. This is generically shown by an anticlockwise arrow so this should be used so the user knows straight away what this does. Having the word undo on the button may also add to the confirmation.

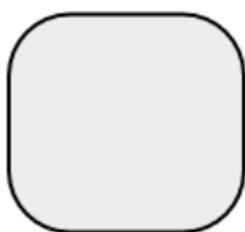
**GameBoard style choices** - These should represent instantly what style this selection will make the game board. As the board is made up of light and dark squares, perhaps an icon showing the two squares is a clear way of showing the style to the user.

**Back Button** - A generic button so this should be clear. A back arrow is probably the most effective here.

**Add/Delete buttons** - adding and deleting are important actions so these buttons should also be quite clear. A green addition sign would represent “adding” of something while being green to represent confirmation. A red cross would represent deletion while the red would represent warning.

**Viewing/Editing buttons** - Slightly less generic buttons so the contents of the buttons should make the action clear - viewing could be represented by a pair of binoculars or a magnifying glass or similar while editing could be represented by a pencil or toolbox or similar.

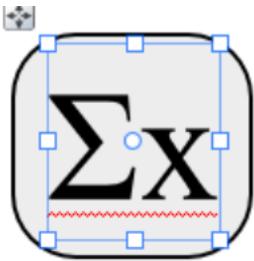
**Unicode symbol buttons** - These should be extremely clear, so as discussed, appropriate size and fonts. Also the button should contain only the symbol that the button will provide.



As planned, a curved square with black border and a light colour background.

With a light grey background, the contrast between the grey and black makes the button stand out quite a lot, so potentially good combination.

The light green colour as a secondary border colour works quite well to indicate that the button has been selected.

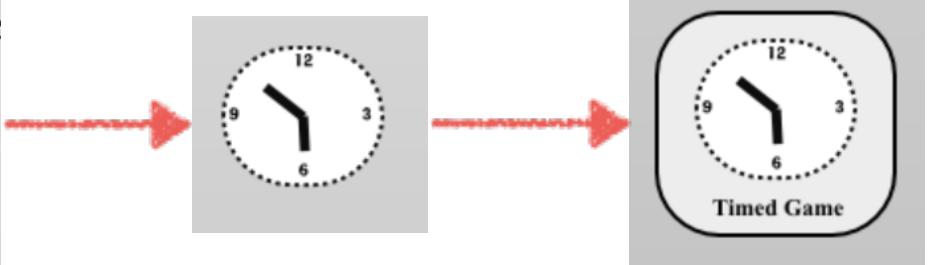
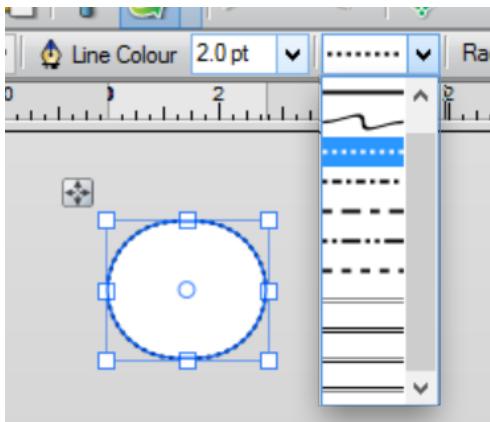
**prototyping button designs:**

The sigma sign works well for the maths button, with bold black text, the contrast between the grey and black again, makes the button stand out. Using times new roman actually works quite well as the pointed nature of the font adds a better effect to the button opposed to more 'fun' fonts.



Adding a slight white shadow to the sigma sign worked quite well with the design and gave it a slight 3D nature too instead of a 'flat' design. This is not too much, however, as simplicity is the goal here.

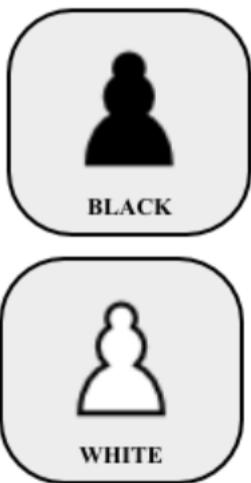
Adding the title of the button seemed to be worthy as the



The clock seemed to work well to represent a timer button, changing the circle border to discontinuous lines, made the icon stand out better as with a solid border, it matched the border of the button itself, so was too similar.



The chess pieces arranged in black and white pairs works well for the standard game mode button. The design is simple but clear.



Extremely simply designed but the intended purpose is clear and to the point. These are possibilities for the buttons which are used to select the players colour in the game.



The actual chess pieces are rather simple black and white icon-style designs. This simplicity ties in quite nicely with the different style of boards and adds a lot of clarity to positioning of pieces and the state of the game. This way the pieces remain the same but the board itself can change.

After experimenting with other pieces and colours, a lot of uncleanness came about, so it would probably be best to stick to these black and white icon pieces.

These are of course royalty free images.



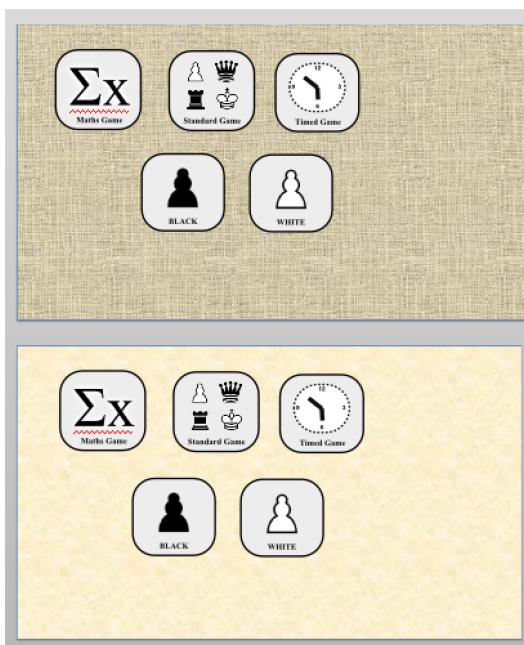
Similarly with the maths button, a red cross seemed the best way to represent no maths, however the red cross conflicted with the 'x' in the button icon, so a red circle seemed better suited to this button.



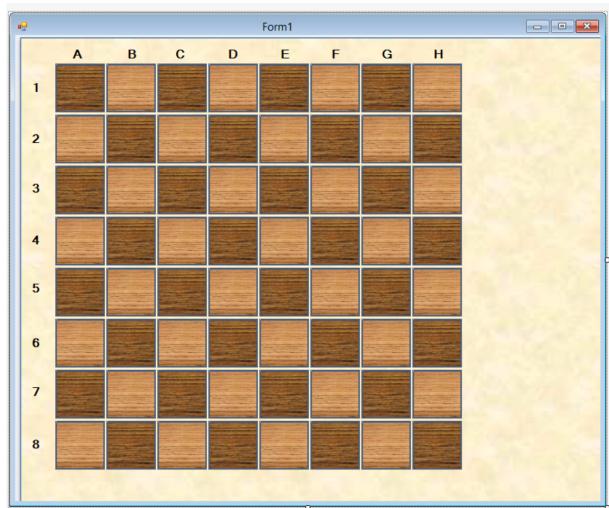
With all of these buttons, a simple change of border creates a whole other set of the same buttons but with the green highlighting. These will be the buttons the images change to if one is selected. The light green proves to be a clear, effective colour to represent this confirmation of selection.



Quite an effective way of representing board styles when offering the selection to the user is to use the squares from the board design and show the lighter and darker share together. This looks effective but also makes it quite clear what style of board you are choosing.

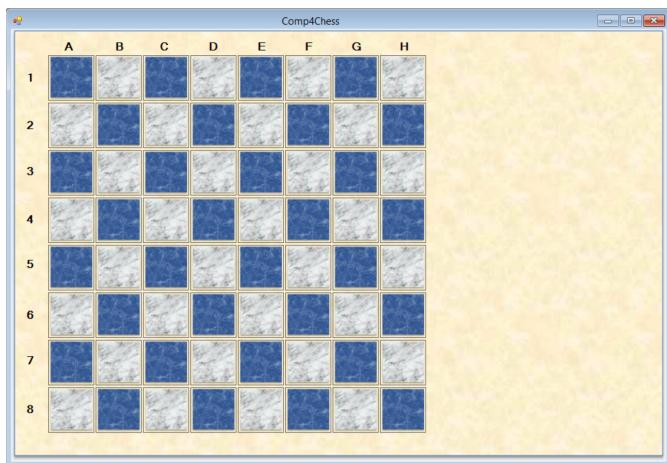


Testing these buttons with various texture backgrounds is a good way to tell whether they fit. The current designs seem to bode well with most backgrounds as they are quite simple and go with a lot of colours. The simple textures found from most vector drawing programs are quite useful here and prove the most effective. The browns and beiges give a nice feel to the style of the system, keeping some traditional look and feel to the game without having too much of a modernised look.



This background texture needs to go well with the game board form as well and so a mockup was needed.

The individual squares with dark wood and light wood textures will be a style optional in the game but the squares looked realistic which was a good thing for the system. Therefore having these in place, the lighter background seemed more appropriate for clarity purposes.



Another possibility is a marble option instead of a wooden style. This option will be available to choose from at the main menu.

Originally, this design was white and brown marble but the brown marble made the black chess pieces hard to see, so blue marble was a more suitable design choice, as clarity is a must.

Other designs will be done in a similar way, then each can be a choice at the beginning of the game.



A simple mock up of the promote pawn module in the system.  
Making sure the background and buttons are consistent.

This form is a simple selection so there will not be a lot present. So



When the timed game is selected, the other groups of controls that depend on a timer game become available.

The math's difficulty selection is a discrete numeric selection from 1 -3

The style of the game board can be chosen here too, it is easier to select it here and also more logical. It also removes clutter from the game board form.

For the timer value, a numeric box is used as discrete values can be stored and selected. By setting a minimum value of 10 and maximum of 50, with increments of 5 this control allows effective selection of the timer value in the game.



Some groups of controls only become available when the corresponding selection has taken place.

If the standard game is selected then there is no need to select math's or timer values

The form just contains simple buttons to determine the nature of the game.

The simple beige background complements the system design whilst keeping the layout clear and easy to understand.

The buttons are arranged into groups based on what they do.

The buttons, when selected change image to that with a green border to confirm the selection of that button. With the help of the mutual exclusive nature of radio buttons, it means only one from each group can be selected which is extremely handy.

## ● Measures planned for security and integrity of data

The system is a game and revision program and therefore the data is not really in need of any security as the data is not sensitive and does not contain anyone's personal details. Data will, however, need to remain consistent and important changes cannot be made by the user. In terms of my system, this means the transition between the system and the questions file storage. Ensuring that communication between the two is secure and will not fail, catching any exceptions if necessary. If the program tries to read from a line that is not there or a blank line instead of a question answer, these need to be handled appropriately with exception handlers and not just let the user know but process the request again potentially, to ensure a question is presented to the user.

As discussed, to keep the question data storage secure, it file will be stored on a sever so that there is no interaction between the user and the file (except the admin user updating the file).

## ● Measures planned for system security.

The system itself will be installed on many target computers as discussed earlier and so for a secure system, the system files (such as the question bank and resources folder) will need to be locked down from the user environment so no one can go in and accidentally delete files which will then in turn corrupt the system. Only the executable file should be available to the user on the target computers. The school's network and use of user logins will handle most security and the actual program files will be pushed onto the clients C drive from the server. The C drive is inaccessible to student logins so the users will not be able to go into the program files.

If something does happen, the system corrupts or is deleted, a backup will be on the CD-ROM on which the software is installed. As the software is to be installed on computers, the CD-ROM is not needed afterwards so can be kept for backup purposes - a main advantage of using CD's.

## ● Overall Test Strategy

With a chess system, there is a lot of validation involved to make sure the game runs as it should. Also with the maths side of the game comes a lot of user input validation too. With validation comes testing to see whether these validation methods have worked as they should.

So on the most part, the system will need heavy whitebox testing, to test each pathway of code and to make sure that the algorithms coded perform the correct procedures.

This will be the case for most procedures in the code so a strategy I would have for the slightly larger algorithms that are quite dense with code would be to run the program first to see what the outcome is, then based on the outcome, identify an area to start debugging. For example, if the rook's possible moves showed up diagonal squares, or one horizontal square was not shown, then we can narrow the problem down to the check move algorithm for the rook specifically.

Once we know what area to look into, a good method of reaching code efficiently without running through hundreds of lines is to use pop up message boxes. These allow you to then pause the run time of the program and go straight to where the message box is called and then you can debug line by line from there. This method will be extremely efficient as the amount of code is large.

In terms of the user input validation, this can be tested by inputting right and wrong answers or doing and performing actions that you know should bring up an error, this way you can narrow down anything that isn't yet validated and also checks that the existing validation functions correctly.

Once the system is complete, a good test strategy would be to get two students from the chess club to sit down and play a game of chess. Any problems they have will then be highlighted. Having more than one pair of students play a game would be key too, as statistically, it is more efficient for testing to have the whole chess club play in fact. So a 'dummy run' one week of chess club would be an advantageous test plan.

Key tests that will be made:

- Do all pieces show the correct possible moves in green when clicked
- Do special moves work correctly (e.g. castling and pawn promotion)
- Does the game recognise checkmate
- Does the system prevent movement of other pieces when the king is in check

# Technical Solution

## Main Menu



```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO;

namespace test_chess
{
    public partial class MainMenu : Form
    {

        BoardUI gameBoardForm;

        public MainMenu()
        {
            InitializeComponent();
            // sets these controls to not enabled as they are not immediately accessible
until other selections have been made
            MathsDiffGroupBox.Enabled = false;
            MathsGroupBox.Enabled = false;
            TimerGroupBox.Enabled = false;

            Player1MathsDifficulty.ReadOnly = true;
            Player2MathsDifficulty.ReadOnly = true;
            timerValue.ReadOnly = true;
        }
    }
}

```

```
Player1MathsDifficulty.BackColor = Color.White;
Player2MathsDifficulty.BackColor = Color.White;
timerValue.BackColor = Color.White;

// gathers the questions from the text file on the sever and stores the contents
in question objects in the system.
GatherQuestionsInputFromFile();

}

// selections are shown with green highlighting
private void StandardGame_CheckedChanged_1(object sender, EventArgs e)
{
    if (StandardGame.Checked == true)
    {
        StandardGame.BackgroundImage =
test_chess.Properties.Resources.StandardButton_GreenBorder;
    }
    else
    {
        StandardGame.BackgroundImage =
test_chess.Properties.Resources.StandardButton;
    }
}

private void TimedGame_CheckedChanged(object sender, EventArgs e)
{
    if (TimedGame.Checked == true)
    {
        TimedGame.BackgroundImage =
test_chess.Properties.Resources.TimedGameButton_GreenBorder;
        MathsGroupBox.Enabled = true;
        TimerGroupBox.Enabled = true;
    }
    else
    {
        TimedGame.BackgroundImage = test_chess.Properties.Resources.TimedGameButton;
        MathsGroupBox.Enabled = false;
        TimerGroupBox.Enabled = false;
    }
}

private void MathsGame_CheckedChanged(object sender, EventArgs e)
{
    if (MathsGame.Checked == true)
    {
        MathsGame.BackgroundImage =
test_chess.Properties.Resources.MathsButton_GreenBorder;
        MathsDiffGroupBox.Enabled = true;
    }
    else
    {
        MathsGame.BackgroundImage = test_chess.Properties.Resources.MathsButton;
        MathsDiffGroupBox.Enabled = false;
    }
}
```

```
private void NoMathsGame_CheckedChanged(object sender, EventArgs e)
{
    if (NoMathsGame.Checked == true)
    {
        NoMathsGame.BackgroundImage =
test_chess.Properties.Resources.NoMathsButton_GreenBorder;
    }
    else
    {
        NoMathsGame.BackgroundImage = test_chess.Properties.Resources.NoMathsButton;
    }
}

private void Black_CheckedChanged(object sender, EventArgs e)
{
    if (Black.Checked == true)
    {
        Black.BackgroundImage =
test_chess.Properties.Resources.BlackButton_GreenBorder;
    }
    else
    {
        Black.BackgroundImage = test_chess.Properties.Resources.BlackButton;
    }
}

private void White_CheckedChanged(object sender, EventArgs e)
{
    if (White.Checked == true)
    {
        White.BackgroundImage =
test_chess.Properties.Resources.WhiteButton_GreenBorder;
    }
    else
    {
        White.BackgroundImage = test_chess.Properties.Resources.WhiteButton;
    }
}

private void WoodStyle_CheckedChanged(object sender, EventArgs e)
{
    if (WoodStyle.Checked == true)
    {
        WoodStyle.BackgroundImage =
test_chess.Properties.Resources.WoodStyleButton_GreenBorder;
    }
    else
    {
        WoodStyle.BackgroundImage = test_chess.Properties.Resources.WoodStyleButton;
    }
}

private void MarbleStyle_CheckedChanged(object sender, EventArgs e)
{
    if (MarbleStyle.Checked == true)
    {
        MarbleStyle.BackgroundImage =
test_chess.Properties.Resources.MarbleStyleButton_GreenBorder;
    }
    else
```

```
        {
            MarbleStyle.BackgroundImage =
test_chess.Properties.Resources.MarbleStyleButton;

        }
    }

// sets the game object and populates it accordingly
private void PlayButton_Click(object sender, EventArgs e)
{
    Game game1;
    String gameStyle = "NULL";
    Piece.Piececolour player1Colour = Piece.Piececolour.BLACK ;

    if (StandardGame.Checked == true)
    {
        game1 = new StandardGame();
        if (ValidateSelection(ref player1Colour, ref gameStyle) == true)
        {
            game1.setGameType(Game.gameType.Standard);
            BoardAdmin.game1 = game1;
            BoardAdmin.gameStyle = gameStyle;
            BoardAdmin.player1Colour = player1Colour;
            gameBoardForm = new BoardUI();
            gameBoardForm.Show();

        }
    }
    else
    {
        if(MathsGame.Checked == true)
        {
            game1 = new MathsGame();
            if (ValidateSelection(ref player1Colour, ref gameStyle) == true)
            {
                game1.setGameType(Game.gameType.Maths);
                game1.setTimerValue((int)timerValue.Value);
                BoardAdmin.game1 = game1;
                BoardAdmin.gameStyle = gameStyle;
                BoardAdmin.player1Colour = player1Colour;
                BoardAdmin.player1Diff = (int)Player1MathsDifficulty.Value;
                BoardAdmin.player2Diff = (int)Player2MathsDifficulty.Value;
                BoardUI gameBoardForm = new BoardUI();
                gameBoardForm.Show();

            }
        }
        if (NoMathsGame.Checked == true)
        {
            game1 = new TimedGame();
            if (ValidateSelection(ref player1Colour, ref gameStyle) == true)
            {
                game1.setGameType(Game.gameType.Timed);
                game1.setTimerValue((int)timerValue.Value);
                BoardAdmin.game1 = game1;
                BoardAdmin.gameStyle = gameStyle;
                BoardAdmin.player1Colour = player1Colour;
                BoardUI gameBoardForm = new BoardUI();
                gameBoardForm.Show();

            }
        }
    }
}
```

```
        }
        if (MathsGame.Checked == false && NoMathsGame.Checked == false)
        {
            System.Windows.Forms.MessageBox.Show("Please select an Option for a
maths game");
        }

    }

}

// makes sure that all the necessary selections have been made
private bool ValidateSelection(ref Piece.Piececolour player1Colour, ref String
gameStyle)
{
    if (Black.Checked == false && White.Checked == false)
    {
        System.Windows.Forms.MessageBox.Show("Please select an Option for Player1
Colour");
        return false;
    }
    else
    {
        if (Black.Checked == true)
        {
            player1Colour = Piece.Piececolour.BLACK;
        }
        else
        {
            player1Colour = Piece.Piececolour.WHITE;
        }
    }

    if (MarbleStyle.Checked == false && WoodStyle.Checked == false &&
PlainBlueStyle.Checked == false && PlainRedStyle.Checked == false &&
PlainPurpleStyle.Checked == false && PlainBrownStyle.Checked == false)
    {
        System.Windows.Forms.MessageBox.Show("Please select an Option for game board
style");
        return false;
    }
    else
    {
        if (MarbleStyle.Checked == true)
        {
            gameStyle = "MARBLE";
        }
        if (WoodStyle.Checked == true)
        {
            gameStyle = "WOOD";
        }
        if (PlainBlueStyle.Checked == true)
        {
            gameStyle = "PLAINBLUE";
        }
        if (PlainRedStyle.Checked == true)
        {
            gameStyle = "PLAINRED";
        }
    }
}
```

```
        if (PlainPurpleStyle.Checked == true)
        {
            gameStyle = "PLAINPURPLE";
        }
        if (PlainBrownStyle.Checked == true)
        {
            gameStyle = "PLAINBROWN";
        }

    }
    return true;
}

private void MainMenu_Load(object sender, EventArgs e)
{
}

private void PlainBlueStyle_CheckedChanged(object sender, EventArgs e)
{
    if (PlainBlueStyle.Checked == true)
    {
        PlainBlueStyle.BackgroundImage =
test_chess.Properties.Resources.PlainBlueStyleButton_GreenBorder;
    }
    else
    {
        PlainBlueStyle.BackgroundImage =
test_chess.Properties.Resources.PlainBlueStyleButton;
    }
}

private void PlainRedStyle_CheckedChanged(object sender, EventArgs e)
{
    if (PlainRedStyle.Checked == true)
    {
        PlainRedStyle.BackgroundImage =
test_chess.Properties.Resources.PlainRedStyleButton_GreenBorder;
    }
    else
    {
        PlainRedStyle.BackgroundImage =
test_chess.Properties.Resources.PlainRedStyleButton;
    }
}

private void PlainBrownStyle_CheckedChanged(object sender, EventArgs e)
{
    if (PlainBrownStyle.Checked == true)
    {
        PlainBrownStyle.BackgroundImage =
test_chess.Properties.Resources.PlainBrownStyleButton_GreenBorder;
    }
    else
    {
        PlainBrownStyle.BackgroundImage =
test_chess.Properties.Resources.PlainBrownStyleButton;
    }
}

private void PlainPurpleStyle_CheckedChanged(object sender, EventArgs e)
{
    if (PlainPurpleStyle.Checked == true)
    {
        PlainPurpleStyle.BackgroundImage =
```

```
test_chess.Properties.Resources.PlainPurpleStyleButton_GreenBorder;
    }
    else
    {
        PlainPurpleStyle.BackgroundImage =
test_chess.Properties.Resources.PlainPurpleStyleButton;
    }
}

// retrieves the contents of the text file and splits each line up and populates a
question object
public void GatherQuestionsInputFromFile()
{

    const string filename = "QuestionBank 2.csv"; //FilePath of the question
textfile.
    StreamReader aSW = new StreamReader(File.Open(filename, FileMode.Open)); ////
opens a new stream reader which can read from the file
    BoardAdmin.questions = new List<QuestionBank>(); // initiates the list of
questions variable in the board admin class

    while (aSW.EndOfStream == false) // happens until the end of the file has been
reached
    {
        string[] a = aSW.ReadLine().Split(','); // read a line in from the file and
split the data from that line based on the comma separation, store each portion in a string
array

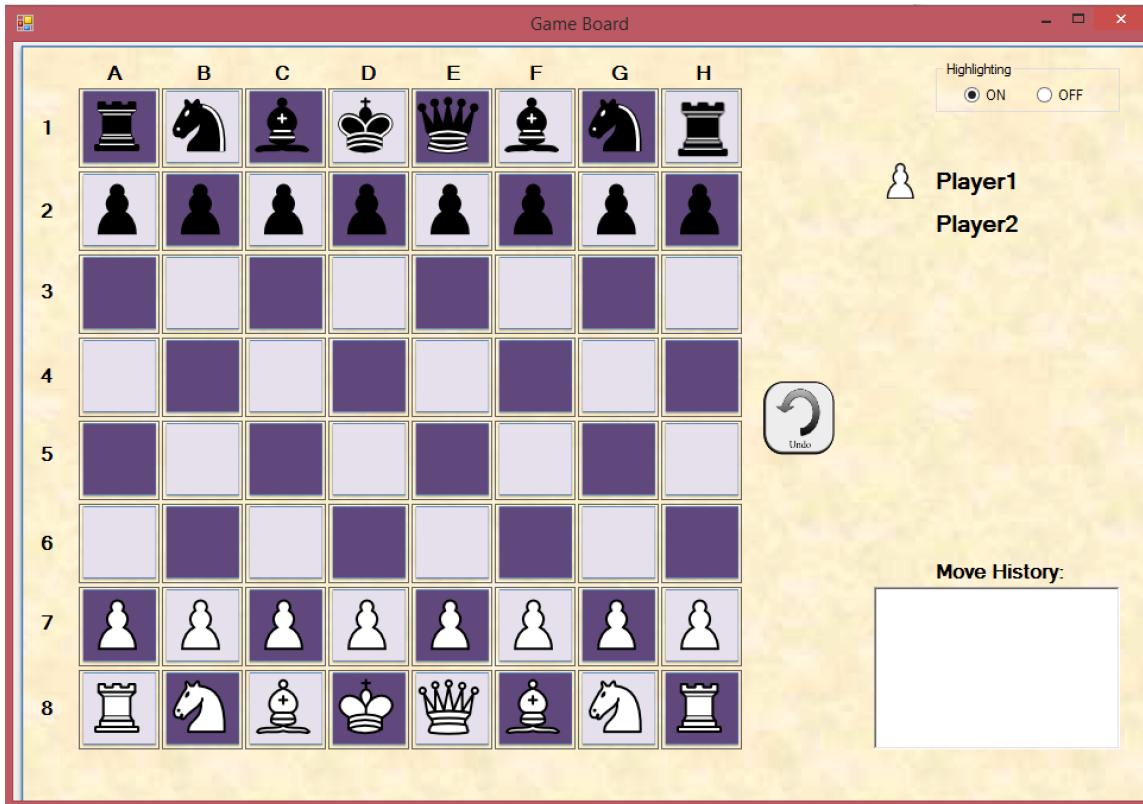
        // adds a question object at the same time as populating that object using
the overridden constructor
        BoardAdmin.questions.Add(new QuestionBank(a[0], a[1], a[2],
Convert.ToInt16(a[3])));
    }

    aSW.Close(); // closes the stream
}

private void AdminButton_Click(object sender, EventArgs e)
{
    // creates the admin login form
    AdminLogin aAdminLogin = new AdminLogin();

    if (aAdminLogin.ShowDialog() != System.Windows.Forms.DialogResult.OK)
    {
        //Waits until the question form has been dealt with
    }

}
}
```

BoardUI (GameBoard)

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO;

namespace test_chess
{
    public partial class BoardUI : Form
    {
        //variables referenced everywhere, so made global
        PictureBox[,] squares = new PictureBox[8, 8];      // picturebox array
        Board gameBoard;
        PieceSet[] pieceSets;
        PictureBox currentPicBox;   // current picture box being processed
        Piece clickedPiece;         // current piece being processed
        Square[] possiblemoves;    // the current possible moves for a clicked piece

        public BoardUI()
        {
            InitializeComponent();
            assignBoardsquares();
            Piece.Piececolour player2Colour;
        }
    }
}

```

```

//set player 2 colour, now we know player 1 colour
if (BoardAdmin.player1Colour == Piece.Piececolour.BLACK)
{
    player2Colour = Piece.Piececolour.WHITE;
}
else
{
    player2Colour = Piece.Piececolour.BLACK;
}
//create the game by calling the method in the game class.
//this will override depending on what type of game it is
BoardAdmin.game1.createGame(BoardAdmin.player1Colour,player2Colour);
gameBoard = BoardAdmin.game1.getGameBoard();
pieceSets = BoardAdmin.game1.getGameBoard().getPieceSets();
BoardAdmin.game1.setGameResult(Game.result.None);

//if the game mode is timed or maths, then set the timer values for the game and
players
//also make the timer controls visible
if (BoardAdmin.game1.getGameType() == Game.gameType.Timed ||
BoardAdmin.game1.getGameType() == Game.gameType.Maths)
{

    TimerLabel.Visible = true;
    Player1TimerLabel.Visible = true;
    Player2TimerLabel.Visible = true;

    BoardAdmin.game1.getPlayer1().setTimeLeft(BoardAdmin.game1.getTimerValue());
    BoardAdmin.game1.getPlayer2().setTimeLeft(BoardAdmin.game1.getTimerValue());

    Player1TimerLabel.Text =
BoardAdmin.game1.getPlayer1().getTimeLeft().ToString();
    Player2TimerLabel.Text =
BoardAdmin.game1.getPlayer2().getTimeLeft().ToString();

}

//if a maths game, set the difficulties of the players
if (BoardAdmin.game1.getGameType() == Game.gameType.Maths)
{
    BoardAdmin.game1.setDifficulty(BoardAdmin.player1Diff,
BoardAdmin.player2Diff);

    MathsScores.Visible = true;
}
else
{
    MathsScores.Visible = false;
}

UpdateViewTimer.Enabled = true;

}

//updates view of the board
public void updateView()
{

    //sets all picturebox images to empty
    for (int i = 0; i < 8; i++)

```

```
{  
    for (int j = 0; j < 8; j++)  
    {  
        squares[i, j].Image = null;  
    }  
}  
  
//sets all images of picture boxes representing a piece's position from black  
piece list  
foreach (Piece piece in pieceSets[0].getPieceSet())  
{  
    int x = piece.getPosition().X;  
    int y = piece.getPosition().Y;  
  
    squares[x, y].Image = piece.getImage();  
  
}  
//sets all images of picture boxes representing a piece's position from white  
piece list  
  
foreach (Piece piece in pieceSets[1].getPieceSet())  
{  
    int x = piece.getPosition().X;  
    int y = piece.getPosition().Y;  
  
    squares[x, y].Image = piece.getImage();  
}  
  
// if the timer values are visible then it is a timed game, perform the  
following things  
if (Player1TimerLabel.Visible == true && Player2TimerLabel.Visible == true)  
{  
    Player1TimerLabel.Text =  
BoardAdmin.game1.getPlayer1().getTimeLeft().ToString();  
    Player2TimerLabel.Text =  
BoardAdmin.game1.getPlayer2().getTimeLeft().ToString();  
  
    if (BoardAdmin.game1.getPlayer1().getTimeLeft() == 0) // if the timer falls  
to zero  
    {  
        UpdateViewTimer.Enabled = false;  
        undoMoveButton.Enabled = false;  
  
        if (BoardAdmin.game1.getPlayer1().getPlayerColour() ==  
Piece.Piececolour.BLACK)  
        {  
            BoardAdmin.game1.setGameResult(Game.result.WhiteWon);  
        }  
        else  
        {  
            BoardAdmin.game1.setGameResult(Game.result.BlackWon);  
        }  
    }  
    if (BoardAdmin.game1.getPlayer2().getTimeLeft() == 0) // if the timer falls  
to zero  
    {  
        if (BoardAdmin.game1.getPlayer2().getPlayerColour() ==  
Piece.Piececolour.BLACK)  
        {  
            BoardAdmin.game1.setGameResult(Game.result.WhiteWon);  
        }  
        else  
        {  
    }
```

```

        BoardAdmin.game1.setGameResult(Game.result.BlackWon);
    }

}

if (BoardAdmin.game1.getGameResult() == Game.result.BlackWon)
{
    foreach (PictureBox square in squares)
    {
        square.Enabled = false;
    }

    MessageBox.Show("BLACK WON! \x265A (white time out)");
}
if (BoardAdmin.game1.getGameResult() == Game.result.WhiteWon)
{
    foreach (PictureBox square in squares)
    {
        square.Enabled = false;
    }

    MessageBox.Show("WHITE WON! \x2654 (black time out)");
}

}

//updates the players timers by counting down and subtracting a 'second' from
the double value
//This update view procesure is called every second so this is why this happens
here.
//This also updates all the corresponding labels such as maths scores etc
if (BoardAdmin.game1.getGameType() == Game.gameType.Timed ||
BoardAdmin.game1.getGameType() == Game.gameType.Maths)
{
    if (BoardAdmin.game1.getTurn() ==
BoardAdmin.game1.getPlayer1().getPlayerColour())
    {
        double seconds = (BoardAdmin.game1.getPlayer1().getTimeLeft() -
Math.Floor(BoardAdmin.game1.getPlayer1().getTimeLeft()));
        double minutes =
(Math.Floor(BoardAdmin.game1.getPlayer1().getTimeLeft()));
        if (seconds == 0)
        {
            seconds = 00.60;
            minutes -= 1;
            BoardAdmin.game1.getPlayer1().setTimeLeft(Math.Round(minutes +
seconds, 2));
        }
        else
        {

BoardAdmin.game1.getPlayer1().setTimeLeft(Math.Round(BoardAdmin.game1.getPlayer1().getTimeLe
ft() - 00.01, 2));
        }
    }
    else
    {
        double seconds = (BoardAdmin.game1.getPlayer2().getTimeLeft() -
Math.Floor(BoardAdmin.game1.getPlayer2().getTimeLeft()));
        double minutes =
(Math.Floor(BoardAdmin.game1.getPlayer2().getTimeLeft()));
        if (seconds == 0)
        {
            seconds = 00.60;
            minutes -= 1;
            BoardAdmin.game1.getPlayer2().setTimeLeft(Math.Round(minutes +

```

```

seconds, 2));
}
else
{

BoardAdmin.game1.getPlayer2().setTimeLeft(Math.Round(BoardAdmin.game1.getPlayer2().getTimeLeft() - 00.01, 2));
}
}

if (BoardAdmin.game1.getGameType() == Game.gameType.Maths)
{
    MathsScoreLabel1.Text = BoardAdmin.player1Score + "/" +
BoardAdmin.NumOfQuestionsAsked1;
    MathsScoreLabel2.Text = BoardAdmin.player2Score + "/" +
BoardAdmin.NumOfQuestionsAsked2;
}

}

}

//makes a link between the picture boxes on board and a picturebox array in code
public void assignBoardsquares()
{
    string[,] boardPositions = generateBoardPositions();

    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            //assigns the control to the index in the array.
            squares[i, j] = (PictureBox)this.Controls.Find(boardPositions[i, j],
true)[0];
        }
    }
}

//generates a 2-dimensional array of board positions, type : STRING
public String[,] generateBoardPositions()
{
    String[,] positions = new string[8,8];

    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            if (i == 0)
            {
                positions[i, j] = "A" + (j + 1);
            }
            if (i == 1)
            {
                positions[i, j] = "B" + (j + 1);
            }
            if (i == 2)
            {
                positions[i, j] = "C" + (j + 1);
            }
        }
    }
}

```

```
        if (i == 3)
        {
            positions[i, j] = "D" + (j + 1);
        }
        if (i == 4)
        {
            positions[i, j] = "E" + (j + 1);
        }
        if (i == 5)
        {
            positions[i, j] = "F" + (j + 1);
        }
        if (i == 6)
        {
            positions[i, j] = "G" + (j + 1);
        }
        if (i == 7)
        {
            positions[i, j] = "H" + (j + 1);
        }
    }

}

return positions;

}

private void Form1_Load(object sender, EventArgs e)
{
    //combines all click events so they run the same procedure
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            squares[i, j].Click += new EventHandler(clickEvent);
        }
    }

    resetSquarebackgrounds();
    updateView();
}

private void UpdateViewTimer_Tick(object sender, EventArgs e)
{
    //update the turn icon to indicate who's turn it is currently
    if (BoardAdmin.game1.getTurn() ==
BoardAdmin.game1.getPlayer1().getPlayerColour())
    {
        player2TurnPictureBox.Image = null;

        if (BoardAdmin.game1.getPlayer1().getPlayerColour() ==
Piece.Piececolour.BLACK)
        {
            Player1TurnPictureBox.Image = test_chess.Properties.Resources.Chess_pdt60;
        }
        else
        {
            Player1TurnPictureBox.Image = test_chess.Properties.Resources.Chess_plt60;
        }
    }
    else
    {
```

```

        Player1TurnPicBox.Image = null;

        if (BoardAdmin.game1.getPlayer2().getPlayerColour() ==
Piece.Piececolour.BLACK)
        {
            player2TurnPicBox.Image = test_chess.Properties.Resources.Chess_pdt60;
        }
        else
        {
            player2TurnPicBox.Image = test_chess.Properties.Resources.Chess_plt60;
        }
    }

    //update view every second
    updateView();

}

//a generic click event for all picture boxes on the form
private void clickEvent(object sender, EventArgs e)
{
    currentPictureBox = (PictureBox)sender;

    resetSquarebackgrounds();

    Piece clickedPieceTemp = getPieceOnSquare(currentPictureBox);

    if (clickedPieceTemp != null)
    {
        // if the clicked piece is of the opposite colour to the current turn of the
        // game,
        // then capturing must be taking place, so call the "moving" procedure
        if (clickedPieceTemp.getColour() != BoardAdmin.game1.getTurn())
        {

            PieceMovingClickEvent(currentPictureBox);

        }
        else
        {
            // otherwise, the clicked piece is the current player's so it can be
            moved.
            standardClickEvent(currentPictureBox, clickedPieceTemp);
        }
    }
    else
    {
        // if the square that has been clicked is empty, then a piece must be being
        moved,
        // the possible moves array will be checked to see whether the clicked
        square is available to move to
        PieceMovingClickEvent(currentPictureBox);
    }

    updateView();
}

```

```

//happens when the user selects a piece to move
private void standardClickEvent(PictureBox currentPicBox, Piece clickedPieceTemp)
{
    clickedPiece = clickedPieceTemp;

    if (clickedPiece.getColour() == BoardAdmin.game1.getTurn())
    {

        possiblemoves = clickedPiece.getPossibleMoves(gameBoard);

        int nullcounter = 1;

        //makes each move in the possible moves to check whether the move would
result in being in check
        foreach (Square possiblemove in possiblemoves)
        {
            if (clickedPiece.getPosition() != possiblemove)
            {
                makeTempMove(possiblemove);

                if (possiblemove == null)
                {
                    nullcounter++;
                }
            }
        }

        // if highlighting option is on, then highlight all the possible moves in
green
        if (HighlightingON.Checked == true)
        {
            foreach (Square possiblemove in possiblemoves)
            {
                if (possiblemove != null)
                {
                    if (squares[possiblemove.X, possiblemove.Y] != null)
                    {
                        if (squares[possiblemove.X, possiblemove.Y].Tag.ToString()
== "Darkwood")
                        {
                            setDarkStyleSquares_Green(squares[possiblemove.X,
possiblemove.Y]);
                        }
                        else
                        {
                            setLightStyleSquares_Green(squares[possiblemove.X,
possiblemove.Y]);
                        }
                    }
                }
            }
        }

        // if the only possible move is the pieces current position, then do not
highlight
        //this indicates to the user that this piece cannot move anywhere.
        if (possiblemoves.Length == 1)
        {

            if ((string)currentPicBox.Tag.ToString() == "Darkwood")
            {
                setDarkStyleSquares(currentPicBox);
            }
        }
    }
}

```

```

        }
        if (currentPicBox.Tag.ToString() == "Lightwood")
        {
            setLightStyleSquares(currentPicBox);
        }
    }
else
{
    if (possiblemoves.Length > 1)
    {
        if ((string)currentPicBox.Tag.ToString() == "Darkwood")
        {
            setDarkStyleSquares_Green(currentPicBox);
        }
        if (currentPicBox.Tag.ToString() == "Lightwood")
        {
            setLightStyleSquares_Green(currentPicBox);
        }
    }
}
}

// this checks whether the current player is in the checkmate game state
private bool isCheckMated()
{
    //the clicked piece variable is going to change as a result of a temporary move
    // so it is stored temporarily to return to the current game state
    Piece tempPieceHolder = clickedPiece;
    int nullCounter = 0;      // a null counter to count all the pieces that have no
possible moves

    if (BoardAdmin.game1.getTurn() == Piece.Piececolour.WHITE)
    {
        PieceSet WhitePieces = gameBoard.getPieceSets()[1];
        foreach (Piece aPiece in WhitePieces.getPieceSet())
        {
            possiblemoves = aPiece.getPossibleMoves(gameBoard);
            clickedPiece = aPiece;

            foreach (Square possiblemove in possiblemoves)
            {
                if (possiblemove != aPiece.getPosition())
                {
                    makeTempMove(possiblemove); // makes the temporary move from
that possible move in the list, the makeTempMove algorithm will make any non-valid moves
null.
                }
                else // otherwise, make the move null as you cant move to the
current square.
                {
                    int index = Array.IndexOf(possiblemoves, possiblemove); // gets
index of the position of the possible move
                    possiblemoves[index] = null;
                }
            }
            bool a = false; // a boolean flag to store a simple result of whether
all moves are invalid.
            int nullcounter2 = 0; // a counter to count how many possible moves are
null for an individual piece
        }
    }
}

```

```

foreach (Square possiblemove in possiblemoves)
{
    if (possiblemove == null)
    {
        nullcounter2++;
    }
}
// if ALL possible moves are null then set the boolean flag to true
if (nullcounter2 == possiblemoves.Length)
{
    a = true;
}
// otherwise set the boolean flag to false
if (nullcounter2 != possiblemoves.Length)
{
    a = false;
}

if (a == true)
{
    nullCounter++;
}
//clear the possible moves array for the next piece in the for loop
Array.Clear(possiblemoves, 0, possiblemoves.Length);

}

clickedPiece = tempPieceHolder;

if (nullCounter == WhitePieces.getPieceSet().Count)
{
    return true;
}
else
{
    return false;
}
}
else // runs to check whether BLACK is in check mate then runs the same code
but for black pieces specifically.
{
    PieceSet BlackPieces = gameBoard.getPieceSets()[0];
    foreach (Piece aPiece in BlackPieces.getPieceSet())
    {
        possiblemoves = aPiece.getPossibleMoves(gameBoard);
        clickedPiece = aPiece;

        foreach (Square possiblemove in possiblemoves)
        {
            if (possiblemove != aPiece.getPosition())
            {
                makeTempMove(possiblemove);
            }
            else
            {
                int index = Array.IndexOf(possiblemoves, possiblemove);
                possiblemoves[index] = null;
            }
        }
    }
    bool a = false;
}

```

```

        int nullcounter2 = 0;

        foreach (Square possiblemove in possiblemoves)
        {
            if (possiblemove == null)
            {
                nullcounter2++;
            }
        }

        if (nullcounter2 == possiblemoves.Length)
        {
            a = true;
        }
        if (nullcounter2 != possiblemoves.Length)
        {
            a = false;
        }

        if (a == true)
        {
            nullCounter++;
        }

        Array.Clear(possiblemoves, 0, possiblemoves.Length);
    }
    clickedPiece = tempPieceHolder;
    if (nullCounter == BlackPieces.getPieceSet().Count)
    {
        return true;
    }
    else
    {
        return false;
    }
}

}

// the procedure to make a move temporarily to evaluate something then retract that
move
private void makeTempMove(Square possiblemove)
{
    BoardAdmin.TempMoving = true; //boolean variable for temp moving

    if (BoardAdmin.game1.getTurn() ==
BoardAdmin.game1.getPlayer1().getPlayerColour())
    {
        //makes temp move
        BoardAdmin.game1.getPlayer1().makeMove(possiblemove, gameBoard,
clickedPiece);

        if (BoardAdmin.game1.getPlayer1().isInCheck(gameBoard) == true) // the
isInCheck method is invoked to determine whether the player is in check
        {
            // if so, disregard the move
            int index = Array.IndexOf(possiblemoves, possiblemove);
            possiblemoves[index] = null;
        }
    }

    BoardAdmin.game1.undoLastMove();
}

```

```

        else // otherwise it is player 2's turn
        {

            BoardAdmin.game1.getPlayer2().makeMove(possiblemove, gameBoard,
clickedPiece);

            if (BoardAdmin.game1.getPlayer2().isInCheck(gameBoard) == true)
            {
                int index = Array.IndexOf(possiblemoves, possiblemove);
                possiblemoves[index] = null;
            }

            BoardAdmin.game1.undoLastMove();

        }

        BoardAdmin.TempMoving = false;
    }

    // event that is run when a piece is being moved
    private void PieceMovingClickEvent(PictureBox currentPicBox)
    {
        Square clickedSquare = new Square();

        // gets the square that was clicked by mapping the picture box array index onto
        // the square array.
        for (int i = 0; i < 8; i++)
        {
            for (int j = 0; j < 8; j++)
            {
                if (squares[i, j] == currentPicBox)
                {
                    clickedSquare = gameBoard.getSquares()[i, j];
                }
            }
        }

        if (possiblemoves != null)
        {
            // if the clicked square is one of the possible moves then make the move
            if (possiblemoves.Contains(clickedSquare) == true)
            {

                if (BoardAdmin.game1.getPlayer1().getPlayerColour() ==
BoardAdmin.game1.getTurn())
                {

                    BoardAdmin.game1.getPlayer1().makeMove(clickedSquare,
gameBoard, clickedPiece);

                }
                if (BoardAdmin.game1.getPlayer2().getPlayerColour() ==
BoardAdmin.game1.getTurn())
                {

                    BoardAdmin.game1.getPlayer2().makeMove(clickedSquare, gameBoard,
clickedPiece);

                }
            }
        }
    }
}

```

```

// update the move history text box with the latest move
updateUnicodeMoveBox(BoardAdmin.game1.getMoveHistory().Last<Move>());

if (BoardAdmin.game1.getTurn() == Piece.Piececolour.WHITE)
{
    BoardAdmin.game1.setTurn(Piece.Piececolour.BLACK);
}
else
{
    BoardAdmin.game1.setTurn(Piece.Piececolour.WHITE);
}

//This is where the maths question is called - a check is needed to
determine that the click event
//corresponds to a moving piece of the current player.
if (clickedPiece != null)
{
    if (clickedPiece.getColour() != BoardAdmin.game1.getTurn())
    {
        if (BoardAdmin.game1.getGameType() == Game.gameType.Maths)
            // if it is a maths game then a question must be presented
to the player
    {
        AskQuestion aAskQuestionForm = new AskQuestion();

        UpdateViewTimer.Enabled = false;

        if (aAskQuestionForm.ShowDialog() !=
System.Windows.Forms.DialogResult.OK)
        {
            //Waits until the question form has been dealt with

        }
        UpdateViewTimer.Enabled = true;
    }
}
}

clickedPiece = null;
Array.Clear(possiblemoves, 0, possiblemoves.Length);

// checks whether this move has resulted in check mate
if (isCheckMated() == true)
{
    UpdateViewTimer.Enabled = false;

    foreach (PictureBox square in squares)
    {
        square.Enabled = false;
    }
    string TempText = UnicodeMoveBox.Text;
    UnicodeMoveBox.Clear();

    // updates the unicode move box with the game result
    if (BoardAdmin.game1.getTurn() == Piece.Piececolour.BLACK)
    {
        BoardAdmin.game1.setGameResult(Game.result.WhiteWon);
        MessageBox.Show("WHITE WON! \x2654 (CheckMate)");
        UnicodeMoveBox.AppendText("WHITE WON! \x2654 (CheckMate)" +
Environment.NewLine + TempText);
    }
    else
    {
        BoardAdmin.game1.setGameResult(Game.result.BlackWon);
    }
}

```

```

        MessageBox.Show("BLACK WON! \x265A (CheckMate)");
        UnicodeMoveBox.AppendText("BLACK WON! \x265A (CheckMate)" +
Environment.NewLine + TempText);

    }
    // disable the undo move button as the game has finished
    // the user may still look through the list of moves made in the
text box
    undoMoveButton.Enabled = false;

}
}

// finds the piece that is on the picturebox that has been clicked
private Piece getPieceOnSquare(PictureBox clickedPicBox)
{
    // the coordinates of the clicked picture box will correspond in parallel to the
cooridnates of the square object.
    int x = 0;
    int y = 0;
    Piece clickedPiece = null;

    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            if (squares[i, j] == clickedPicBox)
            {
                x = i;
                y = j;
            }
        }
    }

    //scans the piece sets to find the piece that was clicked
    foreach (Piece piece in pieceSets[0].getPieceSet())
    {
        if (piece.getPosition().X == x && piece.getPosition().Y == y)
        {
            clickedPiece = piece;
        }
    }
    foreach (Piece piece in pieceSets[1].getPieceSet())
    {
        if (piece.getPosition().X == x && piece.getPosition().Y == y)
        {
            clickedPiece = piece;
        }
    }
}

return clickedPiece; // return the piece that is occupying the square on the
board
}

//resets the backgrounds of the pictureboxes
private void resetSquarebackgrounds()
{
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)

```

```

    {
        if(squares[i,j].Tag.ToString() == "Darkwood")
        {
            setDarkStyleSquares(squares[i, j]);
        }
        if (squares[i, j].Tag.ToString() == "Lightwood")
        {
            setLightStyleSquares(squares[i, j]);
        }
    }
}

// this is needed to change the highlighting of squares dynamically when the
highlighting toggle is switched on and off
private void ChangeStaticHighlighting()
{
    if (possiblemoves != null)
    {

        foreach (Square possiblemove in possiblemoves)
        {
            if (possiblemove != null)
            {
                if (squares[possiblemove.X, possiblemove.Y] != null)
                {
                    if (squares[possiblemove.X, possiblemove.Y].Tag.ToString() ==
"Darkwood")
                    {
                        if (HighlightingOFF.Checked == true)
                        {
                            // if highlighting is off, but the piece can still move
then highlight the piece square only.
                            if (squares[possiblemove.X, possiblemove.Y] ==
currentPicBox)
                            {
                                setDarkStyleSquares_Green(squares[possiblemove.X,
possiblemove.Y]);
                            }
                            else
                            {
                                setDarkStyleSquares(squares[possiblemove.X,
possiblemove.Y]);
                            }
                        }
                        else
                        {
                            // otherwise highlight all possible move squares in
green
                            setDarkStyleSquares_Green(squares[possiblemove.X,
possiblemove.Y]);
                        }
                    }
                    else // otherwise it is light squares
                    {
                        if (HighlightingOFF.Checked == true)
                        {
                            if (squares[possiblemove.X, possiblemove.Y] ==
currentPicBox)
                            {
                                setLightStyleSquares_Green(squares[possiblemove.X,
possiblemove.Y]);
                            }
                            else
                        }
                    }
                }
            }
        }
    }
}

```

```

        setLightStyleSquares(squares[possiblemove.X,
possiblemove.Y]);
    }
}
else
{
}

setLightStyleSquares_Green(squares[possiblemove.X,possiblemove.Y]);
}
}

}

}

}

if (possiblemoves.Length == 1) // if the possible moves array only contains
the occupying square then do not highlight.
{
    if ((string)currentPicBox.Tag.ToString() == "Darkwood")
    {
        setDarkStyleSquares(currentPicBox);
    }
    if (currentPicBox.Tag.ToString() == "Lightwood")
    {
        setLightStyleSquares(currentPicBox);
    }
}
}

// toggling of the highlighting option on the form
private void HighlightingON_CheckedChanged(object sender, EventArgs e)
{
    ChangeStaticHighlighting();
}

private void HighlightingOFF_CheckedChanged(object sender, EventArgs e)
{
    ChangeStaticHighlighting();
}

// this procedure changes the image of the current picture box to one without green
highlighting
private void setDarkStyleSquares(PictureBox aPicBox)
{
    if (BoardAdmin.gameStyle == "WOOD")
    {
        aPicBox.BackgroundImage = test_chess.Properties.Resources.DarkWoodSquare;
    }
    if (BoardAdmin.gameStyle == "MARBLE")
    {
        aPicBox.BackgroundImage = test_chess.Properties.Resources.BlueMarble;
    }
    if (BoardAdmin.gameStyle == "PLAINBLUE")
    {
        aPicBox.BackgroundImage = test_chess.Properties.Resources.DarkBlue;
    }
    if (BoardAdmin.gameStyle == "PLAINPURPLE")
    {
        aPicBox.BackgroundImage = test_chess.Properties.Resources.DarkPurple;
    }
    if (BoardAdmin.gameStyle == "PLAINBROWN")
    {

```

```

        aPicBox.BackgroundImage = test_chess.Properties.Resources.DarkBrown;
    }
    if (BoardAdmin.gameStyle == "PLAINRED")
    {
        aPicBox.BackgroundImage = test_chess.Properties.Resources.DarkRed;
    }
}

// this procedure changes the image of the current picture box to one without green
highlighting
private void setLightStyleSquares(PictureBox aPicBox)
{
    if (BoardAdmin.gameStyle == "WOOD")
    {
        aPicBox.BackgroundImage = test_chess.Properties.Resources.LightWoodSquare;
    }
    if (BoardAdmin.gameStyle == "MARBLE")
    {
        aPicBox.BackgroundImage = test_chess.Properties.Resources.WhiteMarble;
    }
    if (BoardAdmin.gameStyle == "PLAINBLUE")
    {
        aPicBox.BackgroundImage = test_chess.Properties.Resources.LightBlue;
    }
    if (BoardAdmin.gameStyle == "PLAINPURPLE")
    {
        aPicBox.BackgroundImage = test_chess.Properties.Resources.LightPurple;
    }
    if (BoardAdmin.gameStyle == "PLAINBROWN")
    {
        aPicBox.BackgroundImage = test_chess.Properties.Resources.LightBrown;
    }
    if (BoardAdmin.gameStyle == "PLAINRED")
    {
        aPicBox.BackgroundImage = test_chess.Properties.Resources.LightRed;
    }
}
// this procedure changes the image of the current picture box to one *with* green
highlighting
private void setDarkStyleSquares_Green(PictureBox aPicBox)
{
    if (BoardAdmin.gameStyle == "WOOD")
    {
        aPicBox.BackgroundImage =
test_chess.Properties.Resources.DarkWoodSquare_greenBorder;
    }
    if (BoardAdmin.gameStyle == "MARBLE")
    {
        aPicBox.BackgroundImage =
test_chess.Properties.Resources.BlueMarble_GreenBorder;
    }
    if (BoardAdmin.gameStyle == "PLAINBLUE")
    {
        aPicBox.BackgroundImage =
test_chess.Properties.Resources.DarkBlue_GreenBorder;
    }
    if (BoardAdmin.gameStyle == "PLAINPURPLE")
    {
        aPicBox.BackgroundImage =
test_chess.Properties.Resources.DarkPurple_GreenBorder;
    }
    if (BoardAdmin.gameStyle == "PLAINBROWN")
    {
        aPicBox.BackgroundImage =
test_chess.Properties.Resources.DarkBrown_GreenBorder;
    }
}

```

```

        }
        if (BoardAdmin.gameStyle == "PLAINRED")
        {
            aPicBox.BackgroundImage =
test_chess.Properties.Resources.DarkRed_GreenBorder;
        }
    }
    // this procedure changes the image of the current picture box to one *with* green
highlighting
private void setLightStyleSquares_Green(PictureBox aPicBox)
{
    if (BoardAdmin.gameStyle == "WOOD")
    {
        aPicBox.BackgroundImage =
test_chess.Properties.Resources.LightWoodSquare_greenBorder;
    }
    if (BoardAdmin.gameStyle == "MARBLE")
    {
        aPicBox.BackgroundImage =
test_chess.Properties.Resources.WhiteMarble_GreenBorder;
    }
    if (BoardAdmin.gameStyle == "PLAINBLUE")
    {
        aPicBox.BackgroundImage =
test_chess.Properties.Resources.LightBlue_GreenBorder;
    }
    if (BoardAdmin.gameStyle == "PLAINPURPLE")
    {
        aPicBox.BackgroundImage =
test_chess.Properties.Resources.LightPurple_GreenBorder;
    }
    if (BoardAdmin.gameStyle == "PLAINBROWN")
    {
        aPicBox.BackgroundImage =
test_chess.Properties.Resources.LightBrown_GreenBorder;
    }
    if (BoardAdmin.gameStyle == "PLAINRED")
    {
        aPicBox.BackgroundImage =
test_chess.Properties.Resources.LightRed_GreenBorder;
    }
}

// updates the textbox containing the list of moves made in unicode format
// uses unicode hexadecimal codes for chess piece icons
// the move object from the game's history is used as a reference
private void updateUnicodeMoveBox(Move aMove)
{
    string tempText = UnicodeMoveBox.Text;
    UnicodeMoveBox.Clear();

    UnicodeMoveBox.AppendText( BoardAdmin.game1.getMoveHistory().Count + ". ");

    if (aMove.PieceMoved.getType() == Piece.pieceType.BISHOP)
    {
        if (BoardAdmin.game1.getTurn() == Piece.Piececolour.BLACK)
        {
            UnicodeMoveBox.AppendText("\u265D");
        }
        else
        {
            UnicodeMoveBox.AppendText("\u2657");
        }
    }
    if (aMove.PieceMoved.getType() == Piece.pieceType.KING)

```

```

{
    if (BoardAdmin.game1.getTurn() == Piece.Piececolour.BLACK)
    {
        UnicodeMoveBox.AppendText("\u265A");
    }
    else
    {
        UnicodeMoveBox.AppendText("\u2654");
    }
}
if (aMove.PieceMoved.getType() == Piece.pieceType.KNIGHT)
{
    if (BoardAdmin.game1.getTurn() == Piece.Piececolour.BLACK)
    {
        UnicodeMoveBox.AppendText("\u265E");
    }
    else
    {
        UnicodeMoveBox.AppendText("\u2658");
    }
}
if (aMove.PieceMoved.getType() == Piece.pieceType.PAWN)
{
    if (BoardAdmin.game1.getTurn() == Piece.Piececolour.BLACK)
    {
        UnicodeMoveBox.AppendText("\u265F");
    }
    else
    {
        UnicodeMoveBox.AppendText("\u2659");
    }
}
if (aMove.PieceMoved.getType() == Piece.pieceType.QUEEN)
{
    if (BoardAdmin.game1.getTurn() == Piece.Piececolour.BLACK)
    {
        UnicodeMoveBox.AppendText("\u265B");
    }
    else
    {
        UnicodeMoveBox.AppendText("\u2655");
    }
}
if (aMove.PieceMoved.getType() == Piece.pieceType.ROOK)
{
    if (BoardAdmin.game1.getTurn() == Piece.Piececolour.BLACK)
    {
        UnicodeMoveBox.AppendText("\u265C");
    }
    else
    {
        UnicodeMoveBox.AppendText("\u2656");
    }
}

string initialPosition =
squares[aMove.initialPosition.X,aMove.initialPosition.Y].Name;
string finalPosition =
squares[aMove.finalPosition.X,aMove.finalPosition.Y].Name;

UnicodeMoveBox.AppendText(" " + initialPosition + "\u02c3\u02c3 " +
finalPosition);

```

```
if (aMove.pieceCaptured != null)
{
    UnicodeMoveBox.AppendText(Environment.NewLine + "Captured: ");

    if (aMove.pieceCaptured.getType() == Piece.pieceType.BISHOP)
    {
        if (aMove.pieceCaptured.getColour() == Piece.Piececolour.BLACK)
        {
            UnicodeMoveBox.AppendText("\u265D");
        }
        else
        {
            UnicodeMoveBox.AppendText("\u2657");
        }
    }
    if (aMove.pieceCaptured.getType() == Piece.pieceType.KING)
    {
        if (aMove.pieceCaptured.getColour() == Piece.Piececolour.BLACK)
        {
            UnicodeMoveBox.AppendText("\u265A");
        }
        else
        {
            UnicodeMoveBox.AppendText("\u2654");
        }
    }
    if (aMove.pieceCaptured.getType() == Piece.pieceType.KNIGHT)
    {
        if (aMove.pieceCaptured.getColour() == Piece.Piececolour.BLACK)
        {
            UnicodeMoveBox.AppendText("\u265E");
        }
        else
        {
            UnicodeMoveBox.AppendText("\u2658");
        }
    }
    if (aMove.pieceCaptured.getType() == Piece.pieceType.PAWN)
    {
        if (aMove.pieceCaptured.getColour() == Piece.Piececolour.BLACK)
        {
            UnicodeMoveBox.AppendText("\u265F");
        }
        else
        {
            UnicodeMoveBox.AppendText("\u2659");
        }
    }
    if (aMove.pieceCaptured.getType() == Piece.pieceType.QUEEN)
    {
        if (aMove.pieceCaptured.getColour() == Piece.Piececolour.BLACK)
        {
            UnicodeMoveBox.AppendText("\u265B");
        }
        else
        {
            UnicodeMoveBox.AppendText("\u2655");
        }
    }
    if (aMove.pieceCaptured.getType() == Piece.pieceType.ROOK)
    {
        if (aMove.pieceCaptured.getColour() == Piece.Piececolour.BLACK)
        {
            UnicodeMoveBox.AppendText("\u265C");
        }
    }
}
```

```

        else
        {
            UnicodeMoveBox.AppendText("\x2656");
        }
    }

    UnicodeMoveBox.AppendText(Environment.NewLine);
    UnicodeMoveBox.AppendText(tempText);
}

// calls the undo procedure from the game object
private void undoLastMove()
{
    //System.Windows.Forms.MessageBox.Show("TEST");
    BoardAdmin.game1.undoLastMove();

    updateUnicodeText_afterRemoval();
}

// when a
private void updateUnicodeText_afterRemoval()
{
    if (UnicodeMoveBox.Enabled == true)
    {
        // if the contents is not empty
        if (UnicodeMoveBox.Text != "")
        {
            // remove the first line ( the move that has been undone )
            UnicodeMoveBox.Text = UnicodeMoveBox.Text.Remove(0,
UnicodeMoveBox.Lines[0].Length);

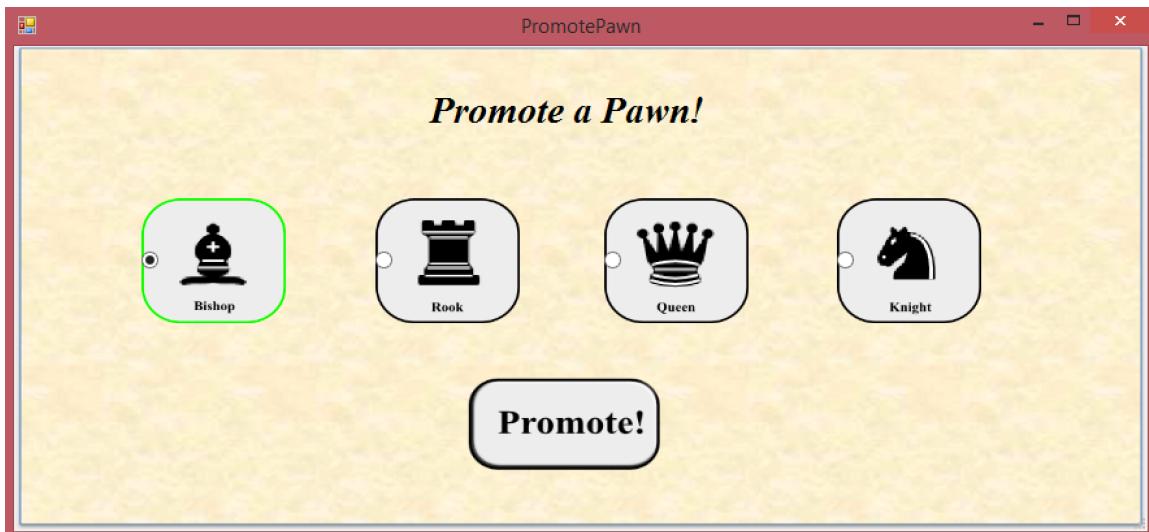
            // store the remaining text temporarily
            string tempText = UnicodeMoveBox.Text;
            tempText = tempText.Trim();      // trim any preceding white trailing
spaces
            UnicodeMoveBox.Text = (tempText); // replace the text without the blank
line

            if (tempText.StartsWith("C")) // if a piece was captured then the line
will start with a C
            {
                updateUnicodeText_afterRemoval(); // run the procedure again,
recursively, to remove this line also
            }
        }
    }
}

private void undoMoveButton_Click(object sender, EventArgs e)
{
    //System.Windows.Forms.MessageBox.Show("TEST3");
    undoLastMove();
}
}
}

```

## PromotePawn



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace test_chess
{
    public partial class PromotePawn : Form
    {
        public PromotePawn()
        {
            InitializeComponent();
        }

        // changes the buttons to those with a green border to confirm selection when
        // clicked
        private void KnightButton_CheckedChanged(object sender, EventArgs e)
        {
            if (KnightButton.Checked == true)
            {
                KnightButton.BackgroundImage =
test_chess.Properties.Resources.KnightButton_GreenBorder;
            }
            else
            {
                KnightButton.BackgroundImage = test_chess.Properties.Resources.KnightButton;
            }
        }

        private void QueenButton_CheckedChanged(object sender, EventArgs e)
        {
            if (QueenButton.Checked == true)
```

```
        {
            QueenButton.BackgroundImage =
test_chess.Properties.Resources.QueenButton_GreenBorder;
        }
        else
        {
            QueenButton.BackgroundImage = test_chess.Properties.Resources.QueenButton ;
        }
    }

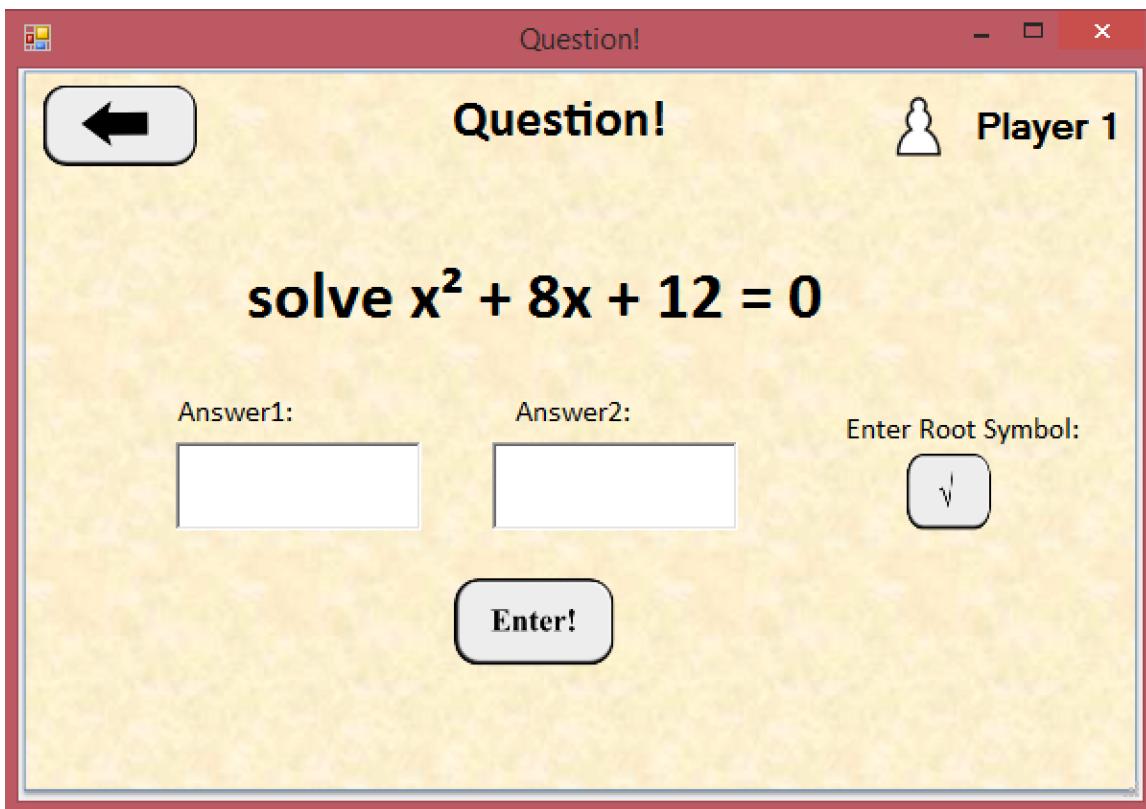
private void RookButton_CheckedChanged(object sender, EventArgs e)
{
    if (RookButton.Checked == true)
    {
        RookButton.BackgroundImage =
test_chess.Properties.Resources.RookButton_GreenBorder;
    }
    else
    {
        RookButton.BackgroundImage = test_chess.Properties.Resources.RookButton;
    }
}

private void BishopButton_CheckedChanged(object sender, EventArgs e)
{
    if (BishopButton.Checked == true)
    {
        BishopButton.BackgroundImage =
test_chess.Properties.Resources.BishopButton_GreenBorder;
    }
    else
    {
        BishopButton.BackgroundImage = test_chess.Properties.Resources.BishopButton;
    }
}

// when the promote button is clicked the user choice is stored in the global
variable so the game board can process this move.
private void PromoteButton_Click(object sender, EventArgs e)
{
    if (RookButton.Checked == true)
    {
        BoardAdmin.promotingPawnType = (Piece.pieceType.ROOK);
    }
    if (BishopButton.Checked == true)
    {
        BoardAdmin.promotingPawnType = (Piece.pieceType.BISHOP);
    }
    if (KnightButton.Checked == true)
    {
        BoardAdmin.promotingPawnType = (Piece.pieceType.KNIGHT);
    }
    if (QueenButton.Checked == true)
    {
        BoardAdmin.promotingPawnType = (Piece.pieceType.QUEEN);
    }

    this.Close();
}
```

## AskQuestion



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO;

namespace test_chess
{
    public partial class AskQuestion : Form
    {
        // a form which presents a question for the player to answer.
        public AskQuestion()
        {
            InitializeComponent();
            gatherQuestion(); // gathers the question when the form is loaded
        }

        public void gatherQuestion()
        {
            Random RND = new Random(); // a random number generator to generate a random
index to retrieve a question from the list
```

```

// creates a second list of questions to store all the questions with the
correct difficulty of the current player
List<QuestionBank> questions2 = new List<QuestionBank>();

//resets form properties
enterButton.Enabled = true;
userAnswer1.Enabled = true;
userAnswer2.Enabled = true;
SquareRootButton.Enabled = true;
BackButton.Enabled = false;
userAnswer1.Clear();
userAnswer2.Clear();
QuestionLabel.ForeColor = Color.Black;
userAnswer1.ForeColor = Color.Black;
userAnswer2.ForeColor = Color.Black;

int targetDifficulty; // the difficulty that the question must be

// retrieves the player difficulty level
if (BoardAdmin.game1.getTurn() ==
BoardAdmin.game1.getPlayer1().getPlayerColour())
{
    targetDifficulty = BoardAdmin.game1.getPlayer2Difficulty();
}
else
{
    targetDifficulty = BoardAdmin.game1.getPlayer1Difficulty();
}

// stores all questions with the target difficulty in the secondary list
foreach (QuestionBank aQuestion in BoardAdmin.questions)
{
    if (aQuestion.difficulty == targetDifficulty)
    {
        questions2.Add(aQuestion);
    }
}

//randomly generates a number to use as an index to retrieve a question, from
the list of question objects
int index = RND.Next(0, questions2.Count);
BoardAdmin.currentQuestion = questions2[index];

QuestionLabel.Text = (BoardAdmin.currentQuestion.questiontext);

//if the question does not have a second answer, then do not show the second
textbox
if (BoardAdmin.currentQuestion.answer2 != "")
{
    userAnswer2.Visible = true;
    Answer2Label.Visible = true;
}
else
{
    userAnswer2.Visible = false;
    Answer2Label.Visible = false;
}

if (BoardAdmin.game1.getTurn() ==
BoardAdmin.game1.getPlayer1().getPlayerColour())
{
    //increment the total number of questions asked for purposes of the UI
}

```

```

BoardAdmin.NumOfQuestionsAsked2++;

PlayerLabel.Text = "Player 2";
//update the piece icon to show that it is player 2's question
if (BoardAdmin.game1.getPlayer2().getPlayerColour() ==
Piece.Piececolour.BLACK)
{
    PlayerTurnPicBox.Image = test_chess.Properties.Resources.Chess_pdt60;
}
else
{
    PlayerTurnPicBox.Image = test_chess.Properties.Resources.Chess_plt60;
}
}
else
{
    BoardAdmin.NumOfQuestionsAsked1++; // increment the number of questions
asked to player 1

    PlayerLabel.Text = "Player 1";
    //update the piece icon to show that it is player 1's question
    if (BoardAdmin.game1.getPlayer1().getPlayerColour() ==
Piece.Piececolour.BLACK)
    {
        PlayerTurnPicBox.Image = test_chess.Properties.Resources.Chess_pdt60;
    }
    else
    {
        PlayerTurnPicBox.Image = test_chess.Properties.Resources.Chess_plt60;
    }
}

}

private void enterButton_Click(object sender, EventArgs e)
{
    if (BoardAdmin.currentQuestion.answer2 == "")
    {
        if (userAnswer1.Text.Trim() == BoardAdmin.currentQuestion.answer) // trims
any white spaces
        {
            if (BoardAdmin.game1.getTurn() ==
BoardAdmin.game1.getPlayer1().getPlayerColour())
            {
                BoardAdmin.player2Score++; // increment score
            }
            else
            {
                BoardAdmin.player1Score++;
            }
            QuestionLabel.ForeColor = Color.Green;
            QuestionLabel.Text = "CORRECT!";
        }

        // clears answer box to the show solution
        userAnswer1.Clear();
        userAnswer1.ForeColor = Color.Green;
        userAnswer1.Text = "The answer was: " +
BoardAdmin.currentQuestion.answer;
    }
}

```

```

        }

        else
        {
            QuestionLabel.ForeColor = Color.Red;
            QuestionLabel.Text = "INCORRECT!";

            subtractTime();

            userAnswer1.Clear();
            userAnswer1.ForeColor = Color.Red;
            userAnswer1.Text = "The answer was: " +
BoardAdmin.currentQuestion.answer;
        }

    }
    else
    {
        //checks whether EITHER answer box contains the correct answers, there are
        two answers and the user could have
        //put them in any order, so check for both.
        if (userAnswer1.Text.Trim() == BoardAdmin.currentQuestion.answer ||
userAnswer2.Text.Trim() == BoardAdmin.currentQuestion.answer)
        {
            if (userAnswer1.Text.Trim() == BoardAdmin.currentQuestion.answer2 ||
userAnswer2.Text.Trim() == BoardAdmin.currentQuestion.answer2)
            {
                if (BoardAdmin.game1.getTurn() ==
BoardAdmin.game1.getPlayer1().getPlayerColour())
                {
                    BoardAdmin.player2Score++;
                }
                else
                {
                    BoardAdmin.player1Score++;
                }

                QuestionLabel.ForeColor = Color.Green;
                QuestionLabel.Text = "CORRECT!";
            }
        }
        else
        {
            QuestionLabel.ForeColor = Color.Red;
            QuestionLabel.Text = "INCORRECT!";

            subtractTime();

            userAnswer1.Clear();
            userAnswer1.ForeColor = Color.Red;
            userAnswer1.Text = "The answer was: " +
BoardAdmin.currentQuestion.answer;
        }
    }
}
else
{
    QuestionLabel.ForeColor = Color.Red;
    QuestionLabel.Text = "INCORRECT!";
}

```

```

        subtractTime();

        userAnswer1.Clear();
        userAnswer1.ForeColor = Color.Red;
        userAnswer1.Text = "The answer was: " +
BoardAdmin.currentQuestion.answer;

    }

    userAnswer1.Clear();
    userAnswer1.ForeColor = Color.Green;
    userAnswer1.Text = "The answer was: " + BoardAdmin.currentQuestion.answer;

    userAnswer2.Clear();
    userAnswer2.ForeColor = Color.Green;
    userAnswer2.Text = "The answer was: " + BoardAdmin.currentQuestion.answer2;

}

// sets properties to false so no more changes can be made, back button is now
available.
enterButton.Enabled = false;
userAnswer1.Enabled = false;
userAnswer2.Enabled = false;
SquareRootButton.Enabled = false;
BackButton.Enabled = true;
}

//This procedure adjusts the timeLeft property for the player who has just ended
their turn.
//This uses a double data type but ensures that the value shown is a time value by
making
//the value start counting down at 60 rather than 99
private void subtractTime()
{
    if (BoardAdmin.game1.getTurn() ==
BoardAdmin.game1.getPlayer2().getPlayerColour())
    {
        if (BoardAdmin.game1.getPlayer1Difficulty() == 1)
        {

BoardAdmin.game1.getPlayer1().setTimeLeft(BoardAdmin.game1.getPlayer1().getTimeLeft() -
0.30);
        }
        if (BoardAdmin.game1.getPlayer1Difficulty() == 2)
        {

BoardAdmin.game1.getPlayer1().setTimeLeft(BoardAdmin.game1.getPlayer1().getTimeLeft() -
0.20);
        }
        if (BoardAdmin.game1.getPlayer1Difficulty() == 3)
        {

BoardAdmin.game1.getPlayer1().setTimeLeft(BoardAdmin.game1.getPlayer1().getTimeLeft() -
0.10);
        }

        double seconds = (BoardAdmin.game1.getPlayer1().getTimeLeft() -
Math.Floor(BoardAdmin.game1.getPlayer1().getTimeLeft()));
        double minutes = (Math.Floor(BoardAdmin.game1.getPlayer1().getTimeLeft()));
        if (seconds > 0.60)
    }
}

```

```

        {
            double difference = 1 - seconds; // the difference needs to be
            subtracted so the timer counts down from 0.60 (i.e 60 seconds)
            seconds = 00.60 - difference;

            BoardAdmin.game1.getPlayer1().setTimeLeft(Math.Round(minutes + seconds,
2));
        }
    }
    //same again but for player 1
    if (BoardAdmin.game1.getTurn() ==
BoardAdmin.game1.getPlayer1().getPlayerColour())
{
    if (BoardAdmin.game1.getPlayer2Difficulty() == 1)
{

BoardAdmin.game1.getPlayer2().setTimeLeft(BoardAdmin.game1.getPlayer2().getTimeLeft() -
0.30);
}
    if (BoardAdmin.game1.getPlayer2Difficulty() == 2)
{

BoardAdmin.game1.getPlayer2().setTimeLeft(BoardAdmin.game1.getPlayer2().getTimeLeft() -
0.20);
}
    if (BoardAdmin.game1.getPlayer2Difficulty() == 3)
{

BoardAdmin.game1.getPlayer2().setTimeLeft(BoardAdmin.game1.getPlayer2().getTimeLeft() -
0.10);
}

        double seconds = (BoardAdmin.game1.getPlayer2().getTimeLeft() -
Math.Floor(BoardAdmin.game1.getPlayer2().getTimeLeft()));
        double minutes = (Math.Floor(BoardAdmin.game1.getPlayer2().getTimeLeft()));
        if (seconds > 0.60)
{
            double difference = 1 - seconds;
            seconds = 00.60 - difference;

            BoardAdmin.game1.getPlayer2().setTimeLeft(Math.Round(minutes + seconds,
2));
}
}

//appends square root symbol when button is clicked
private void SquareRootButton_Click(object sender, EventArgs e)
{
    userAnswer1.AppendText("√");
}

private void BackButton_Click(object sender, EventArgs e)
{
    this.Close();
}

private void AskQuestion_FormClosing(object sender, EventArgs e)
{
    subtractTime();
}

private void AskQuestion_Load(object sender, EventArgs e)
{
}

```

```
}

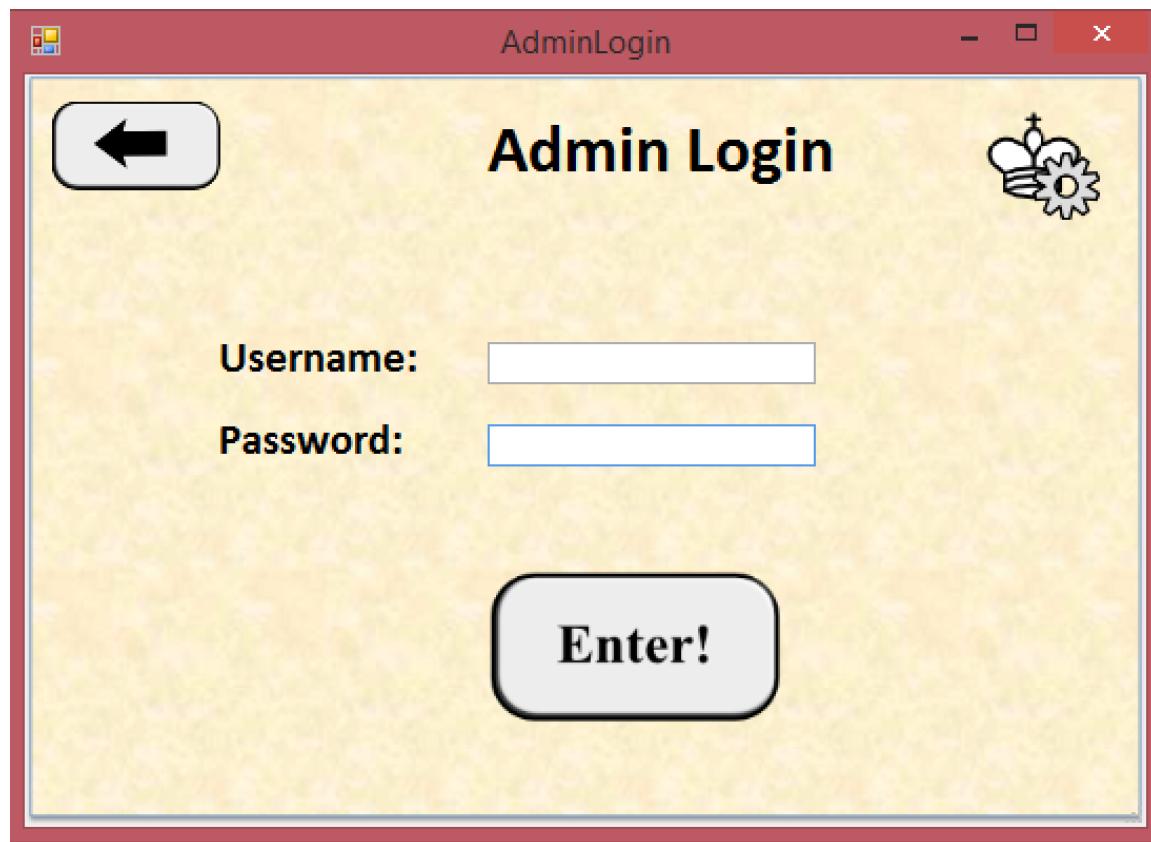
//if the player closes the form before the question has been answered, subtract time.
private void AskQuestion_FormClosing(object sender, FormClosingEventArgs e)
{
    if (BackButton.Enabled != true)
    {
        subtractTime();
    }
}

private void RootSymbolLabel_Click(object sender, EventArgs e)
{

}

}
```

## AdminLogin



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace test_chess
{
    public partial class AdminLogin : Form
    {

        public AdminLogin()
        {
            InitializeComponent();
            //this.KeyDown += new KeyEventHandler(KeyDown);
        }

        private void BackButton_Click(object sender, EventArgs e)
        {
            this.Close();
        }
    }
}
```

```
        }

    private void enterButton_Click(object sender, EventArgs e)
    {
        enterEvent();
    }

    private void checkEnter(object sender, System.Windows.Forms.KeyEventArgs e)
    {

    }

    private void enterEvent()
    {
        //checks the username and password combination
        if (userNameBox.Text.ToUpper() == "CHESSCLUBADMIN" && passwordBox.Text.ToUpper()
== "GREENROOK")
        {
            this.Close();

            AdminPage aAdminPage = new AdminPage(); // create the admin page form

            if (aAdminPage.ShowDialog() != System.Windows.Forms.DialogResult.OK)
            {
                //Waits until the question form has been dealt with
            }
        }
        else
        {
            System.Windows.Forms.MessageBox.Show("INVALID username or password
combination");
        }
    }

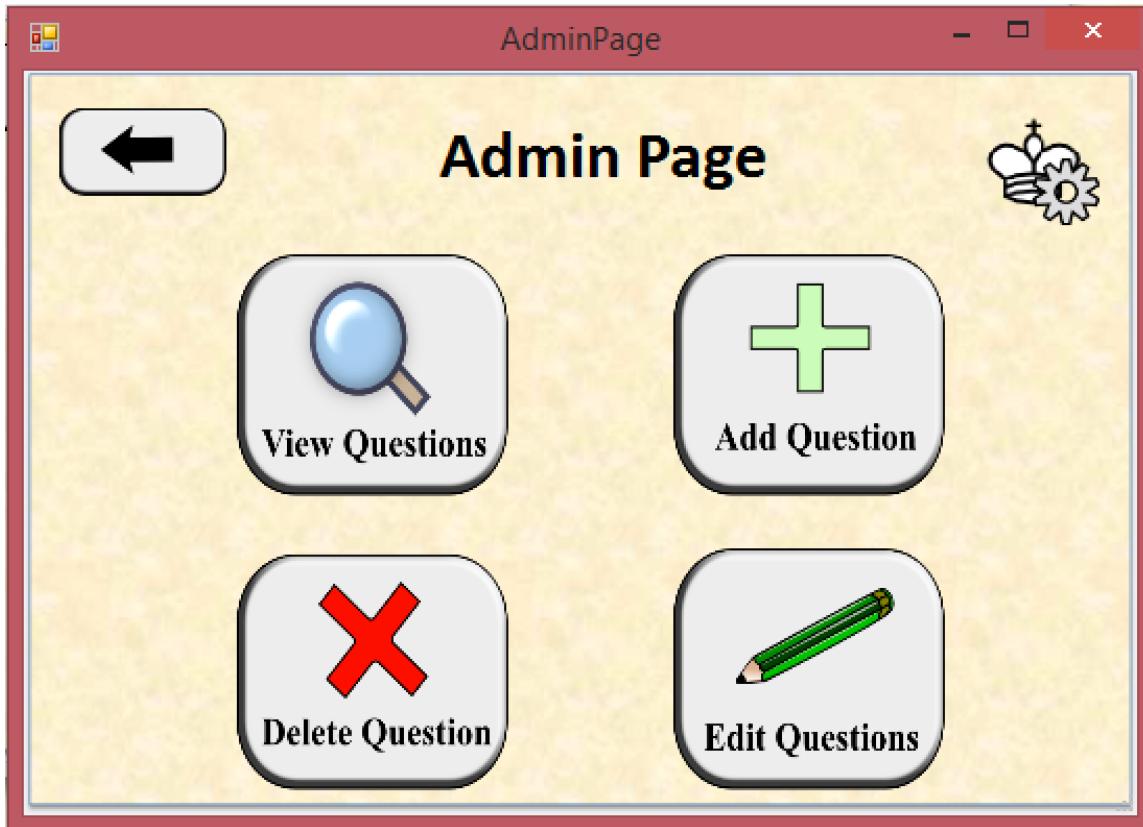
    private void AdminLogin_KeyPress(object sender, KeyPressEventArgs e) // when the
enter button is pressed
{
    MessageBox.Show("TEST");
    if (e.KeyChar == (char)13)
    {
        enterEvent();
    }
}

private void AdminLogin_KeyDown(object sender, KeyEventArgs e) // when the enter
button is pressed
{
    MessageBox.Show("TEST");

    if (e.KeyCode == Keys.Return)
    {
        enterEvent();
    }
}

}
```

## AdminPage



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace test_chess
{
    public partial class AdminPage : Form
    {
        public AdminPage()
        {
            InitializeComponent();
            this.BringToFront(); // brings the form to the front so it is in view.
        }

        // when the view questions button is clicked, the viewQuestions form is loaded.
        private void ViewQuestionsButton_Click(object sender, EventArgs e)
        {
            ViewQuestions aViewQuestionsForm = new ViewQuestions();

            if (aViewQuestionsForm.ShowDialog() != System.Windows.Forms.DialogResult.OK)
            {
                //Waits until the question form has been dealt with
            }
        }
    }
}
```

```
        }
    }
// when the delete questions button is clicked, the deleteQuestions form is loaded.
private void DeleteQuestionButton_Click(object sender, EventArgs e)
{
    DeleteQuestion aDeleteQuestionForm = new DeleteQuestion();

    if (aDeleteQuestionForm.ShowDialog() != System.Windows.Forms.DialogResult.OK)
    {
        //Waits until the question form has been dealt with
    }
}
// when the edit questions button is clicked, the editQuestions form is loaded.
private void EditQuestionsButton_Click(object sender, EventArgs e)
{
    EditQuestion aEditQuestionForm = new EditQuestion();

    if (aEditQuestionForm.ShowDialog() != System.Windows.Forms.DialogResult.OK)
    {
        //Waits until the question form has been dealt with
    }
}
// when the add questions button is clicked, the addQuestions form is loaded.
private void AddQuestionButton_Click(object sender, EventArgs e)
{
    AddQuestion aAddQuestion = new AddQuestion();

    if (aAddQuestion.ShowDialog() != System.Windows.Forms.DialogResult.OK)
    {
        //Waits until the question form has been dealt with
    }
}

// the following are events when the mouse enters and leaves each button, changing
the cursor.
private void ViewQuestionsButton_MouseEnter(object sender, EventArgs e)
{
    Cursor = Cursors.Hand;
}

private void ViewQuestionsButton_MouseLeave(object sender, EventArgs e)
{
    Cursor = Cursors.Default;
}

private void AddQuestionButton_MouseEnter(object sender, EventArgs e)
{
    Cursor = Cursors.Hand;
}

private void AddQuestionButton_MouseLeave(object sender, EventArgs e)
{
    Cursor = Cursors.Default;
}

private void DeleteQuestionButton_MouseEnter(object sender, EventArgs e)
{
    Cursor = Cursors.Hand;
}

private void DeleteQuestionButton_MouseLeave(object sender, EventArgs e)
```

```
        Cursor = Cursors.Default;
    }

    private void EditQuestionsButton_MouseEnter(object sender, EventArgs e)
{
    Cursor = Cursors.Hand;
}

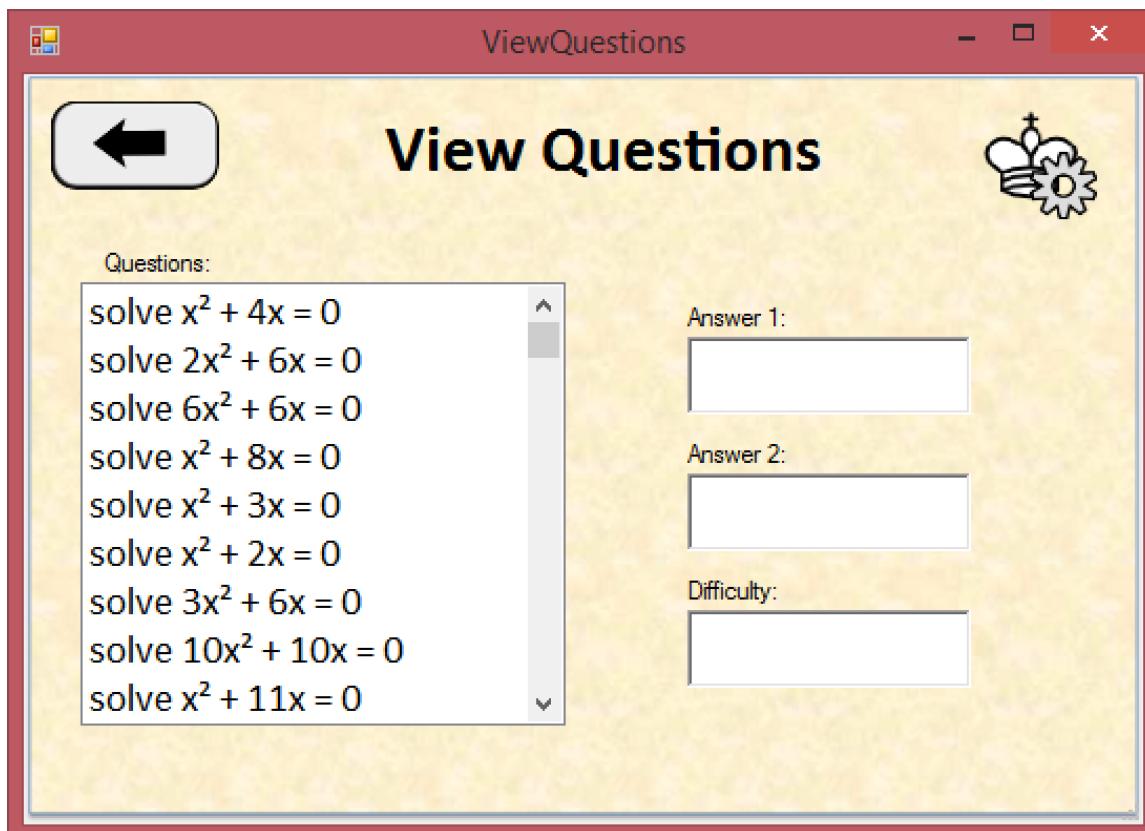
private void EditQuestionsButton_MouseLeave(object sender, EventArgs e)
{
    Cursor = Cursors.Default;
}

private void BackButton_MouseEnter(object sender, EventArgs e)
{
    Cursor = Cursors.Hand;
}

private void BackButton_MouseLeave(object sender, EventArgs e)
{
    Cursor = Cursors.Default;
}

private void BackButton_Click(object sender, EventArgs e)
{
    Close();
}
}
```

## ViewQuestions



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO;
namespace test_chess
{
    public partial class ViewQuestions : Form
    {
        public ViewQuestions()
        {
            InitializeComponent();
            populateListBox();

            //set certain properties on the form
            Answer1Box.ReadOnly = true;
            Answer2Box.ReadOnly = true;
            DifficultyBox.ReadOnly = true;
            Answer1Box.BackColor = Color.White;
            Answer2Box.BackColor = Color.White;
            DifficultyBox.BackColor = Color.White;
        }
    }
```

```
}

//populate the list view control of all the question objects
private void populateListBox()
{
    foreach (QuestionBank aQuestion in BoardAdmin.questions)
    {
        QuestionTextList.Items.Add(aQuestion.questiontext);
    }
}

// when a question is selected in the list box, update the other textboxes to show
the corresponding properties
private void QuestionTextList_SelectedIndexChanged(object sender, EventArgs e)
{

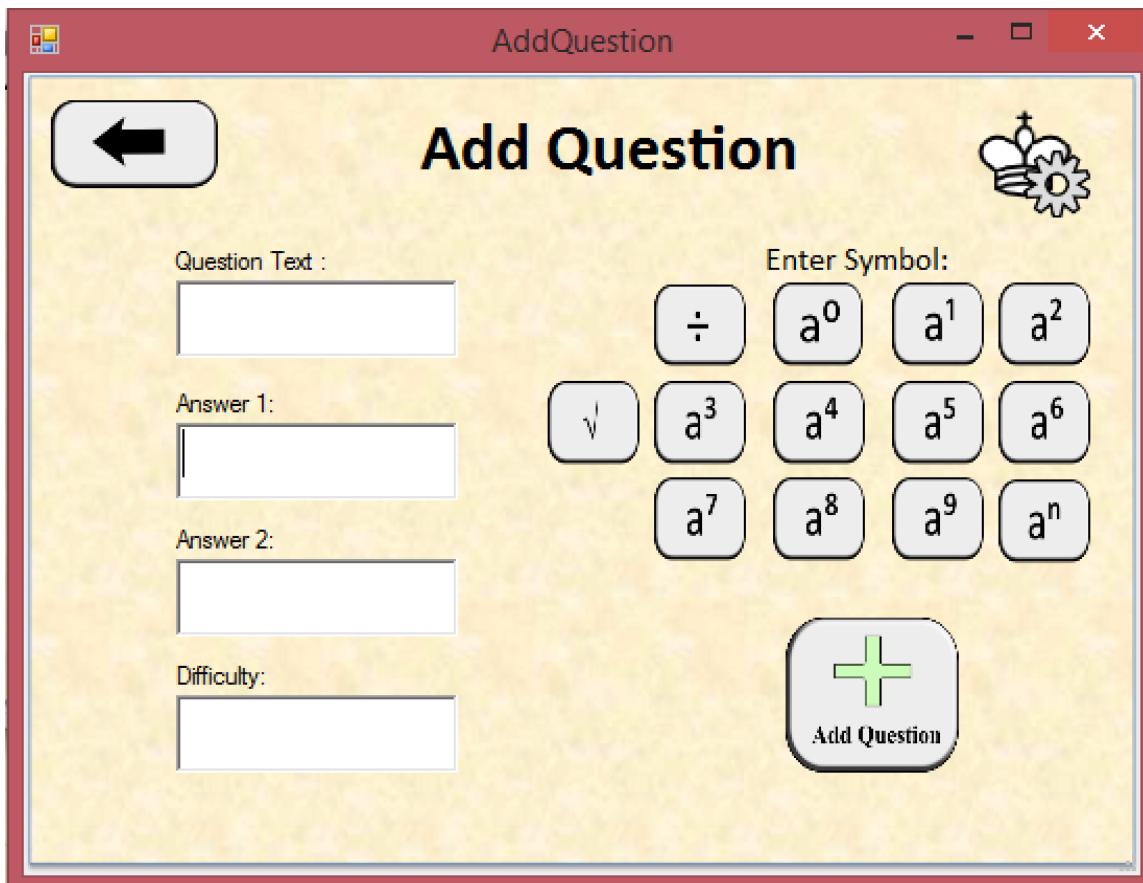
    foreach (QuestionBank aQuestion in BoardAdmin.questions)
    {
        if ((string)QuestionTextList.SelectedItem == aQuestion.questiontext)
        {
            Answer1Box.Text = aQuestion.answer.ToString();
            Answer2Box.Text = aQuestion.answer2.ToString();
            DifficultyBox.Text = aQuestion.difficulty.ToString();
        }
    }

    // if there is no second answer then grey the field out
    if (Answer2Box.Text == "")
    {
        Answer2Box.BackColor = Color.Gray;
    }
    else
    {
        Answer2Box.BackColor = Color.White;
    }
}

private void BackButton_Click(object sender, EventArgs e)
{
    this.Close();
}

}
```

## AddQuestion



```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO;

namespace test_chess
{
    public partial class AddQuestion : Form
    {
        //stores the current focused text box
        private RichTextBox focusedTextBox;

        public AddQuestion()
        {
            InitializeComponent();

            //combines the 'enter' event for all texboxes on the form so they perform the
            same procedure
            //A global variable is assigned to which stores the textbox that the cursor is

```

```
currently in
    foreach (RichTextBox tb in this.Controls.OfType<RichTextBox>())
    {
        tb.Enter += QuestionTextBox_Enter;
    }

    focusedTextBox = QuestionTextBox;
}

//appends the text to the current textbox that is in focus
private void SquareRootButton_Click(object sender, EventArgs e)
{
    focusedTextBox.AppendText("√");
}

//event that is run for all textboxes on the form
private void QuestionTextBox_Enter(object sender, EventArgs e)
{
    focusedTextBox = (RichTextBox)sender;
}

// runs when the back button is clicked.
private void BackButton_Click(object sender, EventArgs e)
{
    Close();
}

// the following appends the symbols corresponding to the buttons clicked to the
text box that has focus.

private void DivideButton_Click(object sender, EventArgs e)
{
    focusedTextBox.AppendText("÷");
}

private void powerButton_0_Click(object sender, EventArgs e)
{
    focusedTextBox.AppendText("0");
}

private void powerButton_1_Click(object sender, EventArgs e)
{
    focusedTextBox.AppendText("1");
}

private void powerButton_2_Click(object sender, EventArgs e)
{
    focusedTextBox.AppendText("2");
}

private void powerButton_3_Click(object sender, EventArgs e)
{
    focusedTextBox.AppendText("3");
}

private void powerButton_4_Click(object sender, EventArgs e)
{
    focusedTextBox.AppendText("4");
}
```

```
}

private void powerButton_5_Click(object sender, EventArgs e)
{
    focusedTextBox.AppendText("5");
}

private void powerButton_6_Click(object sender, EventArgs e)
{
    focusedTextBox.AppendText("6");
}

private void powerButton_7_Click(object sender, EventArgs e)
{
    focusedTextBox.AppendText("7");
}

private void powerButton_8_Click(object sender, EventArgs e)
{
    focusedTextBox.AppendText("8");
}

private void powerButton_9_Click(object sender, EventArgs e)
{
    focusedTextBox.AppendText("9");
}

private void powerButton_n_Click(object sender, EventArgs e)
{
    focusedTextBox.AppendText("n");
}

private void AddQuestion_Load(object sender, EventArgs e)
{
}

private void RootSymbolLabel_Click(object sender, EventArgs e)
{
}

private void Answer1Box_Enter(object sender, EventArgs e)
{
}

private void Answer2Box_Enter(object sender, EventArgs e)
{
}

private void DifficultyBox_Enter(object sender, EventArgs e)
{
}
```

```
//This procedure carries out the task of adding the question to the file stored on
the server.
//It checks that the input is valid and then it appends a new line containing the
new question,
//in the correct format
private void addQuestionToFile()
{
    if (QuestionTextBox.Text == "")
    {
        MessageBox.Show("Please enter the question text", "error");
    }
    if (Answer1Box.Text == "")
    {
        MessageBox.Show("Please enter the question Answer", "error");
    }
    if (DifficultyBox.Text == "")
    {
        MessageBox.Show("Please enter the Difficulty", "error");
    }

    int num; // stores the result of the conversion

    bool isint = int.TryParse(DifficultyBox.Text, out num); // this checks that the
difficulty is an integer
    if (isint == true)
    {
        if(Convert.ToInt16(DifficultyBox.Text) > 3) // the difficulty level cannot
be greater than 3
        {
            MessageBox.Show("the difficulty cannot exceed 3", "error");
        }
        else{
            if (QuestionTextBox.Text != "" && Answer1Box.Text != "")
            {
                QuestionBank newQuestion = new QuestionBank(QuestionTextBox.Text,
Answer1Box.Text, Answer2Box.Text, Convert.ToInt16(DifficultyBox.Text));
                BoardAdmin.questions.Add(newQuestion);

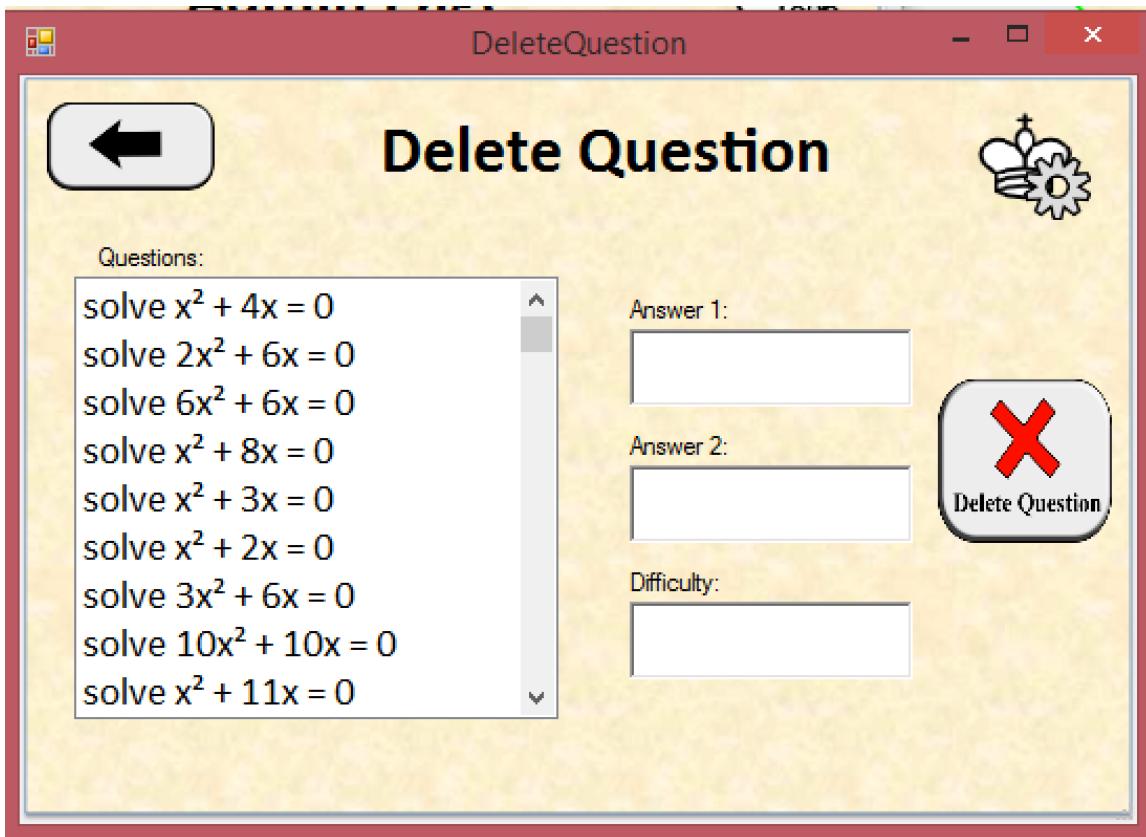
                File.AppendAllText("QuestionBank 2.csv", newQuestion.questiontext + ","
+ newQuestion.answer + "," + newQuestion.answer2 + "," + newQuestion.difficulty);

                MessageBox.Show("Question added successfully!", "Successful!");
            }
        }
    }
}

// runs when the add button is clicked
private void AddQuestionButton_Click(object sender, EventArgs e)
{
    addQuestionToFile();
}

}
```

## DeleteQuestion



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO;

namespace test_chess
{
    public partial class DeleteQuestion : Form
    {
        public DeleteQuestion()
        {
            InitializeComponent();

            // when the form loads, populate the list box with questions from the list of
            question objects
            populateListBox();

            // set properties
            Answer1Box.ReadOnly = true;
        }
    }
}
```

```

        Answer2Box.ReadOnly = true;
        DifficultyBox.ReadOnly = true;
        Answer1Box.BackColor = Color.White;
        Answer2Box.BackColor = Color.White;
        DifficultyBox.BackColor = Color.White;

    }

    // retrieves the questions from the list of question objects and uses them to
    // populate the control
    private void populateListBox()
    {
        foreach (QuestionBank aQuestion in BoardAdmin.questions)
        {
            QuestionTextList.Items.Add(aQuestion.questiontext);
        }
    }

    // when a question from the control is selected, update the other textboxes with the
    // questions properties
    private void QuestionTextList_SelectedIndexChanged(object sender, EventArgs e)
    {
        foreach (QuestionBank aQuestion in BoardAdmin.questions)
        {
            if ((string)QuestionTextList.SelectedItem == aQuestion.questiontext)
            {
                Answer1Box.Text = aQuestion.answer.ToString();
                Answer2Box.Text = aQuestion.answer2.ToString();
                DifficultyBox.Text = aQuestion.difficulty.ToString();
            }
        }

        if (Answer2Box.Text == "")
        {
            Answer2Box.BackColor = Color.Gray;
        }
        else
        {
            Answer2Box.BackColor = Color.White;
        }
    }

    private void BackButton_Click(object sender, EventArgs e)
    {
        Close();
    }

    // carries out the deletion of the question
    // deletes from list of question objects first, then updates text file accordingly
    by rewriting all remaining
    // question objects
    private void deleteQuestionFromFile()
    {
        QuestionBank QToDelete = null;
        if (QuestionTextList.SelectedItem != null)
        {
            foreach (QuestionBank aQuestion in BoardAdmin.questions)
            {

                if (aQuestion.questiontext.ToString() ==
QuestionTextList.SelectedItem.ToString())
                {
                    QToDelete = aQuestion;
                    BoardAdmin.questions.Remove(QToDelete);
                    break;
                }
            }
        }
    }
}

```

```
        }

    }

    // writer for writing to file
    StreamWriter aSW = new StreamWriter(File.Open("QuestionBank 2.csv",
FileMode.Truncate));

    // rewrites each question to the file - OVERWRITES
    foreach (QuestionBank aQuestion in BoardAdmin.questions)
    {
        aSW.WriteLine(aQuestion.questiontext + "," + aQuestion.answer + "," +
aQuestion.answer2 + "," + aQuestion.difficulty);
    }

    aSW.Close(); // close stream

    // re-populate list box with the new contents of the question object list
and hence the text file
    QuestionTextList.Items.Clear();
    populateListBox();
}
}

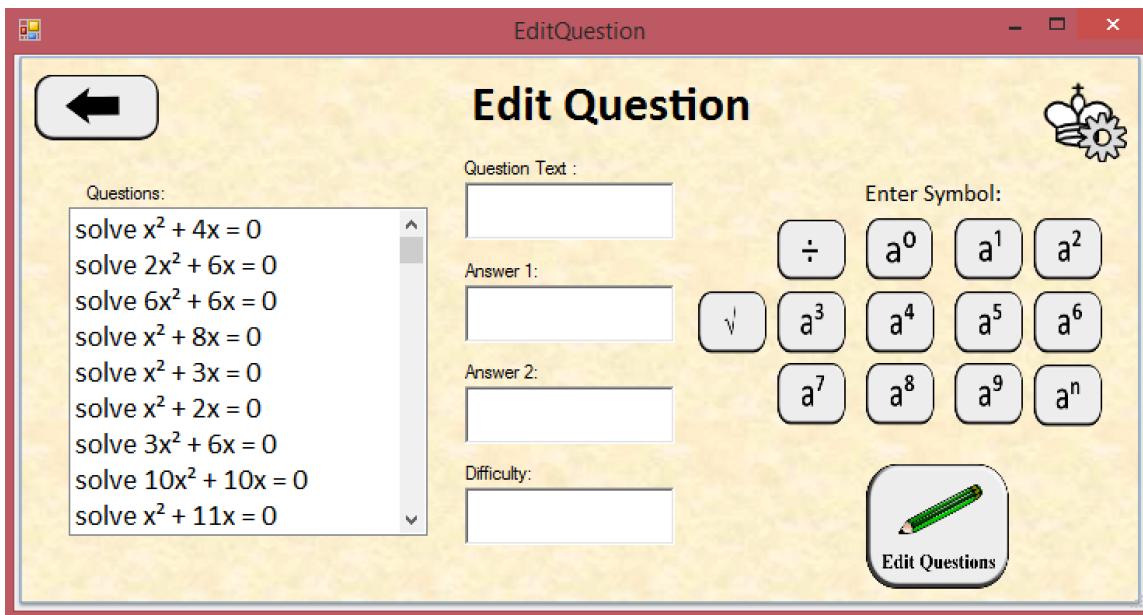
// when the delete button is clicked
private void DeleteQuestionButton_Click(object sender, EventArgs e)
{
    if (QuestionTextList.SelectedItem != null)
    {
        // gives a warning to the user about deleting the question
        DialogResult dialogResult = MessageBox.Show("Are you sure you want to delete
this question? ", "Warning", MessageBoxButtons.YesNo);

        if (dialogResult == DialogResult.Yes)
        {
            deleteQuestionFromFile();
        }
        else if (dialogResult == DialogResult.No)
        {
            //cancels delete operation of question
        }
    }
}

}

}
```

## EditQuestion



```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO;

namespace test_chess
{
    public partial class EditQuestion : Form
    {

        //stores the current focused text box
        private RichTextBox focusedTextBox;

        public EditQuestion()
        {
            InitializeComponent();
            populateListBox(); // populates the list box which contains the list of
questions the user can select to edit

            // combines the "enter" event so that all textboxes run the same event code
            foreach (RichTextBox tb in this.Controls.OfType<RichTextBox>())
            {
                tb.Enter += QuestionTextBox_Enter;
            }

            focusedTextBox = QuestionTextBox;
        }
    }
}

```

```
// populates list control with the question text property of the questions
private void populateListBox()
{
    foreach (QuestionBank aQuestion in BoardAdmin.questions)
    {
        QuestionTextList.Items.Add(aQuestion.questiontext);
    }
}

// when an item is selected/changed in the list box
private void QuestionTextList_SelectedIndexChanged(object sender, EventArgs e)
{
    // change the contents of the other textboxes on the form to match the selected
    question
    foreach (QuestionBank aQuestion in BoardAdmin.questions)
    {
        if ((string)QuestionTextList.SelectedItem == aQuestion.questiontext)
        {
            QuestionTextBox.Text = aQuestion.questiontext.ToString();
            Answer1Box.Text = aQuestion.answer.ToString();
            Answer2Box.Text = aQuestion.answer2.ToString();
            DifficultyBox.Text = aQuestion.difficulty.ToString();
        }
    }

    // if there is no second answer then grey the field out.
    if (Answer2Box.Text == "")
    {
        Answer2Box.BackColor = Color.Gray;
    }
    else
    {
        Answer2Box.BackColor = Color.White;
    }
}

private void BackButton_Click(object sender, EventArgs e)
{
    Close();
}

private void QuestionTextBox_Enter(object sender, EventArgs e)
{
    focusedTextBox = (RichTextBox)sender;
}

//append special symbols to the currently focused textbox
private void SquareRootButton_Click_1(object sender, EventArgs e)
{
    focusedTextBox.AppendText("√");
}

private void DivideButton_Click(object sender, EventArgs e)
{
    focusedTextBox.AppendText("÷");
}

private void powerButton_0_Click(object sender, EventArgs e)
{
    focusedTextBox.AppendText("⁰");
}

private void powerButton_1_Click(object sender, EventArgs e)
```

```
{  
    focusedTextBox.AppendText("1");  
}  
  
private void powerButton_2_Click(object sender, EventArgs e)  
{  
    focusedTextBox.AppendText("2");  
}  
  
private void powerButton_3_Click(object sender, EventArgs e)  
{  
    focusedTextBox.AppendText("3");  
}  
  
private void powerButton_4_Click(object sender, EventArgs e)  
{  
    focusedTextBox.AppendText("4");  
}  
  
private void powerButton_5_Click(object sender, EventArgs e)  
{  
    focusedTextBox.AppendText("5");  
}  
  
private void powerButton_6_Click(object sender, EventArgs e)  
{  
    focusedTextBox.AppendText("6");  
}  
  
private void powerButton_7_Click(object sender, EventArgs e)  
{  
    focusedTextBox.AppendText("7");  
}  
  
private void powerButton_8_Click(object sender, EventArgs e)  
{  
    focusedTextBox.AppendText("8");  
}  
  
private void powerButton_9_Click(object sender, EventArgs e)  
{  
    focusedTextBox.AppendText("9");  
}  
  
private void powerButton_n_Click(object sender, EventArgs e)  
{  
    focusedTextBox.AppendText("n");  
}  
  
// when the edit button is clicked  
private void EditQuestionsButton_Click(object sender, EventArgs e)  
{  
    //checks that there is a selection  
    if (QuestionTextList.SelectedItem != null)  
    {  
  
        // validates to ensure all appropriate data is available  
        if (QuestionTextBox.Text == "" || Answer1Box.Text == "")  
        {  
            MessageBox.Show("fields are empty", "error");  
        }  
        else  
        {  
            int num; // stores result of conversion  
            bool isint = int.TryParse(DifficultyBox.Text, out num); // checks for
```

```

integer values
{
    if (isint == true)
    {
        if (num > 3)
        {
            MessageBox.Show("the difficulty cannot exceed 3", "error");
        }
        else
        {
            QuestionBank QtoEdit = null;

            foreach (QuestionBank aQuestion in BoardAdmin.questions)
            {
                QuestionTextList.SelectedItem = aQuestion;

                if (aQuestion.questiontext.ToString() ==
QuestionTextList.SelectedItem.ToString())
                {
                    QtoEdit = aQuestion;
                    BoardAdmin.questions.Remove(QtoEdit);
                    break;
                }
            }

            QtoEdit.questiontext = QuestionTextBox.Text;
            QtoEdit.answer = Answer1Box.Text;
            QtoEdit.answer2 = Answer2Box.Text;

            QtoEdit.difficulty = Convert.ToInt16(DifficultyBox.Text);

            BoardAdmin.questions.Add(QtoEdit);

            File.WriteAllText("QuestionBank 2.csv", string.Empty);

            StreamWriter aSW = new StreamWriter(File.Open("QuestionBank
2.csv", FileMode.Open));

            foreach (QuestionBank aQuestion in BoardAdmin.questions)
            {
                // re writes each question as a line in the text file
                aSW.WriteLine(aQuestion.questiontext + "," +
aQuestion.answer + "," + aQuestion.answer2 + "," + aQuestion.difficulty);

            }

            aSW.Close(); // closes the stream

            // the following takes the text from the file and removes all
blank lines after the content to ensure no errors occur
            // when reading in from the file on another process
            string temptext = File.ReadAllText("QuestionBank 2.csv");
            temptext = temptext.Trim();
            File.WriteAllText("QuestionBank 2.csv", temptext); // overwrites

            QuestionTextList.Items.Clear();
            populateListBox();

            MessageBox.Show("Question edited successfully!", "Successful!");
        }
    }
}
else

```

```
        {
            MessageBox.Show("The difficulty must be a number", "error");
        }
    }
}
```

# Object orientated Classes

## GAME (Class)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace test_chess
{
    abstract class Game
    {
        public enum result
        {
            BlackWon,
            WhiteWon,
            None
        }

        public enum gameType
        {
            Standard,
            Timed,
            Maths,
        }
    }

    private Player m_player1 = new Player();
    private Player m_player2 = new Player();
    private Board m_gameBoard = new Board();
    private result m_gameResult;
    private Piece.Piececolour m_turn;
    private List<Move> m_MoveHistory = new List<Move>();
    private gameType m_gameType;

    private gameType p_gameType
    {
        get { return m_gameType; }
        set { m_gameType = value; }
    }
}
```

```
}

private result gameResult
{
    get { return m_gameResult; }
    set { m_gameResult = value; }
}

private Piece.Piececolour turn
{
    get { return m_turn; }
    set { m_turn = value; }
}

private Player player1
{
    get { return m_player1; }
    set { m_player1 = value; }
}
private Player player2
{
    get { return m_player2; }
    set { m_player2 = value; }
}
private Board gameBoard
{
    get { return m_gameBoard; }
    set { m_gameBoard = value; }
}

private List<Move> MoveHistory
{
    get { return m_MoveHistory; }
    set { m_MoveHistory = value; }
}

// creates the game by setting player colours and by creating the game board and
// setting it up with pieces
public virtual void createGame(Piece.Piececolour player1colour, Piece.Piececolour
player2colour)
{
    player1.setPlayerColour(player1colour);
    player2.setPlayerColour(player2colour);
    gameBoard = new Board();

    gameBoard.setUpBoard(player1colour);
    // white moves first in chess
    turn = Piece.Piececolour.WHITE;
}

public gameType getGameType()
{
    return p_gameType;
}
public void setGameType(gameType aGameType)
{
    p_gameType = aGameType;
}

public result getGameResult()
{
    return gameResult;
```

```

        }
        public void setGameResult(result aGameResult)
        {
            gameResult = aGameResult ;
        }

        public List<Move> getMoveHistory()
        {
            return MoveHistory;
        }
        public void addToMoveHistory(Move aMove)
        {
            MoveHistory.Add(aMove);
        }

        public Piece.Piececolour getTurn()
        {
            return turn;
        }
        public void setTurn(Piece.Piececolour TurnColour)
        {
            turn = TurnColour;
        }
        public Board getGameBoard()
        {
            return gameBoard;
        }
        public Player getPlayer1()
        {
            return player1;
        }
        public Player getPlayer2()
        {
            return player2;
        }

        // method to undo last move
        // uses the top-most MOVE object in the game objects moveHistory list property to
undo the move
        public void undoLastMove()
        {

            if (_MoveHistory.Count != 0)
            {

                Move lastMove = getMoveHistory().Last<Move>();

                if (_turn == Piece.Piececolour.BLACK)
                {
                    foreach (Piece apiece in gameBoard.getPieceSets()[1].getPieceSet())
                    {
                        if (apiece.getPosition() == lastMove.PieceMoved.getPosition())
                            // incase of pawn promotion, the new piece will have the same
position
                        {
                            if (apiece.getType() != lastMove.PieceMoved.getType())
                            {

                                gameBoard.getPieceSets()[1].getPieceSet().Remove(apiece);
                                gameBoard.getPieceSets()
[1].getPieceSet().Add(lastMove.PieceMoved);

                                break;
                            }
                        }
                    }
                }
            }
        }
    }
}

```

```

        }
    }
}

if (m_turn == Piece.Piececolour.WHITE) // same for white pieces
{
    foreach (Piece apiece in gameBoard.getPieceSets()[0].getPieceSet())
    {
        if (apiece.getPosition() == lastMove.PieceMoved.getPosition())
        {
            if (apiece.getType() != lastMove.PieceMoved.getType())
            {

                gameBoard.getPieceSets()[0].getPieceSet().Remove(apiece);
                gameBoard.getPieceSets()
[0].getPieceSet().Add(lastMove.PieceMoved);
            }
        }
    }
}

// uses properties of move object to reverse the effects of the move
lastMove.PieceMoved.setPosition(lastMove.initialPosition);
lastMove.PieceMoved.decrementMoveCount(1);
lastMove.initialPosition.occ = true;
lastMove.finalPosition.occ = false;

this.setTurn(lastMove.PlayerMoved.getPlayerColour());

if (lastMove.pieceCaptured != null)
{
    Piece pieceToReturn = lastMove.pieceCaptured;
    if (pieceToReturn.getColour() == Piece.Piececolour.BLACK)
    {
        m_gameBoard.getPieceSets()[0].getPieceSet().Add(pieceToReturn);
        pieceToReturn.getPosition().occ = true;
    }
    else
    {
        m_gameBoard.getPieceSets()[1].getPieceSet().Add(pieceToReturn);
        pieceToReturn.getPosition().occ = true;
    }
}

// if the previous move was a castling procedure then the rook needs to be
moved back also
// this is only for temporary movement
if (BoardAdmin.isCastling == true)
{
    if (BoardAdmin.TempMoving == true)
    {
        if (BoardAdmin.KingsideRook != null)
        {
            BoardAdmin.KingsideRook.getPosition().occ = false;
            BoardAdmin.KingsideRook.setPosition(gameBoard.getSquares()
[BoardAdmin.KingsideRook.getPosition().X - 2, BoardAdmin.KingsideRook.getPosition().Y]);
            BoardAdmin.KingsideRook.getPosition().occ = true;
            BoardAdmin.KingsideRook.decrementMoveCount(1);
            BoardAdmin.isCastling = false;
        }
    }
}

```

```
        }

        // removes the move object from the list
        m_MoveHistory.Remove(lastMove);

        if (BoardAdmin.game1.getMoveHistory().Count != 0)
        {
            // if the actual move was a castling procedure and not temporary then the
            rook has its own move object
            // therefore the top TWO moves in the moveHistory list needs to be
            removed
            // Hence this is recursive as the method is called again for the rook to
            undo its move aswell
            // as the king.
            if (BoardAdmin.game1.getMoveHistory().Last<Move>().castled == true)
            {

                if (BoardAdmin.TempMoving == false)
                {
                    BoardAdmin.isCastling = false;
                    this.undoLastMove();    // call method again for rook

                }
                else
                {
                    m_MoveHistory.Remove(m_MoveHistory.Last<Move>());
                }
            }
        }
    }

    public abstract void setTimerValue(int aTimerValue);
    public abstract int getTimerValue();
    public abstract void setDifficulty(int player1, int player2);
    public abstract int getPlayer1Difficulty();
    public abstract int getPlayer2Difficulty();

}

}
```

## **PIECE (Class)**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;

namespace test_chess
{
    abstract class Piece
    {
        public enum Piececolour
        {
            BLACK,
            WHITE
        }
        public enum pieceType
        {
            PAWN,
            KING,
            QUEEN,
            ROOK,
            KNIGHT,
            BISHOP
        }

        private Square m_position;
        private Piececolour m_pieceColour;
        private pieceType m_pieceType;
        private System.Drawing.Image m_image;
        private int m_moveCount;

        private int p_moveCount
        {
            get { return m_moveCount; }
            set { m_moveCount = value; }
        }

        private Square p_position
        {
            get { return m_position; }
            set { m_position = value; }
        }

        private Piececolour p_pieceColour
        {
            get { return m_pieceColour; }
            set { m_pieceColour = value; }
        }

        private pieceType p_pieceType
        {
            get { return m_pieceType; }
            set { m_pieceType = value; }
        }
    }
}
```

```
public System.Drawing.Image p_image
{
    get { return m_image; }
    set{ m_image = value;}
}

public void IncrementMoveCount(int x)
{
    p_moveCount += x;
}
public void decrementMoveCount(int x)
{
    p_moveCount -= x;
}

public int getMoveCount()
{
    return p_moveCount;
}
public void setPosition(Square x)
{
    p_position = x;
}

public Square getPosition()
{
    return p_position;
}

public void setImage(System.Drawing.Image x)
{
    p_image = x;
}

public System.Drawing.Image getImage()
{
    return p_image;
}

public pieceType getType()
{
    return p_pieceType;
}

public void setType(pieceType type)
{
    p_pieceType = type;
}

public void setColour(Piececolour colour)
{
    p_pieceColour = colour;
}

public Piececolour getColour()
{
    return p_pieceColour;
}
```

```

//Check the move is valid for the selected piece
public virtual bool checkMoveIsValid(Square targetPosition, Board gameBoard)
{
    Square[,] board = gameBoard.getSquares();
    int targetYindex = 0, targetXindex = 0, currentYindex = 0, currentXindex = 0;

    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            if (targetPosition == board[i, j])
            {
                targetXindex = i;
                targetYindex = j;
            }
            if (this.getPosition() == board[i, j])
            {
                currentXindex = i;
                currentYindex = j;
            }
        }
    }

    if (CheckTargetPositionIsOnBoard(targetPosition, board) == true)
    {
        return true;
    }
    else
    {
        return false;
    }
}

// this procedure checks the target position to see whether the piece occupying it
can be captured
public virtual bool canCapture(Square targetPosition, Board gameBoard)
{
    Piece.Piececolour colour = this.getColour();
    PieceSet[] piecesets = gameBoard.getPieceSets();

    if (colour == Piece.Piececolour.BLACK)
    {
        foreach (Piece piece in piecesets[1].getPieceSet())
        {
            if (piece.getPosition() == targetPosition)
            {
                return true;
            }
        }
        return false;
    }
    else
    {
        foreach (Piece piece in piecesets[0].getPieceSet())

```

```

        {
            if (piece.getPosition() == targetPosition)
            {
                return true;
            }
        }
        return false;
    }

}

// this populates the possible moves array with all the moves that the piece can
make
// this is used alongside the checkValidMove algorithm
public Square[] getPossibleMoves(Board gameBoard)
{
    Square[,] board = gameBoard.getSquares();
    int index = 0;
    Square[] possiblemoves = new Square[25]; // max number of moves will not exceed
25

    // instantiates the square objects in the array
    for (int x = 0; x < 25; x++)
    {
        possiblemoves[x] = new Square();
    }

    // populates the array with 11 valid moves - after scanning through the board
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            if (this.checkMoveIsValid(board[i, j], gameBoard) == true)
            {

                possiblemoves[index] = board[i, j];
                if (index < 24)
                {
                    index++;
                }
            }
        }
    }
    Array.Resize(ref possiblemoves, index); // "truncates" the array to its length
size
    return possiblemoves;
}

//This checks the target position to see whether it is on the 8x8 board. (ie not
outside the bounds of the array)
public virtual bool CheckTargetPositionIsOnBoard(Square targetposition, Square[,] board2)
{
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            if (board2[i, j] == targetposition)
            {

```

```

        return true;
    }
}
return false;
}

public abstract void promotePawn();

}
}

```

## Board (Class)

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace test_chess
{
    class Board
    {
        //the board is represented by 8 x 8 array of square objects - initiallised in the
        //setupboard method
        private Square[,] squares = new Square[8, 8];

        //two piecesets, black and white for the board.
        private PieceSet[] pieceSets = new PieceSet[2];

        private Square[,] p_squares
        {
            get { return squares; }
            set { squares = value; }
        }

        private PieceSet[] p_pieceSets
        {
            get { return pieceSets; }
            set { pieceSets = value; }
        }

        public PieceSet[] getPieceSets()
        {
            return p_pieceSets;
        }
        public void setPieceSets(PieceSet[] pieceSetVariable)
        {
            p_pieceSets = pieceSetVariable;
        }
        public Square[,] getSquares()
        {
            return p_squares;
        }
        public void setsquares(Square[,] squaresVariable)
        {

```

```

        p_squares = squaresVariable;
    }

    //sets up the board by creating two piecelists, initialising those lists.
    //also initialising the square object array.
    public void setUpBoard(Piece.Piececolour player1colour)
    {

        //instantiates square objects
        for (int j = 0; j < 8; j++)
        {
            for (int k = 0; k < 8; k++)
            {
                squares[j, k] = new Square();
            }
        }
    }

    PieceSet blackPieces = new PieceSet();
    PieceSet whitePieces = new PieceSet();

    // invokes methods which set all the properties if each piece
    blackPieces.setInitialPieceList(Piece.Piececolour.BLACK, squares,
player1colour );
    whitePieces.setInitialPieceList(Piece.Piececolour.WHITE, squares,
player1colour );

    pieceSets[0] = blackPieces;
    pieceSets[1] = whitePieces;

    //assigns the x and y co-ordinate properties to the squares of the board
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            squares[i, j].X = i;
            squares[i, j].Y = j;
        }
    }

    //sets the squares that the pieces are on, to occupied status
    foreach (Piece piece in blackPieces.getPieceSet())
    {
        piece.getPosition().occ = true;
    }
    foreach (Piece piece in whitePieces.getPieceSet())
    {
        piece.getPosition().occ = true;
    }
}

//generates string array of board positions "A2" etc
public String[,] generateBoardPositions()
{
    String[,] positions = new string[8, 8];

    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
    }
}

```

```
{  
    if (i == 0)  
    {  
        positions[i, j] = "A" + (j + 1);  
    }  
    if (i == 1)  
    {  
        positions[i, j] = "B" + (j + 1);  
    }  
    if (i == 2)  
    {  
        positions[i, j] = "C" + (j + 1);  
    }  
    if (i == 3)  
    {  
        positions[i, j] = "D" + (j + 1);  
    }  
    if (i == 4)  
    {  
        positions[i, j] = "E" + (j + 1);  
    }  
    if (i == 5)  
    {  
        positions[i, j] = "F" + (j + 1);  
    }  
    if (i == 6)  
    {  
        positions[i, j] = "G" + (j + 1);  
    }  
    if (i == 7)  
    {  
        positions[i, j] = "H" + (j + 1);  
    }  
}  
  
}  
  
return positions;  
  
}  
}
```

## Player (Class)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace test_chess
{
    class Player
    {
        private Piece.Piececolour m_playercolour;
        private double m_timeleft = 00.00;

        private Piece.Piececolour p_playercolour
        {
            get{return m_playercolour;}
            set{m_playercolour = value;}
        }

        private double timeleft
        {
            get { return m_timeleft; }
            set { m_timeleft = value; }
        }

        public void setTimeLeft(double aTime)
        {
            timeleft = aTime;
        }
        public double getTimeLeft()
        {
            return timeleft;
        }
        public Piece.Piececolour getPlayerColour()
        {
            return p_playercolour;
        }

        public void setPlayerColour(Piece.Piececolour colour)
        {
            p_playercolour = colour;
        }

        // this is where the moves of the game are made, and where the moves are added to history
        public void makeMove(Square targetposition, Board gameBoard, Piece movingPiece)
        {

            PieceSet[] pieceSets = gameBoard.getPieceSets();
            PieceSet BlackPieces = pieceSets[0];
            PieceSet WhitePieces = pieceSets[1];

            //Move object created and populated
            Move aMove = new Move();
            aMove.finalPosition = targetposition;
            aMove.initialPosition = movingPiece.getPosition();
            aMove.PieceMoved = movingPiece;
```

```
if (BoardAdmin.game1.getTurn() ==
BoardAdmin.game1.getPlayer1().getPlayerColour())

{
    aMove.PlayerMoved = BoardAdmin.game1.getPlayer1();

    if (BoardAdmin.game1.getPlayer1().getPlayerColour() ==
Piece.Piececolour.WHITE)

    {
        foreach (Piece piece in gameBoard.getPieceSets()[0].getPieceSet())

        {
            if (piece.getPosition() == targetposition)

            {
                aMove.pieceCaptured = piece;
            }
        }

        BlackPieces.getPieceSet().Remove(aMove.pieceCaptured);

    }
    if (BoardAdmin.game1.getPlayer1().getPlayerColour() ==
Piece.Piececolour.BLACK)

    {
        foreach (Piece piece in gameBoard.getPieceSets()[1].getPieceSet())

        {
            if (piece.getPosition() == targetposition)

            {
                aMove.pieceCaptured = piece;
            }
        }

        WhitePieces.getPieceSet().Remove(aMove.pieceCaptured);
    }
}

else // otherwise it is player 2's turn so do exactly the same but for player 2
{
    aMove.PlayerMoved = BoardAdmin.game1.getPlayer2();

    if (BoardAdmin.game1.getPlayer2().getPlayerColour() ==
Piece.Piececolour.WHITE)
    {
        foreach (Piece piece in gameBoard.getPieceSets()[0].getPieceSet())
        {
            if (piece.getPosition() == targetposition)
            {
                aMove.pieceCaptured = piece;
            }
        }

        BlackPieces.getPieceSet().Remove(aMove.pieceCaptured);
    }
    if (BoardAdmin.game1.getPlayer2().getPlayerColour() ==
Piece.Piececolour.BLACK)
    {
        foreach (Piece piece in gameBoard.getPieceSets()[1].getPieceSet())
        {
```

```

        if (piece.getPosition() == targetposition)
        {
            aMove.pieceCaptured = piece;
        }
    }
    WhitePieces.getPieceSet().Remove(aMove.pieceCaptured);
}

if (movingPiece.getType() == Piece.pieceType.KING)
    // if the moving piece is a king, castling may occur
{
    King movingKing = (King)movingPiece;
    if (movingKing.canCastle(targetposition, gameBoard) == true)
        // invokes the canCastle method of the king class to check whether
castling can occur
    {

        BoardAdmin.isCastling = true;

        Rook movingRook= null;

        if(this.getPlayerColour() == Piece.Piececolour.WHITE)

        {
            foreach(Piece piece in WhitePieces.getPieceSet())
            {
                if(piece.getType() == Piece.pieceType.ROOK &&
Math.Abs(piece.getPosition().X - movingKing.getPosition().X) == 3)

                {
                    movingRook = (Rook)piece;
                }
            }
        }
        if(this.getPlayerColour() == Piece.Piececolour.BLACK )

        {
            foreach(Piece piece in BlackPieces.getPieceSet())
            {
                if(piece.getType() == Piece.pieceType.ROOK &&
Math.Abs(piece.getPosition().X - movingKing.getPosition().X) == 3)
                {
                    movingRook = (Rook)piece;
                }
            }
        }
        BoardAdmin.KingsideRook = movingRook;
        movingKing.PerformCastle(BoardAdmin.KingsideRook, movingKing,
targetposition, gameBoard);
        // invokes the performCastle method in the king class to make the
castling move.

        if (BoardAdmin.TempMoving == false)
            // if temp moving is true, then the move needs to be retracted so
the boolean representing castling needs to remain true otherwise the kingside rook will not
be moved back aswell as the king
        {
            BoardAdmin.isCastling = false;
            // so if temp moving is false, the castling can be made false
aswell as the move is not temporary.
        }
    }
}

```

}

```

        // adjusts properties of the piece moving and the squares involved
        movingPiece.getPosition().occ = false;
        movingPiece.setPosition(targetposition);
        movingPiece.getPosition().occ = true;
        movingPiece.IncrementMoveCount(1);

        if (movingPiece.getType() == Piece.pieceType.PAWN) // if the moving piece is a
        pawn, promotion may occur
        {
            if (movingPiece.getPosition().Y == 0 || movingPiece.getPosition().Y == 7)
                //if either the bottom or the top of the board is reached
            {
                if (BoardAdmin.TempMoving == false) // only promotes pawn if it is not a
                temporary move
                {
                    movingPiece.promotePawn(); // invokes the promotePawn method to
                    perform the move.
                }
            }
        }

        BoardAdmin.game1.addTomoveHistory(aMove);
    }

    // this is an algorithm to determine whether the player is in check
    // this is achieved by searching the other players pieces and seeing whether the
friendly king
    // is occupying one of the other players piece's possible moves
    public bool isInCheck(Board gameBoard)
    {
        PieceSet[] pieceSets = gameBoard.getPieceSets();
        PieceSet BlackPieces = pieceSets[0];
        PieceSet WhitePieces = pieceSets[1];
        Piece kingToCheck = null;

        if (this.getPlayerColour() == Piece.Piececolour.BLACK)
        {
            foreach (Piece piece in BlackPieces.getPieceSet())
            {
                if (piece.getType() == Piece.pieceType.KING)
                {
                    kingToCheck = piece;
                }
            }
        }

        if (this.getPlayerColour() == Piece.Piececolour.WHITE)
        {
            foreach (Piece piece in WhitePieces.getPieceSet())
            {
                if (piece.getType() == Piece.pieceType.KING)
                {
                    kingToCheck = piece; // set the piece variable with this king piece.
                }
            }
        }
    }
}

```

```
if (kingToCheck != null) // makes sure that the piece is not null.  
{  
  
    if (this.getPlayerColour() == Piece.Piececolour.BLACK)  
    {  
        foreach(Piece piece in WhitePieces.getPieceSet())  
        {  
            if  
(piece.getPossibleMoves(gameBoard).Contains(kingToCheck.getPosition()))  
                // populates each pieces possible moves and then scans those possible moves to see  
                whether the kingToCheck has the same position as one of those moves  
  
            {  
                return true;  
            }  
        }  
        return false;  
    }  
    else // otherwise the player is using black pieces  
    {  
        foreach (Piece piece in BlackPieces.getPieceSet())  
        {  
            if  
(piece.getPossibleMoves(gameBoard).Contains(kingToCheck.getPosition()))  
            {  
  
                return true;  
            }  
        }  
        return false;  
    }  
  
}  
  
return false; // if the piece is null then return false as their is no  
comparisons made.  
}  
  
}  
}
```

## pieceList (Class)

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace test_chess
{
    class PieceSet
    {
        // a set of pieces contains 16 pieces initially
        private List<Piece> m_pieceSet = new List<Piece>(16);
        private Piece.Piececolour m_colour;

        private Piece.Piececolour colour
        {
            get { return m_colour; }
            set { m_colour = value; }
        }

        private List<Piece> pieceSet
        {
            get { return m_pieceSet; }
            set { m_pieceSet = value; }
        }

        public List<Piece> getPieceSet()
        {

            return pieceSet;
        }

        //initialises all piece objects and fills up the corresponding piece list.

        public void setInitialPieceList(Piece.Piececolour piececolour, Square[,] squares,
Piece.Piececolour player1colour)
        {

            int index; // the index of the pieces on the board (first row pieces - king,
rook etc)
            int pawnindex; // pawns are in the second row, so have their own index

            setColour(piececolour);

            //creates all the piece objects neccessary
            Pawn[] Blackpawns = new Pawn[8];
            Pawn[] Whitepawns = new Pawn[8];
            Rook[] BlackRooks = new Rook[2];
            Rook[] WhiteRooks = new Rook[2];
            Bishop[] WhiteBishops = new Bishop[2];
            Bishop[] BlackBishops = new Bishop[2];
            Knight[] WhiteKnights = new Knight[2];
            Knight[] BlackKnights = new Knight[2];
            Queen WhiteQueen = new Queen();
            King WhiteKing = new King();
            Queen BlackQueen = new Queen();
            King BlackKing = new King();
        }
    }
}

```

```

int i = 0;

//Initialise all piece objects
for (int j = 0; j < 8; j++)
{
    Blackpawns[j] = new Pawn();
    Whitepawns[j] = new Pawn();
}

for (int j = 0; j < 2; j++)
{
    WhiteBishops[j] = new Bishop();
    BlackBishops[j] = new Bishop();
}

for (int j = 0; j < 2; j++)
{
    BlackRooks[j] = new Rook();
    WhiteRooks[j] = new Rook();
}

for (int j = 0; j < 2; j++)
{
    WhiteKnights[j] = new Knight();
    BlackKnights[j] = new Knight();
}

//fills up the list with the pieces and setting their properties

if (colour == Piece.Piececolour.BLACK)
{
    i = 0; // the x coordinate this can be reassigned as the pieces are
populated

        //player 1 pieces needs to be at the bottom of the board so assign the
indexes accordingly
        if (player1colour == Piece.Piececolour.BLACK)
        {
            index = 7;
            pawnindex = 6;
        }
        else
        {
            index = 0;
            pawnindex = 1;
        }

        // sets properties of pawns
        foreach (Pawn pawn in Blackpawns)
        {
            pawn.setType(Piece.pieceType.PAWN);
            pawn.setColour(Piece.Piececolour.BLACK);
            pawn.setImage(test_chess.Properties.Resources.Chess_pdt60);
            pawn.setPosition(squares[i, pawnindex ]);
            if (player1colour == Piece.Piececolour.BLACK)
            {
                pawn.movedirection = -1;
            }
            else
            {
                pawn.movedirection = 1;
            }
            m_pieceSet.Add(pawn);
            i++;
        }
}

```

```

        }

        i = 0; // x coordinate value
        // sets properties of rooks
        foreach (Rook rook in BlackRooks)
        {
            rook.setType(Piece.pieceType.ROOK);
            rook.setColour(Piece.Piececolour.BLACK);
            rook.setImage(test_chess.Properties.Resources.Chess_rdt60);
            rook.setPosition(squares[i, index]);
            m_pieceSet.Add(rook);

            i = 7;
        }

        i = 1;// x coordinate value
        foreach (Knight knight in BlackKnights)
        {
            knight.setType(Piece.pieceType.KNIGHT);
            knight.setColour(Piece.Piececolour.BLACK);
            knight.setImage(test_chess.Properties.Resources.Chess_ndt60);
            knight.setPosition(squares[i, index]);
            m_pieceSet.Add(knight);
            i = 6;
        }

        i = 2; // x coordinate value

        foreach (Bishop bishop in BlackBishops)
        {
            bishop.setType(Piece.pieceType.BISHOP);
            bishop.setColour(Piece.Piececolour.BLACK);
            bishop.setImage(test_chess.Properties.Resources.Chess_bdt60);
            bishop.setPosition(squares[i, index]);
            m_pieceSet.Add(bishop);
            i = 5;
        }

        BlackKing.setType(Piece.pieceType.KING);
        BlackKing.setColour(Piece.Piececolour.BLACK);
        BlackKing.setImage(test_chess.Properties.Resources.Chess_kdt60);
        BlackKing.setPosition(squares[3, index]);

        BlackQueen.setType(Piece.pieceType.QUEEN);
        BlackQueen.setColour(Piece.Piececolour.BLACK);
        BlackQueen.setImage(test_chess.Properties.Resources.Chess_qdt60);
        BlackQueen.setPosition(squares[4, index]);

        m_pieceSet.Add(BlackKing);

        m_pieceSet.Add(BlackQueen);

    }
}
else
{
    // otherwise player 1 is black and so black pieces must be at the bottom of
the board
    if (player1colour == Piece.Piececolour.BLACK)
    {
        index = 0;
}

```

```
        pawnindex = 1;
    }
    else
    {
        index = 7;
        pawnindex = 6;
    }

    i = 0;

    foreach (Pawn pawn in Whitepawns)
    {
        pawn.setType(Piece.pieceType.PAWN);
        pawn.setColour(Piece.Piececolour.WHITE);
        pawn.setImage(test_chess.Properties.Resources.Chess_plt60);
        pawn.setPosition(squares[i, pawnindex ]);
        if (player1colour == Piece.Piececolour.WHITE)
        {
            pawn.movedirection = -1;
        }
        else
        {
            pawn.movedirection = 1;
        }
        m_pieceSet.Add(pawn);
        i++;
    }

    i = 0;

    foreach (Rook rook in WhiteRooks)
    {
        rook.setType(Piece.pieceType.ROOK);
        rook.setColour(Piece.Piececolour.WHITE);
        rook.setImage(test_chess.Properties.Resources.Chess_rlt60);
        rook.setPosition(squares[i, index]);
        m_pieceSet.Add(rook);
        i = 7;
    }

    i = 2;

    foreach (Bishop bishop in WhiteBishops)
    {
        bishop.setType(Piece.pieceType.BISHOP);
        bishop.setColour(Piece.Piececolour.WHITE);
        bishop.setImage(test_chess.Properties.Resources.Chess_blt60);
        bishop.setPosition(squares[i, index]);
        m_pieceSet.Add(bishop);
        i = 5;
    }

    i = 1;

    foreach (Knight knight in WhiteKnights)
    {
        knight.setType(Piece.pieceType.KNIGHT);
        knight.setColour(Piece.Piececolour.WHITE);
        knight.setImage(test_chess.Properties.Resources.Chess_nlt60);
        knight.setPosition(squares[i, index]);
        m_pieceSet.Add(knight);
        i = 6;
    }
```

```
        WhiteKing.setType(Piece.pieceType.KING);
        WhiteKing.setColour(Piece.Piececolour.WHITE);
        WhiteKing.setImage(test_chess.Properties.Resources.Chess_klt60);
        WhiteKing.setPosition(squares[3, index]);

        WhiteQueen.setType(Piece.pieceType.QUEEN);
        WhiteQueen.setColour(Piece.Piececolour.WHITE);
        WhiteQueen.setImage(test_chess.Properties.Resources.Chess_qlt60);
        WhiteQueen.setPosition(squares[4, index]);

        m_pieceSet.Add(WhiteQueen);
        m_pieceSet.Add(WhiteKing);

    }

}

private void setColour(Piece.Piececolour acolour)
{
    colour = acolour;
}

}
```

## Board Admin (Class)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace test_chess
{
    abstract class BoardAdmin
    {

        //static variables passed from main menu
        public static Game game1;
        public static String gameStyle;
        public static Piece.Piececolour player1Colour;
        public static int player1Diff;
        public static int player2Diff;

        //static variables used in gameplay
        public static Piece.pieceType promotingPawnType ;
        public static bool isCastling = false;
        public static Rook KingsideRook;
        public static bool TempMoving = false;

        //maths static variables
        public static List<QuestionBank> questions;
```

```

        public static QuestionBank currentQuestion;
        public static int player1Score;
        public static int player2Score;
        public static int NumOfQuestionsAsked1;
        public static int NumOfQuestionsAsked2;

    }
}

```

## QuestionBank (Class)

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace test_chess
{
    class QuestionBank
    {
        // properties corresponding to a question
        private String m_questionText;
        private int m_difficulty;
        private string m_answer;
        private string m_answer2;

        public string questiontext
        {
            get { return m_questionText; }
            set { m_questionText = value; }
        }

        public int difficulty
        {
            get { return m_difficulty; }
            set { m_difficulty = value; }
        }

        public string answer
        {
            get { return m_answer; }
            set { m_answer = value; }
        }

        public string answer2
        {
            get { return m_answer2; }
            set { m_answer2 = value; }
        }

        // overriden method which allows the properties to be assigned when the object is
        created
        public QuestionBank(string aQuest, string aAns, string aAns2, int aDiff)
        {
            questiontext = aQuest;
            difficulty = aDiff;
            answer = aAns;
        }
    }
}

```

```
        answer2 = aAns2;  
    }  
}  
}
```

## Bishop (Class)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace test_chess
{
    class Bishop : Piece
    {

        //overriding procedure for the BISHOP
        public override bool checkMoveIsValid(Square targetPosition, Board gameBoard)
        {
            Square[,] board = gameBoard.getSquares();

            if (base.checkMoveIsValid(targetPosition, gameBoard) == false) // generic method
from piece class
            {
                return false;
            }
            else
            {

                //System.Windows.Forms.MessageBox.Show("BISHOP CHECK TEST");
                int targetYindex = targetPosition.Y, targetXindex = targetPosition.X,
currentYindex = this.getPosition().Y, currentXindex = this.getPosition().X;

                //Bishop moves diagonally, so check that the change in x and the change in y
are equal
                //then check which direction the movement is (>0 etc).
                if (Math.Abs(targetXindex - currentXindex) == Math.Abs(targetYindex -
currentYindex))
                {
                    //moving upwards and to the left
                    if (targetXindex - currentXindex < 0 && targetYindex - currentYindex <
0)
                    {
                        for (int i = currentXindex - 1; i >= targetXindex; i--)
                            //loops through to see if any squares inbetween are occupied
                        {
                            currentYindex -= 1;
                            if (board[i, currentYindex].occ == true)
                            {
                                if (i == targetXindex)
                                {
                                    if (canCapture(board[i, currentYindex], gameBoard) ==
true)
                                        return true;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```
        }
    }

}

//moving downwards and to the left
if (targetXindex - currentXindex < 0 && targetYindex - currentYindex >
0)
{
    for (int i = currentXindex - 1; i >= targetXindex; i--)
        //loops through to see if any squares inbetween are occupied
    {
        currentYindex += 1;
        if (board[i, currentYindex].occ == true)
        {
            if (i == targetXindex)
            {
                if (canCapture(board[i, currentYindex], gameBoard) ==
true)
                {
                    return true;
                }
            }
            return false;
        }
    }
}

//moving upwards and to the right
if (targetXindex - currentXindex > 0 && targetYindex - currentYindex <
0)
{
    for (int i = currentXindex + 1; i <= targetXindex; i++)
        //loops through to see if any squares inbetween are occupied
    {
        currentYindex -= 1;
        if (board[i, currentYindex].occ == true)
        {
            if (i == targetXindex)
            {
                if (canCapture(board[i, currentYindex], gameBoard) ==
true)
                {
                    return true;
                }
            }
            return false;
        }
    }
}

//moving downwards and to the right
if (targetXindex - currentXindex > 0 && targetYindex - currentYindex >
0)
{
    for (int i = currentXindex + 1; i <= targetXindex; i++)
        //loops through to see if any squares inbetween are occupied
    {
        currentYindex += 1;
        if (board[i, currentYindex].occ == true)
        {
            if (i == targetXindex)
            {
                if (canCapture(board[i, currentYindex], gameBoard) ==
true)
                {
                    return true;
                }
            }
        }
    }
}
```

```
        }
        return false;
    }
}

return true;
}
return false;
}

}

// abstract method for pawn piece only but must be implemented in all inherited
classes.
public override void promotePawn()
{
    throw new NotImplementedException();
}
}
```

## King (Class)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace test_chess
{
    class King : Piece
    {
        // overrides from Piece class, specific to king piece
        public override bool checkMoveIsValid(Square targetPosition, Board gameBoard)
        {
            Square[,] board = gameBoard.getSquares();

            if (base.checkMoveIsValid(targetPosition, gameBoard) == false)
            {
                return false;
            }
            else
            {

                int targetYindex = targetPosition.Y, targetXindex = targetPosition.X,
currentYindex = this.getPosition().Y, currentXindex = this.getPosition().X;
                // if the target position is 1 square in any horizontal direction, then this
is valid
                if (targetXindex == currentXindex && targetYindex == currentYindex)
                {
                    return true;
                }
                // if the target position is 1 square in any diagonal direction, then this
is valid
                if (Math.Abs(targetXindex - currentXindex) > 1 || Math.Abs(targetYindex -
currentYindex ) > 1)
```

```

    {
        if (Math.Abs(targetXindex - currentXindex) > 1)
        {
            // checks whether the king can perform a castling move
            if (canCastle(targetPosition, gameBoard) == true)
            {

                return true;
            }
            else
            {
                return false;
            }
        }
        else
        {
            return false;
        }
    }
    else
    {
        if (targetPosition.occ == true)
        {
            // checks whether the occupying piece is an enemy piece and can be
            if (canCapture(board[targetXindex], targetYindex, gameBoard) ==
true)
            {
                return true;
            }
            return false;
        }
        else
        {
            return true;
        }
    }
}

// evaluates whether the king can perform the castle procedure by finding the
kingside rook and determining
// whether both the king and rook havent moved yet and whether the spaces between
them are empty
public bool canCastle(Square targetPosition, Board gameBoard)
{
    // if the target position is castling then the square will be 2 x - coordinates
away
    if (Math.Abs(targetPosition.X - this.getPosition().X) == 2)
    {
        PieceSet blackpieces = gameBoard.getPieceSets()[0];
        PieceSet whitepieces = gameBoard.getPieceSets()[1];

        Rook kingsideRook = null;

        if(this.getColour() == Piececolour.BLACK)
        {
            foreach (Piece piece in blackpieces.getPieceSet())
            {
                if (piece.getType() == pieceType.ROOK)
                {
                    if (Math.Abs(this.getPosition().X - piece.getPosition().X) == 3)
                    {

```

captured  
true)

```

        }
    }

}

if (this.getColour() == Piececolour.WHITE)

{
    foreach (Piece piece in whitepieces.getPieceSet())

    {
        if (piece.getType() == pieceType.ROOK)

        {
            if (Math.Abs(this.getPosition().X - piece.getPosition().X) == 3)

            {
                kingsideRook = (Rook)piece;
            }
        }
    }

if (kingsideRook != null)

{
    if (kingsideRook.getMoveCount() == 0)

    {
        if (this.getMoveCount() == 0)

        {
            if (targetPosition.Y == this.getPosition().Y &&
targetPosition.occ != true)
                // checks that the y coordinate of the target square is the
same as the king's and also that the target square is not occupied
            {

                for (int i = this.getPosition().X - 1 ; i >
kingsideRook.getPosition().X; i--)
                    // loops through between the rook and the king, to check the
squares in between to make sure they are not occupied.
                {
                    if (gameBoard.getSquares()[i, this.getPosition().Y].occ ==
true)

                    {
                        return false;
                    }
                }
            if (targetPosition.X < this.getPosition().X)
                // ensures that the algorithm only returns true for the
correct target square ( in between rook and king ) and not the other side of the king
            {
                return true;
            }
        else
        {
            return false;
        }
    }
else // otherwise, if the square is occupied or the y coordinates
}

```

are not the same

```

        {
            return false;
        }
    }
    else // if the king has already moved then return false
    {
        return false;
    }
}
else // if the rook has already moved before then return false
{
    return false;
}
}
else // if the kingside rook variable is null then return false
{
    return false;
}

}
else // if the target square is not 2 x-coordinates away then return false
{
    return false;
}
}

// performs the castling procedure
// moves the kingside rook as the standard makeMove algorithm in the player class
handles the king piece as that is the primary piece involved in the move and is the piece
that is clicked to perform the castle move.
public void PerformCastle(Rook kingsideRook, King movingKing, Square targetPosition,
Board gameBoard)
{
    // sets the target position for the rook
    Square rookTargetPosition = gameBoard.getSquare(targetPosition.X
+1,targetPosition.Y);

    //creates a move object for the rook and populates it
    Move aMove = new Move();
    aMove.initialPosition = kingsideRook.getPosition();
    aMove.finalPosition = rookTargetPosition;
    aMove.PieceMoved = kingsideRook;

    if (kingsideRook.getMoveCount() == 0)
    {
        aMove.castled = true;
    }
    if (BoardAdmin.game1.getTurn() ==
BoardAdmin.game1.getPlayer1().getPlayerColour())

    {
        aMove.PlayerMoved = BoardAdmin.game1.getPlayer1();

    }
    else
    {
        aMove.PlayerMoved = BoardAdmin.game1.getPlayer2();
    }

    //sets properties of kingside rook piece
    BoardAdmin.game1.addMoveHistory(aMove); // adds the move object to the list of
moves for the game history
}

```

```
        kingsideRook.getPosition().occ = false;
        kingsideRook.setPosition(rookTargetPosition);
        kingsideRook.getPosition().occ = true;
        kingsideRook.IncrementMoveCount(1);

    }

    //abstract class needs implementing
    public override void promotePawn()
    {
        throw new NotImplementedException();
    }
}
```

## Knight (Class)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace test_chess
{
    class Knight : Piece
    {
        // override piece for the knight
        public override bool checkMoveIsValid(Square targetPosition, Board gameBoard)
        {
            Square[,] board = gameBoard.getSquares();

            if (base.checkMoveIsValid(targetPosition, gameBoard) == false)
            {
                return false;
            }
            else
            {
                // sets target and current indexes as variables
                int targetYindex = targetPosition.Y, targetXindex = targetPosition.X,
currentYindex = this.getPosition().Y, currentXindex = this.getPosition().X;
                if (targetXindex == currentXindex && targetYindex == currentYindex)
                {
                    return true;
                }

                // a knight can move in an "L" shape and that means either 2 x-coordinates
                // and 1 y-coordinate or vice versa

                if ((Math.Abs((targetYindex - currentYindex)) == 2 && Math.Abs(targetXindex
- currentXindex) == 1) || (Math.Abs((targetXindex - currentXindex)) == 2 &&
Math.Abs(targetYindex - currentYindex) == 1))
                {
                    if (targetPosition.occ == true)
                    {
                        if (canCapture(board[targetXindex,targetYindex ], gameBoard) ==
true)
                        {
                            return true;
                        }
                    }
                }
            }
        }
    }
}
```

```
        return false;
    }
    else
    {
        return true;
    }
}
else
{
    return false;
}

}

//abstract class needs implementing
public override void promotePawn()
{
    throw new NotImplementedException();
}
}
```

## pawn (Class)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace test_chess
{
    class Pawn : Piece
    {
        private int m_movedirection; // the pawn can only move in one direction until it
is promoted

        public int movedirection
        {
            get { return m_movedirection; }
            set { m_movedirection = value; }
        }

        // overriden for pawn piece
        public override bool checkMoveIsValid(Square targetPosition, Board gameBoard)
        {
            Square[,] board = gameBoard.getSquares();

            if (base.checkMoveIsValid(targetPosition, gameBoard) == false)
            {
                return false;
            }
            else
            {
                int targetYindex = targetPosition.Y, targetXindex = targetPosition.X,
currentYindex = this.getPosition().Y, currentXindex = this.getPosition().X;
                if (targetPosition == this.getPosition())

```

```
        {
            return true;
        }
        // only moves upwards, so there is no change in y coordinate
        if (currentXindex == targetXindex)
        {
            if (_movedirection == -1 && targetYindex - currentYindex > 0)
            {

                return false;
            }
            if (_movedirection == 1 && targetYindex - currentYindex < 0)
            {
                return false;
            }
            // if it is the pawns first move, then it may move 2 spaces
            if (getMoveCount() == 0)
            {
                if (Math.Abs(targetYindex - currentYindex) > 2)
                {
                    return false;
                }
                else
                {
                    if (Math.Abs(targetYindex - currentYindex) == 2)
                    {
                        if (movedirection == -1)
                        {
                            if (board[currentXindex, currentYindex - 1].occ == true
|| board[currentXindex, currentYindex - 2].occ == true)
                            {
                                return false;
                            }
                            else
                            {
                                return true;
                            }
                        }
                        if (movedirection == 1)
                        {
                            if (board[currentXindex, currentYindex + 1].occ == true
|| board[currentXindex, currentYindex + 2].occ == true)
                            {
                                return false;
                            }
                            else
                            {
                                return true;
                            }
                        }
                    }
                }
            }
            // but it may also move 1 space should the user wish
            if (Math.Abs(targetYindex - currentYindex) == 1)
            {
                if (movedirection == -1)
                {
                    if (board[currentXindex, currentYindex - 1].occ == true)
                    {
                        return false;
                    }
                    else
                    {
                        return true;
                    }
                }
            }
        }
    }
```

```
        }
        if (movedirection == 1)
        {
            if (board[currentXindex, currentYindex + 1].occ == true)
            {
                return false;
            }
            else
            {
                return true;
            }
        }
    }
    else
    {
        return false;
    }
}

}

if (getMoveCount() != 0) // after the first move, it is only one space per turn
{
    if (Math.Abs(targetYindex - currentYindex) > 1 ||
Math.Abs(targetYindex - currentYindex) == 0)
    {
        return false;
    }
    else
    {

        if (Math.Abs(targetYindex - currentYindex) == 1)
        {
            if (movedirection == -1)
            {
                if (board[currentXindex, currentYindex - 1].occ == true)
                {
                    return false;
                }
            }
            if (movedirection == 1)
            {
                if (board[currentXindex, currentYindex + 1].occ == true)
                {
                    return false;
                }
            }
        }
    }
}
}

}

}

return true;
} // otherwise the pawn may capture a piece diagonally
```

```

        else
    {
        if (Math.Abs(targetXindex - currentXindex) == 1)
        {
            if (movedirection == -1)

                if (targetYindex - currentYindex == -1)

                    if (board[targetXindex, targetYindex].occ == true)
                    {
                        if (canCapture(board[targetXindex, targetYindex],
gameBoard) == true)
                        {
                            return true;
                        }
                    }
                }

            if (movedirection == 1)
            {
                if (targetYindex - currentYindex == 1)

                    if (board[targetXindex, targetYindex].occ == true)
                    {
                        if (canCapture(board[targetXindex, targetYindex],
gameBoard) == true)
                        {
                            return true;
                        }
                    }
                }

            }
        }

        return false;
    }

    // this occurs when the pawn piece reaches the other side of the board, the user may
promote the piece to a
    // better type of piece
public override void promotePawn()
{
    // opens the promote pawn sections and awaits a user selection

    PromotePawn promotingForm = new PromotePawn();

    if (promotingForm.ShowDialog() != System.Windows.Forms.DialogResult.OK)
    {
        //Waits until a selection has occurred on the other form
}

```

}

```
// checks that a selection has been made and stored
if (BoardAdmin.promotingPawnType == pieceType.BISHOP ||
BoardAdmin.promotingPawnType == pieceType.QUEEN || BoardAdmin.promotingPawnType ==
pieceType.ROOK || BoardAdmin.promotingPawnType == pieceType.KNIGHT)
{
    PieceSet[] pieceSets = BoardAdmin.game1.getGameBoard().getPieceSets();
    Piece promotedPiece = null;

    //sets properties of new piece to replace the pawn piece.
    if (BoardAdmin.promotingPawnType == pieceType.QUEEN)
    {
        promotedPiece = new Queen();
        promotedPiece.setType(pieceType.QUEEN);
        if (this.getColour() == Piececolour.BLACK)
        {

promotedPiece.setImage(test_chess.Properties.Resources.Chess_qdt60);
        }
        else
        {

promotedPiece.setImage(test_chess.Properties.Resources.Chess_qlt60);
        }
    }
    if (BoardAdmin.promotingPawnType == pieceType.BISHOP)
    {
        promotedPiece = new Bishop();
        promotedPiece.setType(pieceType.BISHOP);
        if (this.getColour() == Piececolour.BLACK)
        {

promotedPiece.setImage(test_chess.Properties.Resources.Chess_bdt60);
        }
        else
        {

promotedPiece.setImage(test_chess.Properties.Resources.Chess_blt60);
        }
    }
    if (BoardAdmin.promotingPawnType == pieceType.KNIGHT)
    {
        promotedPiece = new Knight();
        promotedPiece.setType(pieceType.KNIGHT);

        if (this.getColour() == Piececolour.BLACK)
        {

promotedPiece.setImage(test_chess.Properties.Resources.Chess_ndt60);
        }
        else
        {

promotedPiece.setImage(test_chess.Properties.Resources.Chess_nlt60);
        }
    }
    if (BoardAdmin.promotingPawnType == pieceType.ROOK)
    {
        promotedPiece = new Rook();
        promotedPiece.setType(pieceType.ROOK);
        if (this.getColour() == Piececolour.BLACK)
```

```
        }

        promotedPiece.setImage(test_chess.Properties.Resources.Chess_rdt60);
    }
    else
    {

        promotedPiece.setImage(test_chess.Properties.Resources.Chess_rlt60);
    }
}

promotedPiece.setColour(this.getColour());
promotedPiece.setPosition(this.getPosition());


// remove the pawn and add the new promoted piece
if (this.getColour() == Piececolour.BLACK)
{
    pieceSets[0].getPieceSet().Remove(this);
    pieceSets[0].getPieceSet().Add(promotedPiece)
}
else
{
    pieceSets[1].getPieceSet().Remove(this);
    pieceSets[1].getPieceSet().Add(promotedPiece)
}

}
```

## Queen (Class)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace test_chess
{
    class Queen : Piece
    {

        // override for queen
        public override bool checkMoveIsValid(Square targetPosition, Board gameBoard)
        {
            Square[,] board = gameBoard.getSquares();

            if (base.checkMoveIsValid(targetPosition, gameBoard) == false) // performs the
generic checkMoveIsValid procedure first
            {
                return false;
            }
            else // otherwise the overriding for the queen happens
            {

```

```

        int targetYindex = targetPosition.Y, targetXindex = targetPosition.X,
currentYindex = this.getPosition().Y, currentXindex = this.getPosition().X;
                // creates variables to store the current coordinates and the target
coordinates

        // a queens movement is both horizontal and diagonal so checks must be made
for both cases
        // for loops are used to check subsequent squares between the queen and the
target position
        if (targetYindex == currentYindex || targetXindex == currentXindex ||
Math.Abs(targetXindex - currentXindex) == Math.Abs(targetYindex - currentYindex))
        {
            // checks whether the target square is either horizontal or diagonal
movement

            if (targetYindex == currentYindex || targetXindex == currentXindex)
            {
                // then checks whether the movement is specifically horizontal
                if (targetYindex == currentYindex)
                {

                    if ((targetXindex - currentXindex) < 0)
                    {
                        // checks whether the movement is to the left
                        for (int i = currentXindex - 1; i >= targetXindex; i--)
                        {

                            if (board[i, currentYindex ].occ == true)
                            {
                                // if a particular square is occupied then the move
                                if (i == targetXindex)
                                {

                                    if (canCapture(board[i, currentYindex ],
gameBoard) == true)
                                    {
                                        // invoke the canCapture procedure to
                                        return true;
                                    }
                                }
                                return false;
                            }
                        }
                    }
                else // otherwise the movement is to the right
                {
                    for (int i = currentXindex + 1; i <= targetXindex; i++)
                    {

                        if (board[i, currentYindex ].occ == true)
                        {

                            if (i == targetXindex)
                            {

                                if (canCapture(board[i, currentYindex ],
gameBoard) == true)
                                {
                                    // invoke canCapture procedure to see
                                    whether the occupants of the square can be captured
                                    return true;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

```

to the target square cannot be made

determine whether capturing can take place

whether the occupants of the square can be captured

```

        }
        return false;
    }

}
}

else // otherwise the movement is in the y direction
{
    if ((targetYindex - currentYindex) < 0)
    {
        // checks whether the movement is upwards

        for (int i = currentYindex - 1; i >= targetYindex; i--)
        {

            if (board[currentXindex,i].occ == true)
            {

                if (i == targetYindex)
                {

                    if (canCapture(board[currentXindex, i],
gameBoard) == true)
                        // if so then check whether the occupants can
be captured
                    {

                        return true;
                    }
                }
                return false;
            }
        }
    }
    else // otherwise the movement is downwards
    {
        for (int i = currentYindex + 1; i <= targetYindex; i++)
        {

            if (board[currentXindex, i].occ == true)
            {
                // checks whether the square is occupied
                if (i == targetYindex)
                {

                    if (canCapture(board[currentXindex, i],
gameBoard) == true)
                        // if so then check whether the occupants can
be captured
                    {

                        return true;
                    }
                }
                return false;
            }
        }
    }
}

if (Math.Abs(targetXindex - currentXindex) == Math.Abs(targetYindex -
currentYindex))
//checks whether the movement is diagonal (equal increments of y

```

and x) by using the absolute function

```

    {
        if ((targetXindex - currentXindex) < 0 && (targetYindex -
currentYindex) < 0)
            // checks whether the movement is upwards, left diagonal
        {

            for (int i = Math.Abs((targetXindex - currentXindex )); i > 0;
i--)
                // loop a number of times which is equal to the difference
in either x or y
            {

                currentXindex = currentXindex - 1;
                currentYindex = currentYindex - 1;
                if (currentXindex >= 0 && currentYindex >= 0)
                {
                    if (board[currentXindex, currentYindex].occ == true)
                    {
                        if (currentXindex == targetXindex && currentYindex
== targetYindex)

                            {
                                if (canCapture(board[currentXindex,
currentYindex], gameBoard) == true)
                                    // invokes canCapture routine to check
whether capturing can take place
                                {
                                    return true;
                                }
                            }
                        return false;
                    }
                }
            }
        }
    }
}

if ((targetXindex - currentXindex) < 0 && (targetYindex -
currentYindex) > 0)
    // checks whether the movement is downwards, left diagonal
{
    for (int i = Math.Abs((targetXindex - currentXindex )); i > 0;
i--)
    {

        currentXindex = currentXindex - 1;
        currentYindex = currentYindex + 1;
        if (currentYindex != 8 && currentXindex >= 0 )
        {
            if (board[currentXindex, currentYindex].occ == true)
            {
                if (currentXindex == targetXindex && currentYindex
== targetYindex)

                    {
                        if (canCapture(board[currentXindex,
currentYindex], gameBoard) == true)
                            {
                                return true;
                            }
                    }
                return false;
            }
        }
    }
}

```

```

        }
        if ((targetXindex - currentXindex) > 0 && (targetYindex -
currentYindex) < 0)
            // checks whether the movement is upwards, right diagonal
        {
            for (int i = Math.Abs((targetXindex - currentXindex)); i > 0;
i--)
            {

                currentXindex = currentXindex + 1;
                currentYindex = currentYindex - 1;
                if (currentXindex != 8 && currentYindex >= 0)
                {
                    if (board[currentXindex, currentYindex].occ == true)
                    {
                        if (currentXindex == targetXindex && currentYindex
== targetYindex)
                            {
                                if (canCapture(board[currentXindex,
currentYindex], gameBoard) == true)
                                    {
                                        return true;
                                    }
                                return false;
                            }
                }
            }
        }

        if ((targetXindex - currentXindex) > 0 && (targetYindex -
currentYindex) > 0)
            // checks whether the movement is downwards, right diagonal
        {
            for (int i = Math.Abs((targetXindex - currentXindex)); i > 0;
i--)
            {

                currentXindex = currentXindex + 1;
                currentYindex = currentYindex + 1;
                if (currentXindex != 8 && currentYindex != 8 )
                {
                    if (board[currentXindex, currentYindex].occ == true)
                    {
                        if (currentXindex == targetXindex && currentYindex
== targetYindex)
                            {
                                if (canCapture(board[currentXindex,
currentYindex], gameBoard) == true)
                                    {
                                        return true;
                                    }
                                return false;
                            }
                }
            }
        }

        return true; // return true for the current position of the queen
    }
}

```

```
        }
    }
}

public override void promotePawn()
{
    throw new NotImplementedException();
}
}
```

## Rook (Class)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace test_chess
{
    class Rook : Piece
    {
        // overriden for rook
        public override bool checkMoveIsValid(Square targetPosition, Board gameBoard)
        {
            Square[,] board = gameBoard.getSquares();

            if (base.checkMoveIsValid(targetPosition, gameBoard) == false)
            {
                return false;
            }
            else{

                int targetYindex = targetPosition.Y, targetXindex = targetPosition.X,
                currentYindex = this.getPosition().Y, currentXindex = this.getPosition().X ;

                //rooks movement is horizontal, so either the y cooridnate or the x coordinates
                will be the same
                // as with all pieces, the squares between the current and target position are
                checked to be empty

                if (targetXindex != currentXindex && targetYindex != currentYindex)
                {
                    return false;
                }
                else
                {
                    if(targetXindex == currentXindex )
                    {
                        if (targetYindex - currentYindex < 0)
                        {
                            return false;
                        }
                    }
                }
            }
        }
    }
}
```

```

Centre Number: 61161
Candidate Number: 0394
for (int i = currentYindex - 1; i >= targetYindex; i--)
// loops through squares inbetween to check fo other pieces
{

    if (board[currentXindex, i].occ == true)
    {
        if (i == targetYindex)
        {
            if (canCapture(board[currentXindex, i], gameBoard)
== true)
                {
                    return true;
                }
        }
        return false;
    }
}
else
{
    for (int i = currentYindex + 1; i <= targetYindex; i++)
// loops through squares inbetween to check fo other pieces
{

    if (board[currentXindex, i].occ == true)
    {
        if (i == targetYindex)
        {
            if (canCapture(board[currentXindex, i], gameBoard)
== true)
                {
                    return true;
                }
        }
        return false;
    }
}
if (targetYindex == currentYindex)
{
    if (targetXindex - currentXindex < 0)
    {
        for (int i = currentXindex - 1; i >= targetXindex; i--)
// loops through squares inbetween to check fo other pieces
{

        if (board[i, currentYindex].occ == true)
        {
            if (i == targetXindex)
            {
                if (canCapture(board[i, currentYindex ], gameBoard)
== true)
                    {
                        return true;
                    }
            }
        }
        return false;
    }
}

```

```
        }
    }
}
else
{
    for (int i = currentXindex + 1; i <= targetXindex; i++)
        // loops through squares inbetween to check fo other pieces

    {
        if (board[i, currentYindex].occ == true)
        {
            if (i == targetXindex)
            {
                if (canCapture(board[i, currentYindex ], gameBoard)
== true)
                    {
                        return true;
                    }
                return false;
            }
        }
    }
}

return true;

}

}

}

public override bool canCapture(Square targetPosition, Board gameBoard)
{
    return base.canCapture(targetPosition, gameBoard);
}

public override void promotePawn()
{
    throw new NotImplementedException();
}

}
```

## Square (Class)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace test_chess
{
    class Square
    {
        // each square has an x and y coordinate
        private int m_X;
        private int m_Y;
        private bool m_occ; // to determine quickly whether the board square is occupied

        public int X
        {
            get { return m_X; }
            set { m_X = value; }
        }

        public int Y
        {
            get { return m_Y; }
            set { m_Y = value; }
        }

        public bool occ
        {
            get { return m_occ; }
            set { m_occ = value; }
        }
    }
}
```

## Move (Class)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace test_chess
{
    class Move
    {
        // properties that correspond to a detail for a move in the game
        private Player m_PlayerMoved;
        private Piece m_PieceMoved;
        private Square m_initialPosition;
        private Square m_finalPosition;
        private Piece m_pieceCaptured;
        private bool m_castled;

        public Player PlayerMoved
        {
            get { return m_PlayerMoved; }
            set { m_PlayerMoved = value; }
        }
        public Piece PieceMoved
        {
            get { return m_PieceMoved; }
            set { m_PieceMoved = value; }
        }
        public Square initialPosition
        {
            get { return m_initialPosition; }
            set { m_initialPosition = value; }
        }
        public Square finalPosition
        {
            get { return m_finalPosition; }
            set { m_finalPosition = value; }
        }
        public Piece pieceCaptured
        {
            get { return m_pieceCaptured; }
            set { m_pieceCaptured = value; }
        }
        public bool castled
        {
            get { return m_castled; }
            set { m_castled = value; }
        }
    }
}
```

## MathsGame (Class)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace test_chess
{
    class MathsGame : TimedGame
    {
        // properties specific to a maths game

        private int m_Player1questionDifficulty;

        private int Player1questionDifficulty
        {
            get { return m_Player1questionDifficulty; }
            set { m_Player1questionDifficulty = value; }
        }

        private int m_Player2questionDifficulty;

        private int Player2questionDifficulty
        {
            get { return m_Player2questionDifficulty; }
            set { m_Player2questionDifficulty = value; }
        }

        public override void setDifficulty(int player1, int player2)
        {
            Player1questionDifficulty = player1;
            Player2questionDifficulty = player2;
        }

        public override int getPlayer1Difficulty()
        {
            return Player1questionDifficulty;
        }

        public override int getPlayer2Difficulty()
        {
            return Player2questionDifficulty;
        }

        public override void createGame(Piece.Piececolour player1colour, Piece.Piececolour
player2colour)
        {
            base.createGame(player1colour,player2colour);

        }
    }
}
```

## **Standard Game (Class)**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace test_chess
{
    class StandardGame : Game
    {

        public override void setTimerValue(int aTimerValue)
        {
        }

        public override int getTimerValue()
        {
            return 0;
        }

        public override void setDifficulty(int player1, int player2)
        {
        }

        public override int getPlayer1Difficulty()
        {
            return 0;
        }
        public override int getPlayer2Difficulty()
        {
            return 0;
        }
    }
}
```

## Timed game (Class)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace test_chess
{
    class TimedGame : Game
    {
        private int m_timervalue;

        private int timerValue
        {
            get { return m_timervalue; }
            set { m_timervalue = value; }
        }

        public override void createGame(Piece.Piececolour player1colour, Piece.Piececolour
player2colour)
        {
            base.createGame(player1colour,player2colour);

        }

        public override void setTimerValue(int aValue)
        {
            timerValue = aValue;
        }

        public override int getTimerValue()
        {
            return timerValue;
        }

        public override void setDifficulty(int player1, int player2)
        {
        }

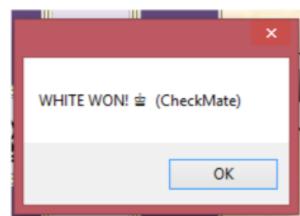
        public override int getPlayer1Difficulty()
        {
            return 0;
        }
        public override int getPlayer2Difficulty()
        {
            return 0;
        }
    }
}
```

# System Testing

**Test Table (Chess)**

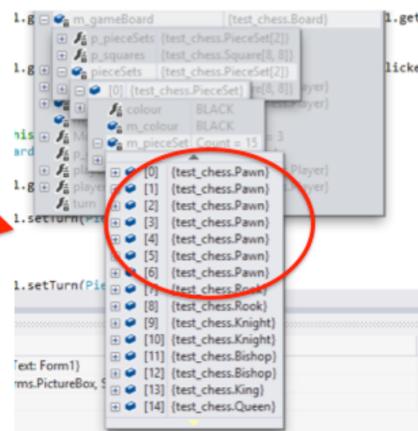
Test Number	Test To Carry Out	Description
1	Does the board dynamically position pieces based on the selection of player 1's colour?	If player 1 selects to be WHITE then the white pieces should be at the bottom of the board
2	Do squares highlight to show a piece's possible moves?	When clicking current square showing the possible pieces

**22)**



**Test Table (Maths System)**

Test Number	Test To Carry Out	Description	Test Type	Expected outcome	Passed? (P/F)
27	Does a maths question form appear with every move of the game, the result of which gets appended to a total	The system is, that a maths question is presented after every move in the game, the result of which gets appended to a total	Typical	A form should appear after every move	PASSED



Change to represent the current players turn	PASSED
--	--------

**Test Plan:**

My test plan is split into three sections, corresponding to the “sub-systems” in the program. These three sections are: Chess system, Maths System & Admin System. This is the most logical thing to do instead of testing the system as one whole because the sub systems do different things and so testing each part individually actually leads to a more thorough testing and hence the system becomes more robust.

The chess system involves many tests about the game itself, such as the movement of pieces, check and checkmate, special case moves, undoing a move etc. This, as discussed in the design section, is primarily whitebox testing as the tests involve individual “pathways” of code which carry out different things. Should an error occur, the root of the error will have to be identified and the way that part of the system executes should be monitored closely to try and debug the problem. The maths system, will test the retrieval of questions from the text file (stored on the server) at the entry point of the application and after that it will test whether the questions are presented correctly, and primarily whether the inputs submitted by the user have the right affect on the system. This will mean monitoring total variables closely (i.e total number of questions correctly answered etc). The Admin section of the system is mainly a min navigation system so the way in which the forms interact will be a key test here. Also, more importantly, the adding, deleting, editing and viewing of questions should all work correctly and the changes made should update the actual file on the server correctly and not just the question objects stored in memory while the program is running.

**Test Table (Chess System)**

Test Number	Test To Carry Out	Description	Test Type	Expected outcome	Passed? (P/F)
1	<b>Does the board dynamically position pieces based on the selection of player 1's colour?</b>	If player 1 selects to be WHITE then the white pieces should be at the bottom of the board (moving upwards). Depending on what colour player 1 chooses, the board should be dynamic.	Typical	White pieces should be at bottom if white selected and vice versa.	PASSED
2	Do squares highlight to show a piece's possible moves?	When a piece is clicked, of the current player's, green squares should show the moves that piece can make	Typical/ Erroneous	Green squares should show	PASSED

Test Number	Test To Carry Out	Description	Test Type	Expected outcome	Passed? (P/F)
3	Is the highlighting of squares dynamic and can the feature be turned on/off with immediate effect on the board?	There is a toggle which allows users to turn highlighting on/off. When toggled, the changes should be made immediately and show up again if turned back on.	Typical	As described	PASSED
4	If a piece has no possible moves, it should not be highlighted in green when clicked	To make the highlighting effective, the square that a piece is occupying should not turn green, to represent further that the piece cannot move	Typical/ Erroneous	As described	PASSED
5	The move history display should update with every move correctly.	The piece that moved should be represented using unicode and its initial and final position should be shown as well as any piece captured during the move	Typical/ Erroneous	As described	PASSED
6	The game should not allow black pieces to move first.	White move first in chess so it should be clear in the game that it is white's turn. The user should not be able to move black pieces	Typical/ Erroneous	As described	PASSED

Test Number	Test To Carry Out	Description	Test Type	Expected outcome	Passed? (P/F)
7	Pawn pieces should move correctly	A pawn may move two squares on its first move and only 1 square subsequently. A pawn may capture diagonally, 1 square.	Typical/ Erroneous	Pawn should move correctly	PASSED
8	Rook pieces should move correctly	A rook may move horizontally any amount of squares	Typical	Rook should move correctly	PASSED
9	Knight Pieces should move correctly	a knight may move in an “L” shape in any direction, a change in 2 squares in one direction and 1 square in the other. The knight should also be able to move “over” other pieces.	Typical	Knight should move correctly	PASSED
10	Bishop pieces should move correctly	A bishop should move diagonally only, any amount of spaces. The change in x and the change in y coordinates are the same	Typical	Bishop should move correctly	PASSED
11	Queen pieces should move correctly	A queen should be able to move in any direction, any amount of spaces	Typical	Queen should move correctly	FAILED
12	King pieces should move correctly	A king may only move 1 space at a time in any direction	Typical	King should move correctly	PASSED

Test Number	Test To Carry Out	Description	Test Type	Expected outcome	Passed? (P/F)
13	All pieces should be able to capture enemy pieces successfully	Piece of the opposite colour that are in the line of movement of a particular piece should be able to be captured, that piece should then be removed from the board and also the list of pieces (in code).	Typical	As described	PASSED
14	The game should allow the user to make the “castling” move successfully	This move allows the king and the rook closest to the king to make a special move under the conditions that both pieces havent moved yet and the spaces between them are empty	Typical/ Erroneous	The move should be made successfully	PASSED
15	The game should allow the user to promote a pawn successfully	When a pawn piece reaches the other side of the board, the user should be able to choose which piece to promote to. This piece should then replace the pawn on the form.	Typical	The pawn piece should be replaced by the promoted piece.	PASSED

Test Number	Test To Carry Out	Description	Test Type	Expected outcome	Passed? (P/F)
16	The undo button should be fully functioning	When the undo button is pressed, the move should be retracted and the turn of the game must change back to player who moved. Any captured pieces must be returned too. As well as this, the move history box should remove the last move	Typical	Moves should be retracted	PASSED
17	The special moves should also be able to be retracted using the undo feature	Using the undo button, the castling move and the promoting pawn move are special cases and must also be undone successfully	Typical	Castling should be undone and then be able to castle again and the pawn promotion should be changed back to a pawn.	PASSED
18	The selection of board style should change the background image of the squares on the chess board	In the main menu the user can select many colours and styles for the game board, the board should load with the correct chosen option	Typical	As described	PASSED

Test Number	Test To Carry Out	Description	Test Type	Expected outcome	Passed? (P/F)
19	The timer value of each player is correct	The initial value of the timer should be equal to the selected value of the user and also the timer label so be in time format so should count down from 60 seconds (not 99)	Typical/ Erroneous	Timer to countdown correctly	PASSED
20	The timer for a player not moving should be paused.	When it is not a players turn, their timer value should not be decreasing. It should be paused and resumed when it is their turn again	Typical/ Erroneous	Timer pauses correctly	PASSED
21	Game recognises a loss from zero timer	When a players timer is reduced to zero, the other player wins. The game should stop the game and provide this feedback. Players should not be able to move after this	Typical	Game recognises end and stops.	PASSED
22	The game should recognise checkmate successfully	When a player makes a move that is a checkmating move, the game should stop and display the game result. Players should not subsequently be able to move as the game has ended.	Typical	As described	PASSED

Test Number	Test To Carry Out	Description	Test Type	Expected outcome	Passed? (P/F)
23	“check” situations are handled successfully	When a king is in check, that king must move out of check or another piece should “block” it. Therefore a move should not be able to be made if the king is still in check afterwards. The system should prevent this from happening	Typical	A piece should not show any highlighted squares if it cannot be moved due to the king being in check.	PASSED
24	The little chess icon fluctuates correctly	A little icon is used to show the user who's turn it is, this icon should dynamically move next to the players labels representing who's turn it is and also the piece should be of the same colour as the players.	Typical/ Erroneous	As described	PASSED
25	Main Menu validation	The necessary options on the main menu should be selected before clicking the play button, validation needs to warn the user of missing selections	Typical/ Erroneous	message box warning to user	PASSED

Test Number	Test To Carry Out	Description	Test Type	Expected outcome	Passed? (P/F)
26	User typed values in numerical boxes	The numeric selections are a drop down list on the main menu but the user should not be able to manually change these values	Erroneous/Boundary	Program wont allow changes	PASSED

### Test Table (Maths System)

Test Number	Test To Carry Out	Description	Test Type	Expected outcome	Passed? (P/F)
27	Does a maths question form appear with every move of the game?	The system is, that a maths question is presented after every move in the game, the result of which gets appended to a total running score for maths.	Typical	A form should appear after every move	PASSED
28	On the question form, the player indicator is dynamic	In the top right corner of the form is an icon and label which indicates who's maths question it is for, this should work correctly and change accordingly	Typical	Change to represent the current players turn	PASSED
29	The question presented to the user is of the right difficulty	Each user has an assigned difficulty which they selected at the beginning of the game, the question retrieved should be of this difficulty	Typical/Erroneous	As described	PASSED

Test Number	Test To Carry Out	Description	Test Type	Expected outcome	Passed? (P/F)
30	If the question only has one answer, the second text box should be disabled and hidden	On the question form there is two boxes for input, but if the question retrieved only has one answer then the other box should not be available to the user.	Typical/ Erroneous	As described	PASSED
31	Does the system cater for invalid user input	When inputting data, appropriate validation should handle the mixture of string symbols (ie root sign) and integers	Erroneous	As described	PASSED
32	Solutions shown after question has been answered	After the user answers, the correct answers should be shown	Typical	As described	PASSED
33	Feedback should be given on a correct or incorrect answer	The question label should tell the user whether their inputs are correct or incorrect	Typical	As described	PASSED
34	Is time deducted correctly	if answered incorrectly, time should be deducted. 30 seconds for an easy question, 20 for level 2 and 10 for level 3.	Typical/ Erroneous	Time labels should show correctly subtracted time	PASSED
35	Is time deducted if the user exits the question form	The user may click to exit the form, but as a result of this, time should still be deducted	Typical	Time labels should show correctly subtracted time	PASSED

Test Number	Test To Carry Out	Description	Test Type	Expected outcome	Passed? (P/F)
36	Does the maths scores update and keep track correctly	There are maths scores shown on the game board form, these should update and show the latest scores after each question	Typical/ Erroneous	Maths labels should update correctly	PASSED
37	Does the back button only become available after a question has been answered?	The user should not be able to click the back button until they have answered the question	Typical	As described	PASSED
38	Do the timers pause while the player is answering the question	The respective players timer should pause while the question form is active	Typical	As described	PASSED

### Test Table (Admin System)

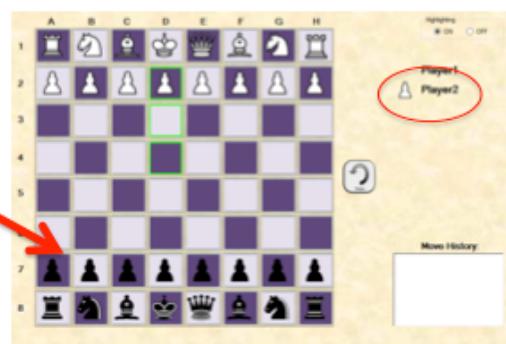
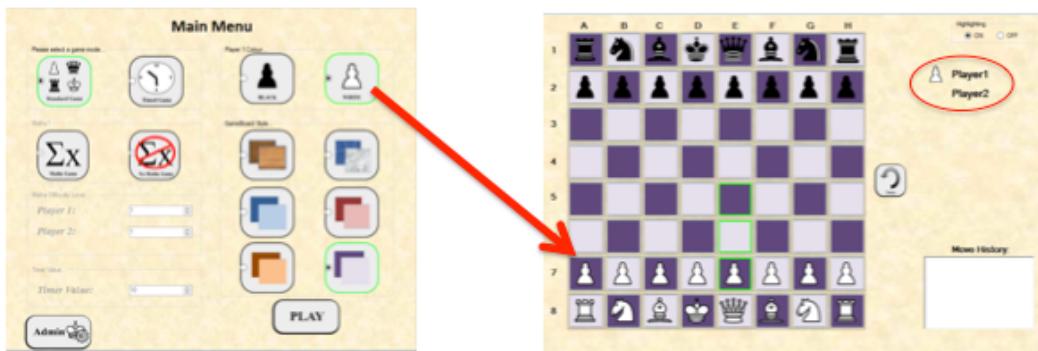
Test Number	Test To Carry Out	Description	Test Type	Expected outcome	Passed? (P/F)
39	Does the admin login work correctly	The admin login page should accept only the username and password, the correctly entered combination should allow the user to enter	Typical/ Erroneous	As Described	PASSED
40	Is the password hidden as it is typed?	For security purposes the password should be hidden with symbols when typing	Typical	As Described	PASSED

Test Number	Test To Carry Out	Description	Test Type	Expected outcome	Passed? (P/F)
41	Is appropriate feedback presented to the user if an incorrect login is used	The user should be notified if the username/ password combination is incorrect	Typical/ Erroneous	As Described	PASSED
42	Throughout the admin system, are the current forms focused	The current form should be at the front and the user should not be able to click off the form, making it go to the back.	Typical	As Described	PASSED
43	Are the contents of the text file correctly stored as question objects in memory?	The program, when executed, retrieves the contents of the text file on the server and populates question objects.	Typical	A list of question objects with correctly populated properties for question text, answers and difficulty.	PASSED
44	Do all buttons and navigation work properly?	Correct forms should appear and all buttons should work	Typical	As Described	PASSED
45	Does the view questions form work correctly?	The view questions form should correctly populate a list box with the question text properties from each question object, then when the user clicks one, its corresponding properties (answers and difficulty) should be shown in the adjacent text boxes - these should be readonly to the user.	Typical/ Erroneous	As Described	PASSED

Test Number	Test To Carry Out	Description	Test Type	Expected outcome	Passed? (P/F)
46	Do symbol buttons append text correctly	When symbol buttons are clicked, their corresponding symbol should be added to the current textbox that the cursor is in	Typical/ Erroneous	As Described	PASSED
47	The adding and editing forms should handle empty input or incorrect input correctly	The forms should not change anything when the add/edit button is clicked, when the changes are made, confirmation should be made to the user.	Typical/ Erroneous	Message boxes appearing to confirm changes if correct input and nothing if not correct.	PASSED
48	Adding a question should work correctly	When the user adds a question, the text file should update with that question in the correct format.	Typical	The text file should now have the added question	PASSED
49	Editing a question should work correctly	When the user edits a question, the question should be rewritten to the file with the updated properties - it will be at the bottom of the text file now due to the nature of making changes to a text file	Typical	As Described	FAILED

Test Number	Test To Carry Out	Description	Test Type	Expected outcome	Passed? (P/F)
50	Deleting a question should work correctly	When the user deletes question, the file should be updated accordingly and correctly to avoid any empty lines in the text file which would lead to further errors	Typical	As Described	PASSED
51	When making changes to the question file, the changes should be immediate and the list of questions viewable on the form should update	Each form in the admin section has a list view of questions and when changes are made, this list view should update immediately to show the effects	Typical	As Described	PASSED
52	If the admin page is closed, the user should be prompted for login again	The login screen should be presented every time the admin section is accessed, even in the same runtime.	Typical	As Described	PASSED
53	When adding or editing questions, the entered difficulty must not be lower than 1 or higher than 3.	The difficulty rating of questions only ranges 1-3 so any value other than this is incorrect for the system	Boundary	The system should give a warning of values entered outside the range and stop any processing.	PASSED

1)

**TEST 1 (Passed) :**

As shown, depending on what colour player 1 chooses, the board adjusts accordingly, but the white pieces still move first as it should be.

Also the player icons are adjusted too, with player 2 being white.

2)

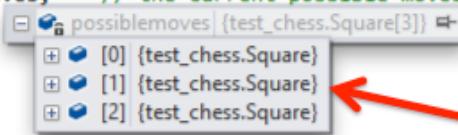
**TEST 2 (Passed) :**

Shown in three different scenarios, when a piece is clicked, squares are highlighted in green to represent possible moves for that piece. This also applies as highlighting is turned on in all three cases.

**TEST 2 (continued):**

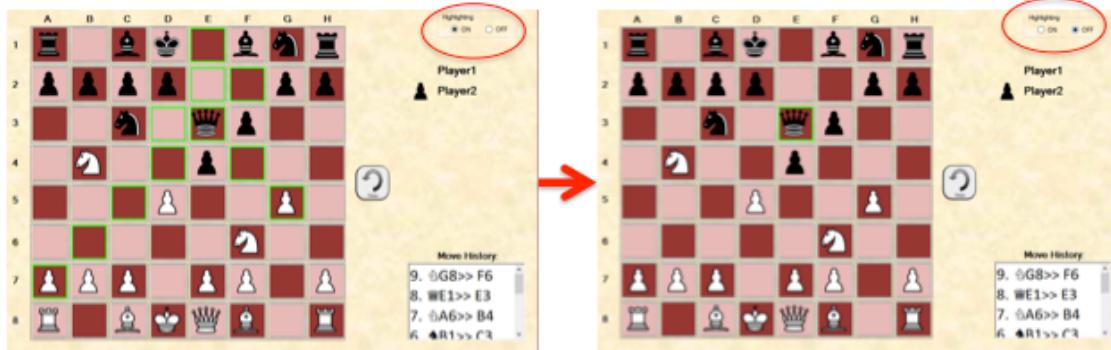
To test this further, a message box break was placed at end of the click event and the possible moves array was checked to see that the highlighted squares correspond to the square objects stored in the array and this test proved the highlighting to be working correctly.

```
Piece clickedPiece;           // current piece being processed
Square[] possiblemoves;      // the current possible moves for a clicked piece
public BoardUI()
{
    InitializeComponent();
    assignBoardsquares();
}
```



The squares stored in the possible moves array can be checked to see whether they correspond to the highlighted squares, which they do.

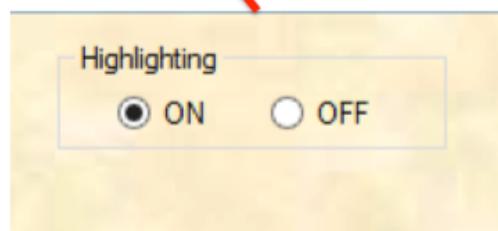
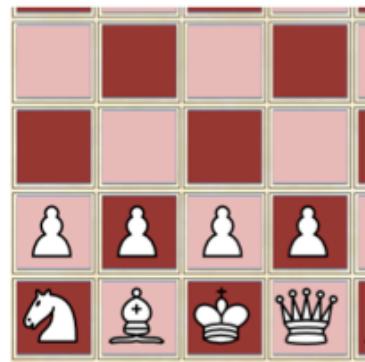
3)

**TEST 3 (Passed) :**

When the game starts the highlighting toggle is automatically selected as on. As shown, the possible moves highlight in green and when the option is toggled to no (second screenshot) the possible moves are no longer highlighted. Then when the toggle is on again, the green highlights reappear dynamically.

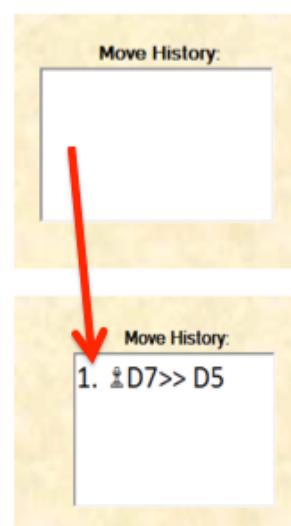
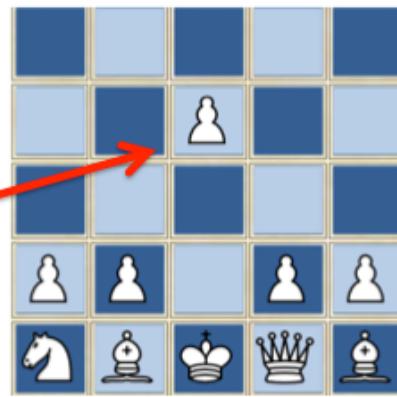
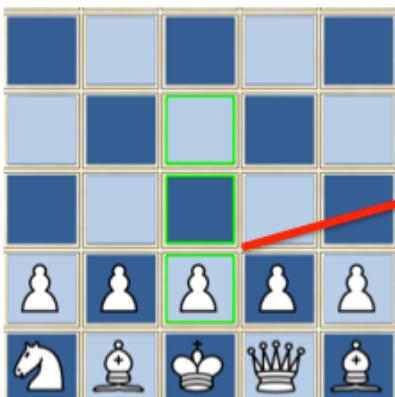


4)

**TEST 4 (Passed) :**

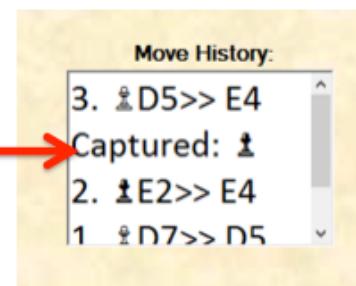
In the first screenshot, the pawn was clicked and its possible moves were highlighted. In the second screenshot, the BISHOP was clicked and because the bishop cannot move at the moment, it has no possible moves so the bishop's square was not highlighted in green, despite highlighting being turned on.

5)

**TEST 5 (Passed) :**

The move history box should update to represent each move made in the game. Here we can see it updating with the pawn's movement, the number representing which move it is, the piece moved and the positions involved.

Shown right, AFTER the pawn captures the black pawn, the move box is updated to show the capture too.

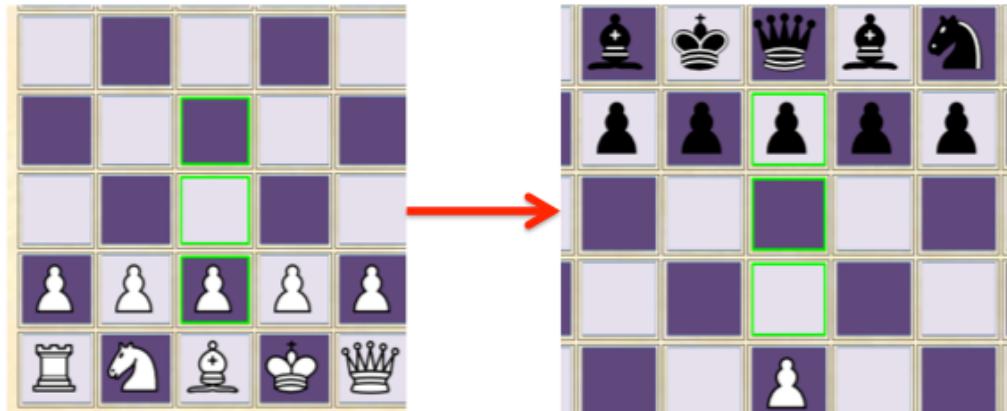


6)

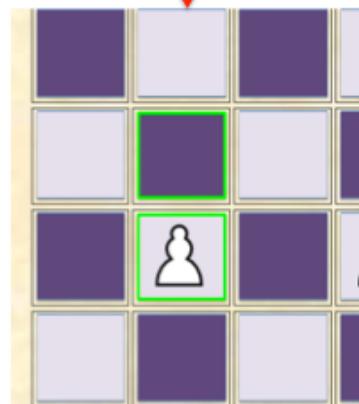
**TEST 6 (Passed) :**

When the game starts, white should move first regardless of the positioning of the pieces. Shown above, the system makes white move first – it is player 2's turn as they are the white pieces – when a black piece is clicked, nothing happens

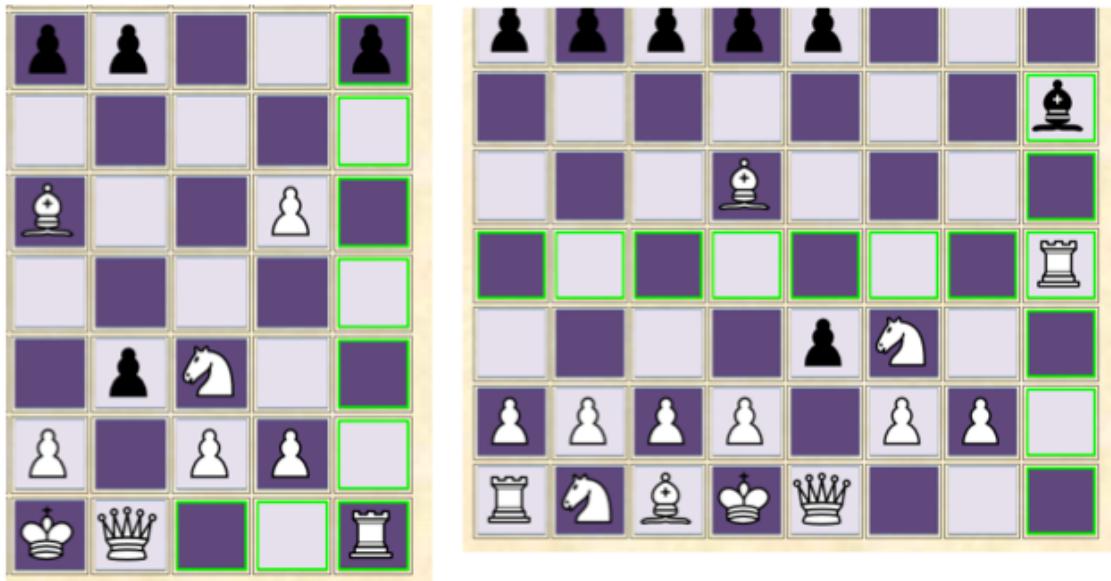
7)

**TEST 7 (Passed) :**

The pawns can move two spaces on their first move which is shown to be working with the highlighted squares. Subsequently, the black pieces are exactly the same, the pawn pieces behave the same. Then after the first move, the pawns only move one space. These show that the movement of the pawn is correct and working (this test doesn't include capturing, this is a separate test).

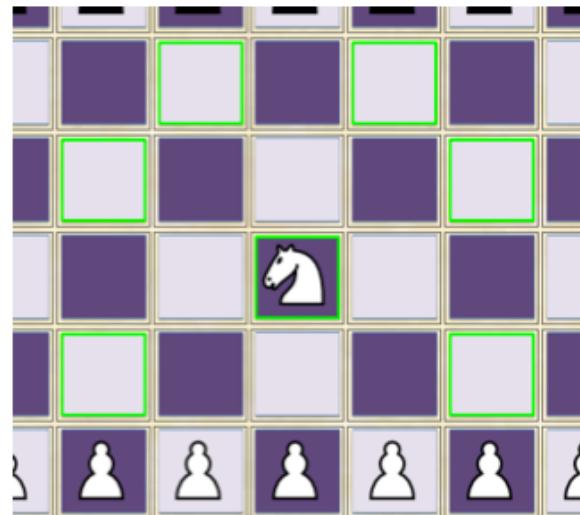


8)

**TEST 8 (Passed) :**

As shown above, the rook is seen to move correctly – only horizontal movement is allowed and this is shown by the highlighted squares. Having the highlighting tested before hand makes the movement of pieces easier to check, using the highlight as a guide. Diagonal squares are not highlighted and all horizontal squares are highlighted as far as the rook may move.

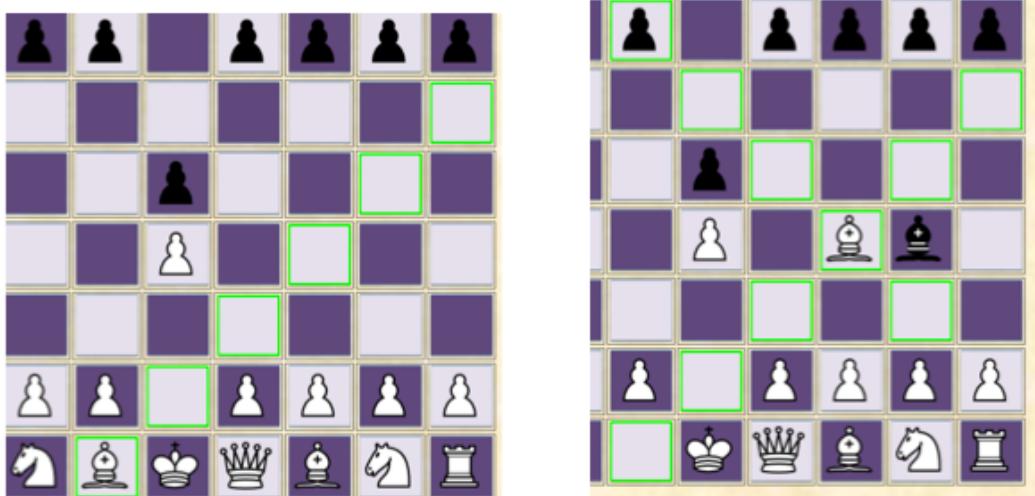
9)

**TEST 9 (Passed) :**

The knight only moves in an “L” shape, which can be in any direction – as explained in the algorithm designs, this is done mathematically by checking the change in x and the change in y.

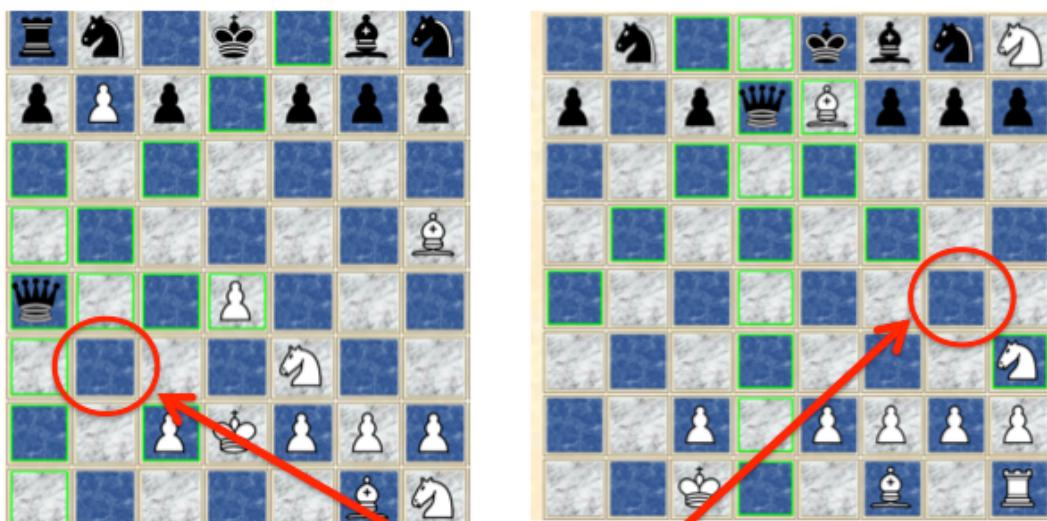
Shown above, the movement of the knight is correct as all possible moves are highlighted and these are valid moves.

10)

**TEST 10 (Passed) :**

A bishops movement should only diagonal which is shown to be true above. The bishop cannot move further in the bottom right direction as the square is occupied by a friendly piece, but it may move to the square occupied by the black pawn, as it may capture this piece. These screenshots show that the movement of the bishop is running correctly as it should.

11)

**TEST 11 (Failed) :**

As shown above, the Queens movement is all fine except there is a problem involving its diagonal movement, here two scenarios are shown, showing a square that does not highlight and as a result the queen cannot be moved to these squares, despite them being valid moves. The white pieces occupying the squares one ahead, however, can still be captured. Horizontal Movement is fine, as seen. The problem lies with the queens diagonal movement, specifically, its downwards, right diagonal movement.

# Test 11 - Debugging

## TEST 11 (Failed) :

As shown left, when debugging, when checking the **board** object property of the **game** class, and looking at the array of squares representing the board, I can see that the square not highlighted, has its **occupied** property set to false – which is correct, so this is not the problem.

[4, 4]	{test_chess.Square}	Cannot eval
[4, 5]	{test_chess.Square}	
[4, 6]	{test_chess.Square}	
[4, 7]	{test_chess.Square}	
m_occ	false	Type
m_X	5	test_chess
m_Y	5	

The problem, therefore must lie within the **checkValidMove** procedure for the queen – which returns true or false for a given square for the possible moves...

```

for (int i = Math.Abs((targetXindex - currentXindex )); i > 0; i--)
{
    currentXindex = currentXindex - 1;
    currentYindex = currentYindex + 1;
    if (currentYindex != 8 && currentXindex >= 0 )
    {
        if (board[currentXindex, currentYindex].occ == true)
        {
            if (currentXindex == targetXindex && currentYindex == targetYindex)
            {
                if (canCapture(board[currentXindex, currentYindex], gameBoard) == true)
                {
                    return true;
                }
            }
            return false;
        }
    }
}

if ((targetXindex - currentXindex) > 0 && (targetYindex - currentYindex) < 0)
{
    for (int i = Math.Abs((targetXindex - currentXindex)); i > 0; i--)
    {
}
}

```

Shown above is the IF and FOR loop procedures for Two directions of diagonal movement. The FOR loop within each of the 4 diagonal movement blocks of code is the same, it using the x coordinate of the final and current position as a loop counter. As shown, the upper bound is “i > 0” which scans the subsequent squares between the current and target position.

```

        }
        return false;
    }
}

if ((targetXindex - currentXindex) > 0 && (targetYindex - currentYindex) > 0)
{
    for (int i = Math.Abs((targetXindex - currentXindex)); i >= 0; i--)
    {
        currentXindex = currentXindex + 1;
        currentYindex = currentYindex + 1;
        if (currentXindex != 8 && currentYindex != 8 )
        {
            if (board[currentXindex, currentYindex].occ == true)
            {
                if (currentXindex == targetXindex && currentYindex == targetYindex)
                {
                    if (canCapture(board[currentXindex, currentYindex], gameBoard) == true)
                    {
                        return true;
                    }
                }
            }
        }
    }
}

```

Looking at the section of code for downwards and right diagonal movement, it is seen that the upper bound of the loop is “ $i \geq 0$ ” which is one more than it should be, so this is what needs to be changed so it functions properly. (otherwise it scans the current position as well and determines this as occupied and then returns false).



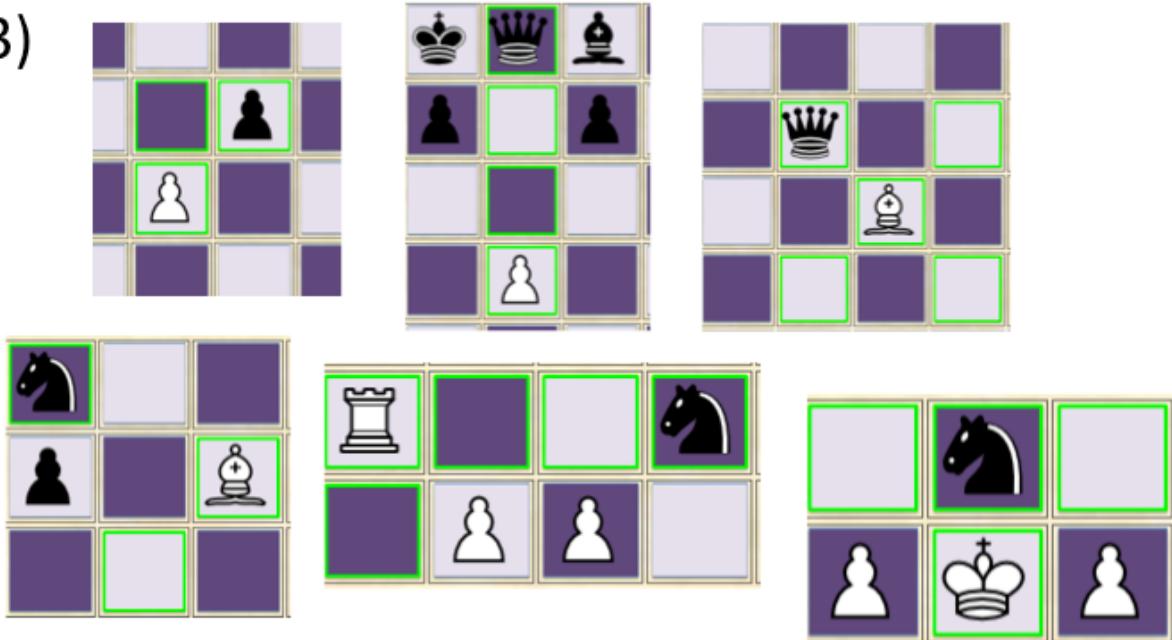
After the changes had been made, the queen movement now functioned correctly. To make the test thorough, many different scenarios were looked at, not just the same piece positions – just to ensure that all the movement is indeed working correctly.

12)

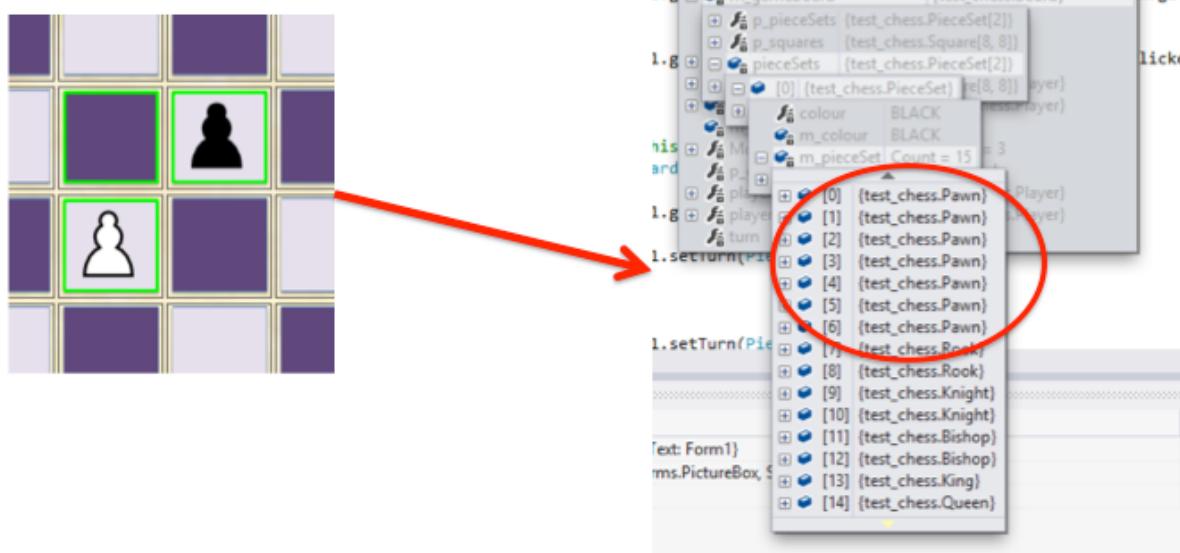
**TEST 12 (Passed) :**

As shown above, the kings movement is working correctly. Only one space in any direction is allowed. Occupied squares are also dealt with in the **checkValidMove** algorithm and this translates to no highlighting, which works in the test run for the king.

13)

**TEST 13 (Passed) :**

As shown above, All pieces correctly show valid capturing moves and whenever an enemy piece is in the line of movement of a friendly piece, the algorithm always returns true to being able to capture that piece.



To further test that the capturing procedure has executed correctly, I inspected the population of the `pieceSet` object within the `gameboard` property of the `game` class. As shown, there is now only 7 pawn objects in the list and a count of 15 pieces instead of 16. This shows that the pawn piece was successfully removed. (But stored in the `move` object which is appending to the games move history list property)

14)

**TEST 14 (Passed) :**

The king has a function called `canCastle` which determines whether the king may perform the castling move or not. In this procedure the kingside rook is also moved to perform the move. As shown above, when the king is clicked, the second square on the left is highlighted indicating that the king may castle and when that square is clicked (right) the move is performed and the pieces are interchanged.

This shows the castling move working correctly – this also proves the `canCastle` algorithm to be working too.

15)

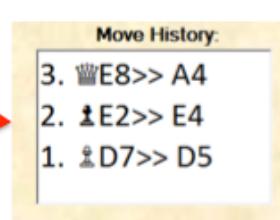
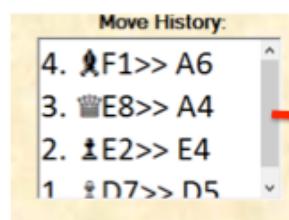
**TEST 15 (Passed) :**

When the pawn reaches the other side and captures the rook, the promote pawn form is launched, prompting a user selection. When the queen piece is chosen and the promote button is clicked, the form closes and the pawn piece changes immediately into a queen piece successfully.

This works correctly for all promotions and other scenarios – this test was carried out for multiple to ensure the system is robust.

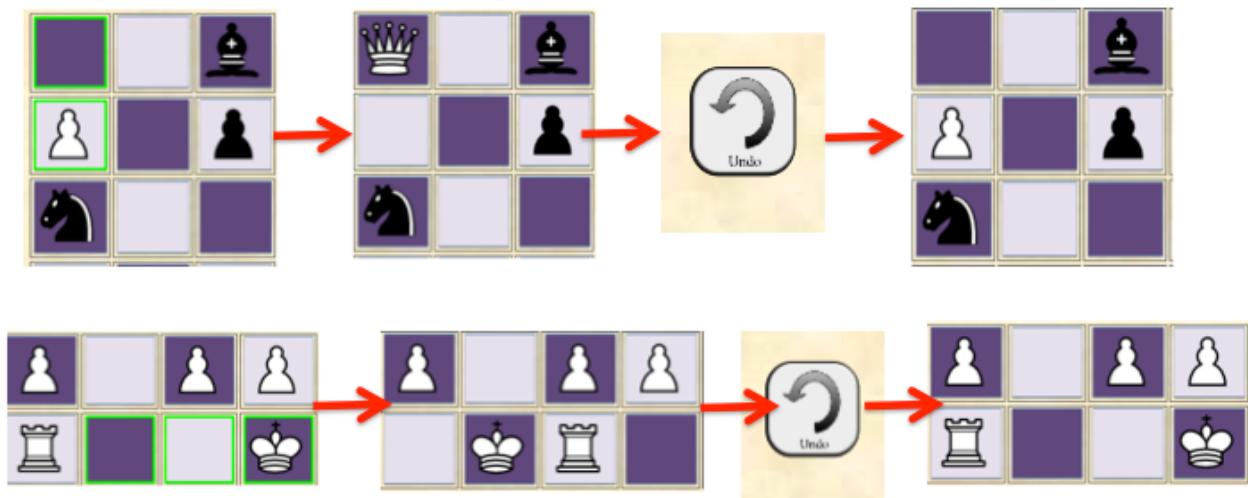


16)

**TEST 16 (Passed) :**

When a move is made, the move history box is appended with the move details. When the undo button is pressed, the move retracted immediately, and the move box removed the move details successfully as shown. This test had the same results for all other variations of moving. The player who moved regains the turn of the game and can move again - The undo button works as it is meant to.

17)

**TEST 17 (Passed) :**

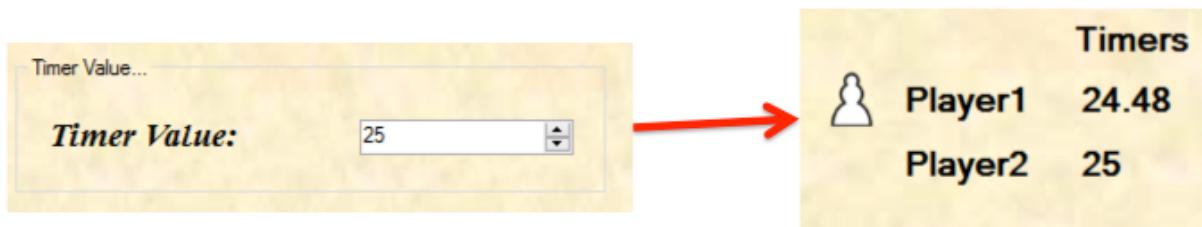
Showing the promoting pawn and castling moves above, the undo function works for both special cases. When the promoting pawn is undone and the pawn is moved back again, another selection is prompted, as if the move never happened. The king, also, may castle again after the move has been retracted.

18)

**TEST 18 (Passed) :**

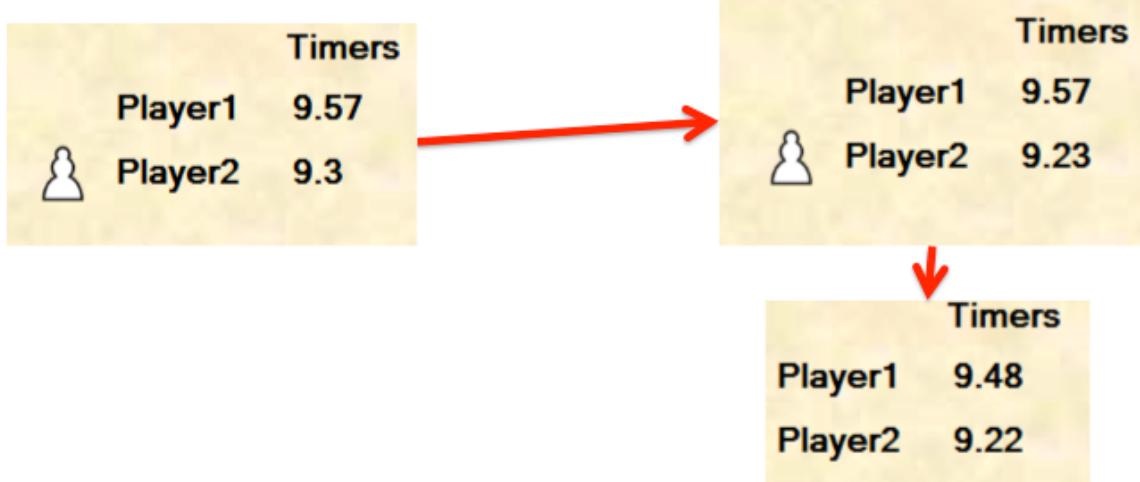
This test was implied in previous tests, but the system works correctly for all selective styles. The style is selected at the main menu and when the game board loads, the squares and all functionality (highlighting) is corresponding to the style of board chosen.

19)

**TEST 19 (Passed) :**

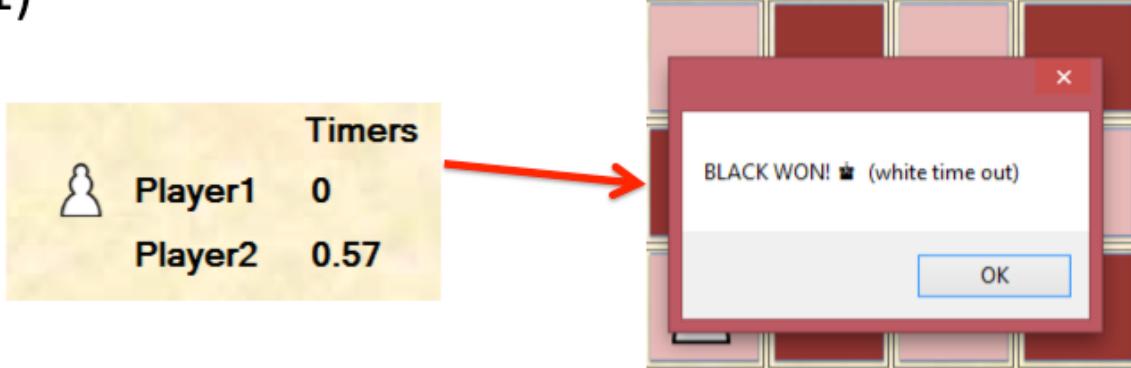
The value of the timer selected in the main menu should correctly be translated into the game. Here it is shown this works as it should, with player 1's timer counting down instantly from 25 minutes.

20)

**TEST 20 (Passed) :**

It is player 2's turn and while they are taking their turn, player 1's timer is paused as shown. Then after player 2 is done, player 1's timer starts again.

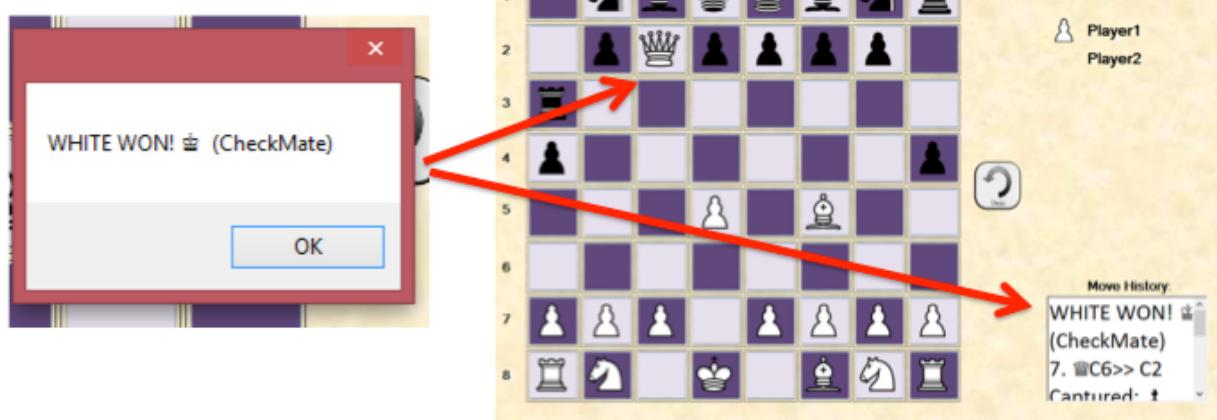
21)

**TEST 21 (Passed) :**

Player 1's timer fell to zero and as a result, a message box was shown indicating that black has therefore won the game due to a timeout.

Also the board disables and the user cannot click or move pieces anymore as the game has ended. The undo button is also disabled.

22)

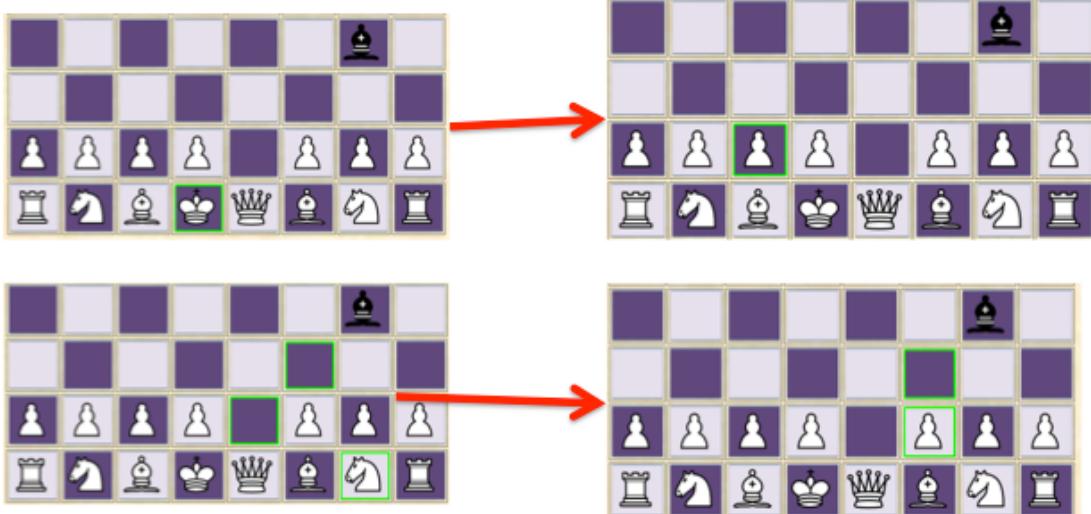
**TEST 22 (Passed) :**

As shown, white has checkmated black. When the final move was made, the message box appeared letting the user know that white won due to check mate. Also as further confirmation, the move list box was updated to show the result of the game.

As with the zero-timer situation, the board itself is disabled and also the undo button to prevent the game being messed with.

The check mate algorithm therefore works as it should, as demonstrated by this test.

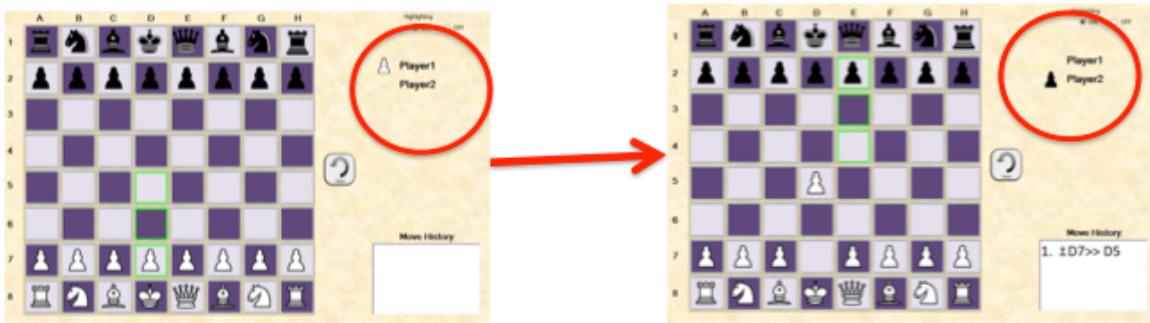
23)

**TEST 23 (Passed) :**

As shown, The king has been put into check by the black bishop. The king cannot move as it is trapped, when the king is clicked, no possible moves show up – which is correct.

Also, any other other piece that is clicked that cannot be move (ie the pawn in the top right picture) has no highlighted possible moves and the user cannot move that piece. The game does however, allow other pieces to be moved, if the result of that move would mean the king is out of check ( bottom two pictures) – this uses the **temporary move** procedure and the **isInCheck** procedure –This test shows that these work correctly.

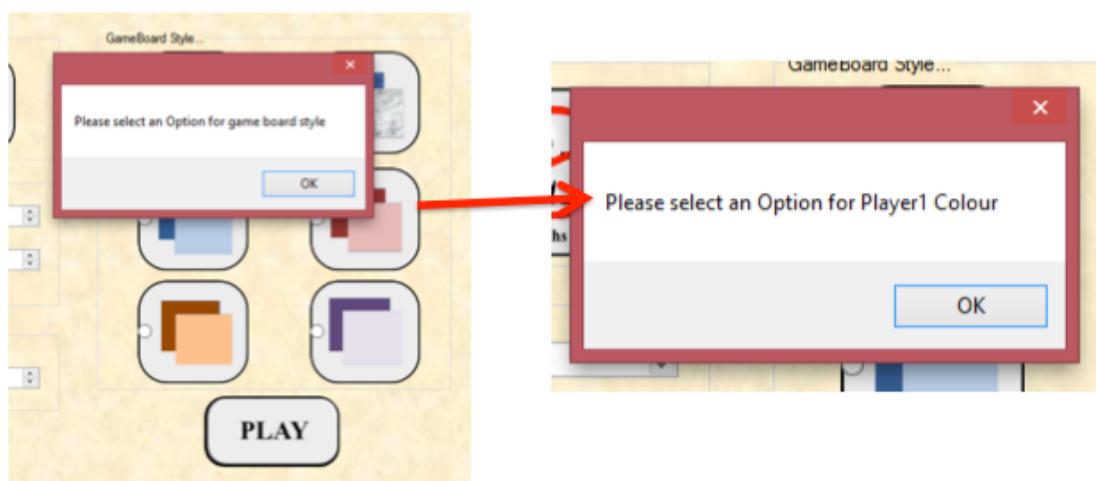
24)

**TEST 24 (Passed) :**

It is initially player 1's move and the chess icon is next to the player 1 label in white, to represent the colour of player 1's pieces. After the player has made their move, the icon fluctuates, as shown, to the player 2 label and changes colour to represent the colour pieces of player 2.

This changing of the icon every turn therefore works correctly.

25)

**TEST 25 (Passed) :**

The main menu needed to handle any unselected options and tell the user what they haven't selected. As shown this works correctly, if the user doesn't select a particular option a warning is given as a message box saying which they need to select.

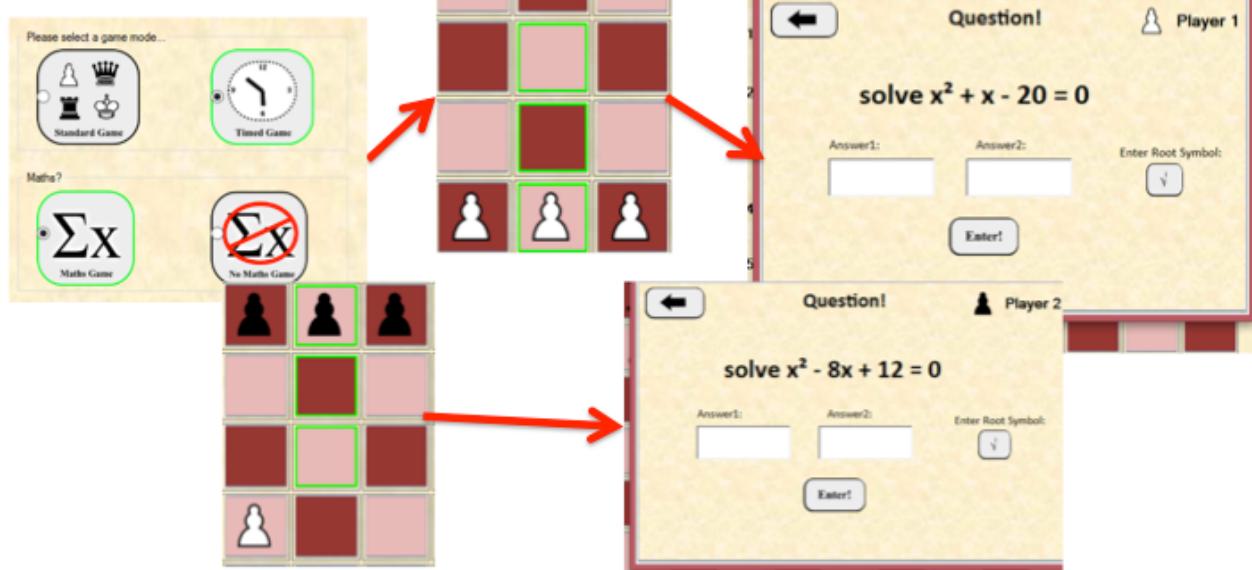
This is determined when the play button is clicked and if an option hasn't been selected, the game board form does not load up, only when all sufficient options are selected.

26)

**TEST 26 (Passed) :**

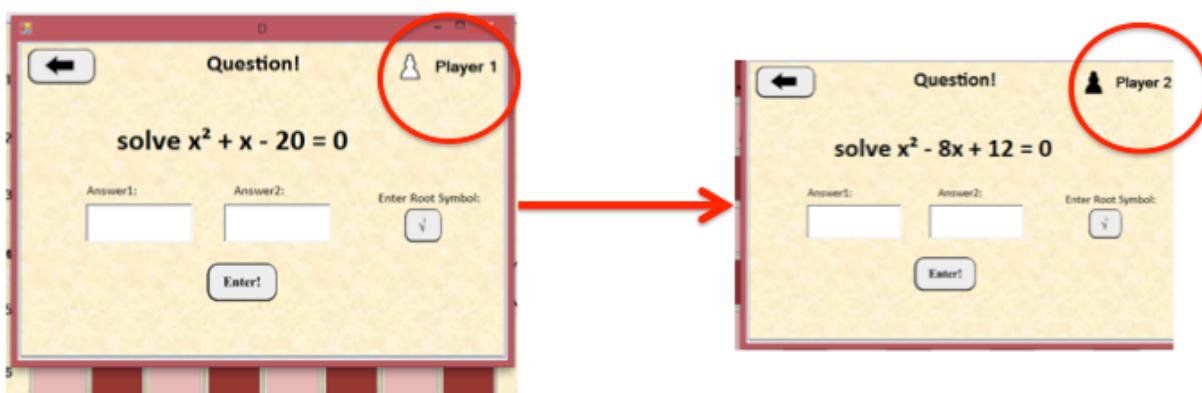
The numerical choices on the main menu form should be certain values only – so the user should not be able to manually change the value. This is shown above, when trying to edit the value of 15, the control stops any editing. The only changes made are those using the up and down arrows which change the values as they should – in steps of 5.

27)

**TEST 27 (Passed) :**

With the maths mode selected, it is seen that with every move, a question is presented to the user correctly. The piece is clicked first then after the piece is move, the question form appears with focus and prompts a user answer.

28)

**TEST 28 (Passed) :**

As shown, when the players make a move, a question is presented to them. In the top right of the form is an indicator to tell who's question it is for, depending on who's turn it is in the game. The first move was player 1 which correctly shows up on the question form and then the second move was by player 2 and the changes are made correctly and the icon shows that it is now player 2's question.

29)

The screenshot shows two windows side-by-side. The left window is titled 'Question!' and is labeled 'Player 1'. It contains a question: 'Find n:  $(a^{23} \div a^{17})^4 = a^n$ '. Below the question are input fields for 'Answer1' and 'Enter Root Symbol'. The right window is titled 'Question!' and is labeled 'Player 2'. It contains a question: 'simplify  $\sqrt{8} \div 2$ '. Below the question are input fields for 'Answer1' and 'Enter Root Symbol'. A red arrow points from the Player 1 screen to a text file on the right. The text file lists various mathematical operations and their results, with the last parameter of each line indicating the difficulty level. Two specific lines are circled in red: 'Find n:  $(a^9 \times a^3)^{12} = a^n, 144,,2$ ' and 'simplify  $\sqrt{8} \div 2, 2\sqrt{2},,1$ . The circled numbers 144 and 2 correspond to the difficulty levels shown in the Player 1 and Player 2 screens respectively.

```

Find n: (a9 x a3)12 = an, 144,,2
Find n: (a1 x a-1)4 = an, 0,,2
Find n: (a2 x a13)4 = an, 60,,2
Find n: (a1 + a-1)4 = an, 8,,2
Find n: (a3 + a13)4 = an, -40,,2
Find n: (a22 + a11)4 = an, 44,,2
Find n: (a3 + a1)4 = an, -40,,2
Find n: (a23 + a17)4 = an, 24,,2
Find n: (a50 + a14) = an, 36,,2
Find n: (a5 + a15)2 = an, -20,,2
Find n: (a2 + a11)2 = an, -18,,2
simplify √76 , 2√19,,3
simplify √68 , 2√17,,3
simplify √56 , 2√14,,3

```

```

solve x2 + 6x + 8 = 0, -4, -2, 1
simplify √8, 2√2,,1
simplify √4, 2,,1
simplify √12, 2√3,,1
simplify √20, 2√5,,1
simplify √8 + 2, √2,,1
simplify √4 + 2, 1,,1
simplify √12 + 2, √3,,1
simplify √20 + 2, √5,,1
simplify √50, 5√2,,1
simplify √18, 3√2,,1
simplify √50 + 5, √2,,1
simplify √24, 2√6,,1

```

**TEST 29 (Passed) :**

At the main menu, player 1 selects level 2 and player 2 selects level 1. When their questions are presented to them, they are of the correct difficulty. To show this, a reference to the text file is shown and it can be seen that the question presented is of the correct difficulty which is the last parameter of each line in the file.

30)

The screenshot shows two windows side-by-side. The left window is titled 'Question!' and is labeled 'Player 2'. It contains a question: 'simplify  $\sqrt{8} \div 2$ '. Below the question are input fields for 'Answer1' and 'Enter Root Symbol'. The right window is titled 'Question!' and is labeled 'Player 1'. It contains a question: 'solve  $x^2 + x - 20 = 0$ '. Below the question are input fields for 'Answer1' and 'Answer2'. A red arrow points from the Player 2 screen to the Player 1 screen. Both the 'Answer1' and 'Answer2' fields in the Player 1 screen are circled in red, indicating they are both active or visible for this question type.

**TEST 30 (Passed) :**

In the system, some questions only have 1 answer and others have two (i.e quadratics). If the question only has one answer then two answer boxes should not be shown as this would confuse the player. As shown above, this works correctly and the second answer box is made invisible to the user for particular questions.

This also further adds confidence in the working of the question objects used, as the properties of the question objects are used to determine whether the question has a second answer or not.

31)

Question!

Player 1

solve  $x^2 + 4x + 4 = 0$

Answer1:  Enter Root Symbol:

Enter!

**TEST 31 (Passed) :**

The user input is fully validated by providing the user with buttons for certain symbols where necessary, this reduces a lot of potential problems.

As the input is string based, any invalid characters will simply be compared as a string and when that string is not the same as the answer then it will return incorrect.

With the admin section, again it is string based and it is down to the user to enter the questions they wish in this process.

32/33)

Question!

Player 2

CORRECT!

Answer1:  Enter Root Symbol:

Enter!

Question!

Player 1

INCORRECT!

Answer1:  Answer2:  Enter Root Symbol:

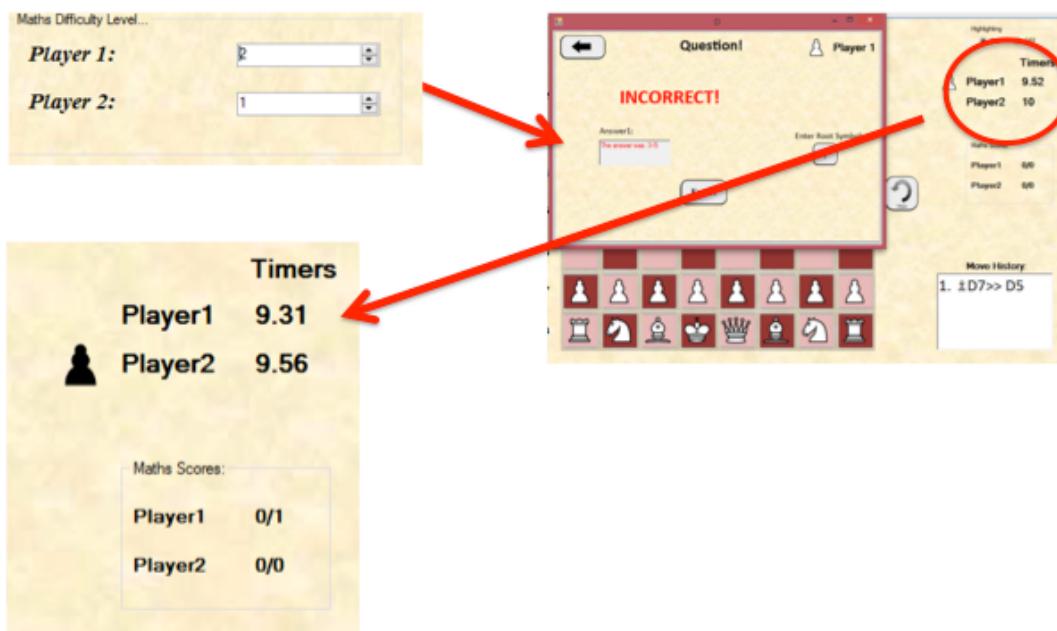
Enter!

**TEST 32/33 (Passed) :**

As shown above, when the user clicks enter to submit their answer, their inputs are processed and determined whether they are correct or incorrect. This is given a feedback in the main question label. If the answer is correct then the label states this in green and if incorrect, the label is red to symbolize a wrong answer. The solutions are always written to the answer boxes in green to symbolize the correct answer and correct solution, even if they got it right.

This works every time a question is answered, so this function works.

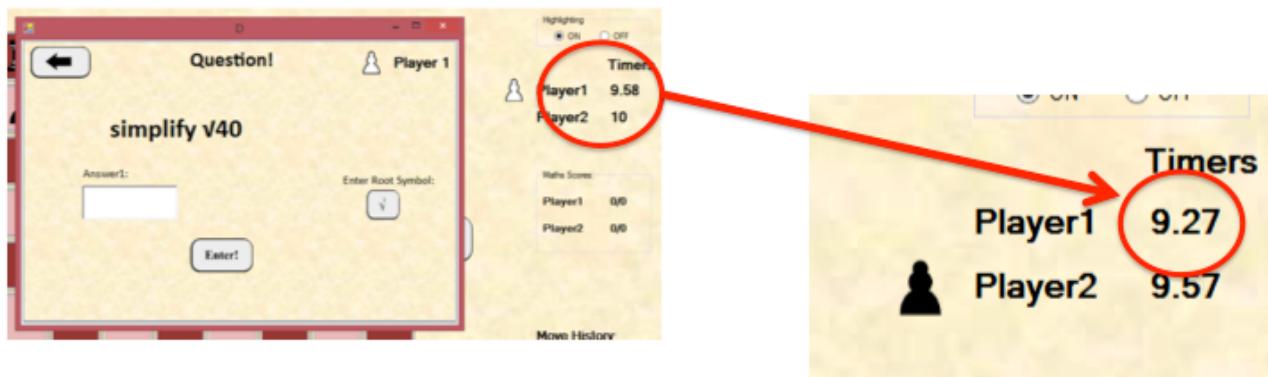
34)

**TEST 34 (Passed) :**

Depending on the difficulty chosen by the player, varying time should be deducted for an incorrect answer. Shown above is for player one who selected level 2 difficulty – therefore 20 seconds should be deducted, which worked correctly.

The same works correctly but for the other levels of difficulties – 30 seconds for level 1 and 10 seconds for level 3.

35)

**TEST 35 (Passed) :**

If the user simply exits the question form without answering then time must still be subtracted as if it was an incorrect answer. As shown, this applies correctly during gameplay. The form was exited and 30 seconds was deducted as the player was on level 1 difficulty questions.

36)

**INCORRECT!**

Answer1: The answer was: -7  
Answer2: The answer was: -1

**CORRECT!**

Answer1: The answer was:  $2\sqrt{10}$

Maths Scores:	
Player1	0/1
Player2	0/0

Maths Scores:	
Player1	0/1
Player2	1/1

**TEST 36 (Passed) :**

In the top screenshots, player 1 has answered incorrectly and as seen, their score is updated to 0/1. Crucially, player 2's score stays out of 0 because they haven't had any questions yet. When player 2 answers a question correctly (bottom screenshots) their score is updated to 1/1.

This scoring works correctly and carries on throughout the game – keeping track of each players score separately and the total being specific to how many questions that particular player has been asked at a point in the game.

37)

**Question 1** Player 1

Find  $n$ :  $a^9 \div a^9 = a^n$

Answer:   
Enter Root Symbol:  $\sqrt[n]{ }$

**CORRECT!**

Answer: The answer was: 0  
Enter Root Symbol:  $\sqrt[n]{ }$

Enter!

Chessboard: A8, B8, C8, D8, E8, F8, G8, H8, A7, B7, C7, D7, E7, F7, G7, H7, A6, B6, C6, D6, E6, F6, G6, H6, A5, B5, C5, D5, E5, F5, G5, H5, A4, B4, C4, D4, E4, F4, G4, H4, A3, B3, C3, D3, E3, F3, G3, H3, A2, B2, C2, D2, E2, F2, G2, H2, A1, B1, C1, D1, E1, F1, G1, H1

Timers: Player1 9.58, Player2 9.57

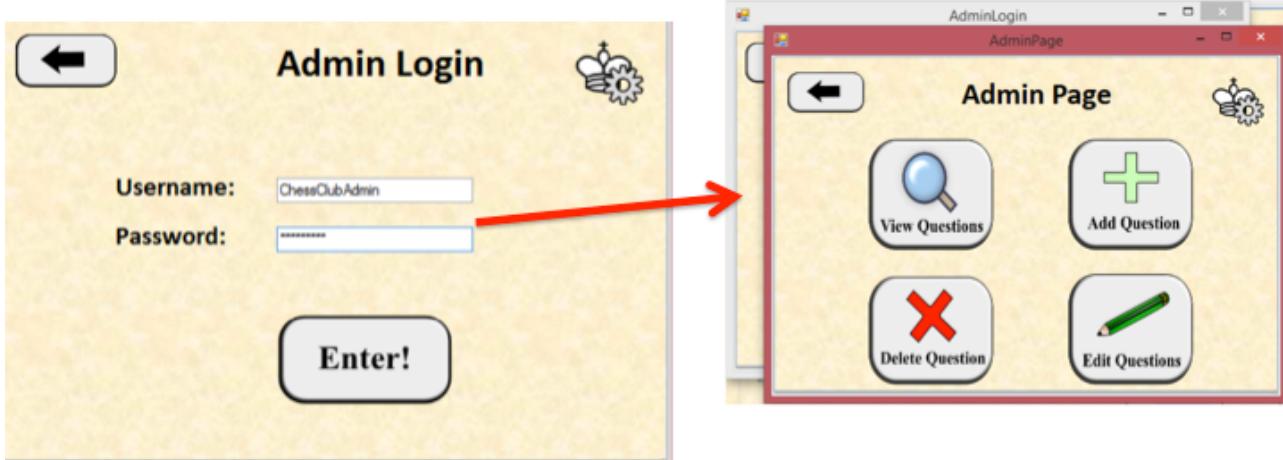
Move History: 1. e4 d5 2. Nf3 Nf6 3. c4 e6 4. Nc3 Be7 5. Bb5 a6 6. Ba4 Nc6 7. O-O b5 8. Bb3 Nf6 9. d5 Nxd5 10. Nxd5 Bb7 11. Nc3 O-O 12. f4 Ne4 13. Nxe4 Bxe4 14. Qd2 Nc6 15. Rfd1 Nf6 16. Nc3 Nc6 17. Nxe4 Bxe4 18. Qd2 Nc6 19. Rfd1 Nf6 20. Nc3 Nc6 21. Nxe4 Bxe4 22. Qd2 Nc6 23. Rfd1 Nf6 24. Nc3 Nc6 25. Nxe4 Bxe4 26. Qd2 Nc6 27. Rfd1 Nf6 28. Nc3 Nc6 29. Nxe4 Bxe4 30. Qd2 Nc6 31. Rfd1 Nf6 32. Nc3 Nc6 33. Nxe4 Bxe4 34. Qd2 Nc6 35. Rfd1 Nf6 36. Nc3 Nc6 37. Nxe4 Bxe4 38. Qd2 Nc6 39. Rfd1 Nf6 40. Nc3 Nc6 41. Nxe4 Bxe4 42. Qd2 Nc6 43. Rfd1 Nf6 44. Nc3 Nc6 45. Nxe4 Bxe4 46. Qd2 Nc6 47. Rfd1 Nf6 48. Nc3 Nc6 49. Nxe4 Bxe4 50. Qd2 Nc6 51. Rfd1 Nf6 52. Nc3 Nc6 53. Nxe4 Bxe4 54. Qd2 Nc6 55. Rfd1 Nf6 56. Nc3 Nc6 57. Nxe4 Bxe4 58. Qd2 Nc6 59. Rfd1 Nf6 60. Nc3 Nc6 61. Nxe4 Bxe4 62. Qd2 Nc6 63. Rfd1 Nf6 64. Nc3 Nc6 65. Nxe4 Bxe4 66. Qd2 Nc6 67. Rfd1 Nf6 68. Nc3 Nc6 69. Nxe4 Bxe4 70. Qd2 Nc6 71. Rfd1 Nf6 72. Nc3 Nc6 73. Nxe4 Bxe4 74. Qd2 Nc6 75. Rfd1 Nf6 76. Nc3 Nc6 77. Nxe4 Bxe4 78. Qd2 Nc6 79. Rfd1 Nf6 80. Nc3 Nc6 81. Nxe4 Bxe4 82. Qd2 Nc6 83. Rfd1 Nf6 84. Nc3 Nc6 85. Nxe4 Bxe4 86. Qd2 Nc6 87. Rfd1 Nf6 88. Nc3 Nc6 89. Nxe4 Bxe4 90. Qd2 Nc6 91. Rfd1 Nf6 92. Nc3 Nc6 93. Nxe4 Bxe4 94. Qd2 Nc6 95. Rfd1 Nf6 96. Nc3 Nc6 97. Nxe4 Bxe4 98. Qd2 Nc6 99. Rfd1 Nf6 100. Nc3 Nc6 101. Nxe4 Bxe4 102. Qd2 Nc6 103. Rfd1 Nf6 104. Nc3 Nc6 105. Nxe4 Bxe4 106. Qd2 Nc6 107. Rfd1 Nf6 108. Nc3 Nc6 109. Nxe4 Bxe4 110. Qd2 Nc6 111. Rfd1 Nf6 112. Nc3 Nc6 113. Nxe4 Bxe4 114. Qd2 Nc6 115. Rfd1 Nf6 116. Nc3 Nc6 117. Nxe4 Bxe4 118. Qd2 Nc6 119. Rfd1 Nf6 120. Nc3 Nc6 121. Nxe4 Bxe4 122. Qd2 Nc6 123. Rfd1 Nf6 124. Nc3 Nc6 125. Nxe4 Bxe4 126. Qd2 Nc6 127. Rfd1 Nf6 128. Nc3 Nc6 129. Nxe4 Bxe4 130. Qd2 Nc6 131. Rfd1 Nf6 132. Nc3 Nc6 133. Nxe4 Bxe4 134. Qd2 Nc6 135. Rfd1 Nf6 136. Nc3 Nc6 137. Nxe4 Bxe4 138. Qd2 Nc6 139. Rfd1 Nf6 140. Nc3 Nc6 141. Nxe4 Bxe4 142. Qd2 Nc6 143. Rfd1 Nf6 144. Nc3 Nc6 145. Nxe4 Bxe4 146. Qd2 Nc6 147. Rfd1 Nf6 148. Nc3 Nc6 149. Nxe4 Bxe4 150. Qd2 Nc6 151. Rfd1 Nf6 152. Nc3 Nc6 153. Nxe4 Bxe4 154. Qd2 Nc6 155. Rfd1 Nf6 156. Nc3 Nc6 157. Nxe4 Bxe4 158. Qd2 Nc6 159. Rfd1 Nf6 160. Nc3 Nc6 161. Nxe4 Bxe4 162. Qd2 Nc6 163. Rfd1 Nf6 164. Nc3 Nc6 165. Nxe4 Bxe4 166. Qd2 Nc6 167. Rfd1 Nf6 168. Nc3 Nc6 169. Nxe4 Bxe4 170. Qd2 Nc6 171. Rfd1 Nf6 172. Nc3 Nc6 173. Nxe4 Bxe4 174. Qd2 Nc6 175. Rfd1 Nf6 176. Nc3 Nc6 177. Nxe4 Bxe4 178. Qd2 Nc6 179. Rfd1 Nf6 180. Nc3 Nc6 181. Nxe4 Bxe4 182. Qd2 Nc6 183. Rfd1 Nf6 184. Nc3 Nc6 185. Nxe4 Bxe4 186. Qd2 Nc6 187. Rfd1 Nf6 188. Nc3 Nc6 189. Nxe4 Bxe4 190. Qd2 Nc6 191. Rfd1 Nf6 192. Nc3 Nc6 193. Nxe4 Bxe4 194. Qd2 Nc6 195. Rfd1 Nf6 196. Nc3 Nc6 197. Nxe4 Bxe4 198. Qd2 Nc6 199. Rfd1 Nf6 200. Nc3 Nc6 201. Nxe4 Bxe4 202. Qd2 Nc6 203. Rfd1 Nf6 204. Nc3 Nc6 205. Nxe4 Bxe4 206. Qd2 Nc6 207. Rfd1 Nf6 208. Nc3 Nc6 209. Nxe4 Bxe4 210. Qd2 Nc6 211. Rfd1 Nf6 212. Nc3 Nc6 213. Nxe4 Bxe4 214. Qd2 Nc6 215. Rfd1 Nf6 216. Nc3 Nc6 217. Nxe4 Bxe4 218. Qd2 Nc6 219. Rfd1 Nf6 220. Nc3 Nc6 221. Nxe4 Bxe4 222. Qd2 Nc6 223. Rfd1 Nf6 224. Nc3 Nc6 225. Nxe4 Bxe4 226. Qd2 Nc6 227. Rfd1 Nf6 228. Nc3 Nc6 229. Nxe4 Bxe4 230. Qd2 Nc6 231. Rfd1 Nf6 232. Nc3 Nc6 233. Nxe4 Bxe4 234. Qd2 Nc6 235. Rfd1 Nf6 236. Nc3 Nc6 237. Nxe4 Bxe4 238. Qd2 Nc6 239. Rfd1 Nf6 240. Nc3 Nc6 241. Nxe4 Bxe4 242. Qd2 Nc6 243. Rfd1 Nf6 244. Nc3 Nc6 245. Nxe4 Bxe4 246. Qd2 Nc6 247. Rfd1 Nf6 248. Nc3 Nc6 249. Nxe4 Bxe4 250. Qd2 Nc6 251. Rfd1 Nf6 252. Nc3 Nc6 253. Nxe4 Bxe4 254. Qd2 Nc6 255. Rfd1 Nf6 256. Nc3 Nc6 257. Nxe4 Bxe4 258. Qd2 Nc6 259. Rfd1 Nf6 260. Nc3 Nc6 261. Nxe4 Bxe4 262. Qd2 Nc6 263. Rfd1 Nf6 264. Nc3 Nc6 265. Nxe4 Bxe4 266. Qd2 Nc6 267. Rfd1 Nf6 268. Nc3 Nc6 269. Nxe4 Bxe4 270. Qd2 Nc6 271. Rfd1 Nf6 272. Nc3 Nc6 273. Nxe4 Bxe4 274. Qd2 Nc6 275. Rfd1 Nf6 276. Nc3 Nc6 277. Nxe4 Bxe4 278. Qd2 Nc6 279. Rfd1 Nf6 280. Nc3 Nc6 281. Nxe4 Bxe4 282. Qd2 Nc6 283. Rfd1 Nf6 284. Nc3 Nc6 285. Nxe4 Bxe4 286. Qd2 Nc6 287. Rfd1 Nf6 288. Nc3 Nc6 289. Nxe4 Bxe4 290. Qd2 Nc6 291. Rfd1 Nf6 292. Nc3 Nc6 293. Nxe4 Bxe4 294. Qd2 Nc6 295. Rfd1 Nf6 296. Nc3 Nc6 297. Nxe4 Bxe4 298. Qd2 Nc6 299. Rfd1 Nf6 300. Nc3 Nc6 301. Nxe4 Bxe4 302. Qd2 Nc6 303. Rfd1 Nf6 304. Nc3 Nc6 305. Nxe4 Bxe4 306. Qd2 Nc6 307. Rfd1 Nf6 308. Nc3 Nc6 309. Nxe4 Bxe4 310. Qd2 Nc6 311. Rfd1 Nf6 312. Nc3 Nc6 313. Nxe4 Bxe4 314. Qd2 Nc6 315. Rfd1 Nf6 316. Nc3 Nc6 317. Nxe4 Bxe4 318. Qd2 Nc6 319. Rfd1 Nf6 320. Nc3 Nc6 321. Nxe4 Bxe4 322. Qd2 Nc6 323. Rfd1 Nf6 324. Nc3 Nc6 325. Nxe4 Bxe4 326. Qd2 Nc6 327. Rfd1 Nf6 328. Nc3 Nc6 329. Nxe4 Bxe4 330. Qd2 Nc6 331. Rfd1 Nf6 332. Nc3 Nc6 333. Nxe4 Bxe4 334. Qd2 Nc6 335. Rfd1 Nf6 336. Nc3 Nc6 337. Nxe4 Bxe4 338. Qd2 Nc6 339. Rfd1 Nf6 340. Nc3 Nc6 341. Nxe4 Bxe4 342. Qd2 Nc6 343. Rfd1 Nf6 344. Nc3 Nc6 345. Nxe4 Bxe4 346. Qd2 Nc6 347. Rfd1 Nf6 348. Nc3 Nc6 349. Nxe4 Bxe4 350. Qd2 Nc6 351. Rfd1 Nf6 352. Nc3 Nc6 353. Nxe4 Bxe4 354. Qd2 Nc6 355. Rfd1 Nf6 356. Nc3 Nc6 357. Nxe4 Bxe4 358. Qd2 Nc6 359. Rfd1 Nf6 360. Nc3 Nc6 361. Nxe4 Bxe4 362. Qd2 Nc6 363. Rfd1 Nf6 364. Nc3 Nc6 365. Nxe4 Bxe4 366. Qd2 Nc6 367. Rfd1 Nf6 368. Nc3 Nc6 369. Nxe4 Bxe4 370. Qd2 Nc6 371. Rfd1 Nf6 372. Nc3 Nc6 373. Nxe4 Bxe4 374. Qd2 Nc6 375. Rfd1 Nf6 376. Nc3 Nc6 377. Nxe4 Bxe4 378. Qd2 Nc6 379. Rfd1 Nf6 380. Nc3 Nc6 381. Nxe4 Bxe4 382. Qd2 Nc6 383. Rfd1 Nf6 384. Nc3 Nc6 385. Nxe4 Bxe4 386. Qd2 Nc6 387. Rfd1 Nf6 388. Nc3 Nc6 389. Nxe4 Bxe4 390. Qd2 Nc6 391. Rfd1 Nf6 392. Nc3 Nc6 393. Nxe4 Bxe4 394. Qd2 Nc6 395. Rfd1 Nf6 396. Nc3 Nc6 397. Nxe4 Bxe4 398. Qd2 Nc6 399. Rfd1 Nf6 400. Nc3 Nc6 401. Nxe4 Bxe4 402. Qd2 Nc6 403. Rfd1 Nf6 404. Nc3 Nc6 405. Nxe4 Bxe4 406. Qd2 Nc6 407. Rfd1 Nf6 408. Nc3 Nc6 409. Nxe4 Bxe4 410. Qd2 Nc6 411. Rfd1 Nf6 412. Nc3 Nc6 413. Nxe4 Bxe4 414. Qd2 Nc6 415. Rfd1 Nf6 416. Nc3 Nc6 417. Nxe4 Bxe4 418. Qd2 Nc6 419. Rfd1 Nf6 420. Nc3 Nc6 421. Nxe4 Bxe4 422. Qd2 Nc6 423. Rfd1 Nf6 424. Nc3 Nc6 425. Nxe4 Bxe4 426. Qd2 Nc6 427. Rfd1 Nf6 428. Nc3 Nc6 429. Nxe4 Bxe4 430. Qd2 Nc6 431. Rfd1 Nf6 432. Nc3 Nc6 433. Nxe4 Bxe4 434. Qd2 Nc6 435. Rfd1 Nf6 436. Nc3 Nc6 437. Nxe4 Bxe4 438. Qd2 Nc6 439. Rfd1 Nf6 440. Nc3 Nc6 441. Nxe4 Bxe4 442. Qd2 Nc6 443. Rfd1 Nf6 444. Nc3 Nc6 445. Nxe4 Bxe4 446. Qd2 Nc6 447. Rfd1 Nf6 448. Nc3 Nc6 449. Nxe4 Bxe4 450. Qd2 Nc6 451. Rfd1 Nf6 452. Nc3 Nc6 453. Nxe4 Bxe4 454. Qd2 Nc6 455. Rfd1 Nf6 456. Nc3 Nc6 457. Nxe4 Bxe4 458. Qd2 Nc6 459. Rfd1 Nf6 460. Nc3 Nc6 461. Nxe4 Bxe4 462. Qd2 Nc6 463. Rfd1 Nf6 464. Nc3 Nc6 465. Nxe4 Bxe4 466. Qd2 Nc6 467. Rfd1 Nf6 468. Nc3 Nc6 469. Nxe4 Bxe4 470. Qd2 Nc6 471. Rfd1 Nf6 472. Nc3 Nc6 473. Nxe4 Bxe4 474. Qd2 Nc6 475. Rfd1 Nf6 476. Nc3 Nc6 477. Nxe4 Bxe4 478. Qd2 Nc6 479. Rfd1 Nf6 480. Nc3 Nc6 481. Nxe4 Bxe4 482. Qd2 Nc6 483. Rfd1 Nf6 484. Nc3 Nc6 485. Nxe4 Bxe4 486. Qd2 Nc6 487. Rfd1 Nf6 488. Nc3 Nc6 489. Nxe4 Bxe4 490. Qd2 Nc6 491. Rfd1 Nf6 492. Nc3 Nc6 493. Nxe4 Bxe4 494. Qd2 Nc6 495. Rfd1 Nf6 496. Nc3 Nc6 497. Nxe4 Bxe4 498. Qd2 Nc6 499. Rfd1 Nf6 500. Nc3 Nc6 501. Nxe4 Bxe4 502. Qd2 Nc6 503. Rfd1 Nf6 504. Nc3 Nc6 505. Nxe4 Bxe4 506. Qd2 Nc6 507. Rfd1 Nf6 508. Nc3 Nc6 509. Nxe4 Bxe4 510. Qd2 Nc6 511. Rfd1 Nf6 512. Nc3 Nc6 513. Nxe4 Bxe4 514. Qd2 Nc6 515. Rfd1 Nf6 516. Nc3 Nc6 517. Nxe4 Bxe4 518. Qd2 Nc6 519. Rfd1 Nf6 520. Nc3 Nc6 521. Nxe4 Bxe4 522. Qd2 Nc6 523. Rfd1 Nf6 524. Nc3 Nc6 525. Nxe4 Bxe4 526. Qd2 Nc6 527. Rfd1 Nf6 528. Nc3 Nc6 529. Nxe4 Bxe4 530. Qd2 Nc6 531. Rfd1 Nf6 532. Nc3 Nc6 533. Nxe4 Bxe4 534. Qd2 Nc6 535. Rfd1 Nf6 536. Nc3 Nc6 537. Nxe4 Bxe4 538. Qd2 Nc6 539. Rfd1 Nf6 540. Nc3 Nc6 541. Nxe4 Bxe4 542. Qd2 Nc6 543. Rfd1 Nf6 544. Nc3 Nc6 545. Nxe4 Bxe4 546. Qd2 Nc6 547. Rfd1 Nf6 548. Nc3 Nc6 549. Nxe4 Bxe4 550. Qd2 Nc6 551. Rfd1 Nf6 552. Nc3 Nc6 553. Nxe4 Bxe4 554. Qd2 Nc6 555. Rfd1 Nf6 556. Nc3 Nc6 557. Nxe4 Bxe4 558. Qd2 Nc6 559. Rfd1 Nf6 560. Nc3 Nc6 561. Nxe4 Bxe4 562. Qd2 Nc6 563. Rfd1 Nf6 564. Nc3 Nc6 565. Nxe4 Bxe4 566. Qd2 Nc6 567. Rfd1 Nf6 568. Nc3 Nc6 569. Nxe4 Bxe4 570. Qd2 Nc6 571. Rfd1 Nf6 572. Nc3 Nc6 573. Nxe4 Bxe4 574. Qd2 Nc6 575. Rfd1 Nf6 576. Nc3 Nc6 577. Nxe4 Bxe4 578. Qd2 Nc6 579. Rfd1 Nf6 580. Nc3 Nc6 581. Nxe4 Bxe4 582. Qd2 Nc6 583. Rfd1 Nf6 584. Nc3 Nc6 585. Nxe4 Bxe4 586. Qd2 Nc6 587. Rfd1 Nf6 588. Nc3 Nc6 589. Nxe4 Bxe4 590. Qd2 Nc6 591. Rfd1 Nf6 592. Nc3 Nc6 593. Nxe4 Bxe4 594. Qd2 Nc6 595. Rfd1 Nf6 596. Nc3 Nc6 597. Nxe4 Bxe4 598. Qd2 Nc6 599. Rfd1 Nf6 600. Nc3 Nc6 601. Nxe4 Bxe4 602. Qd2 Nc6 603. Rfd1 Nf6 604. Nc3 Nc6 605. Nxe4 Bxe4 606. Qd2 Nc6 607. Rfd1 Nf6 608. Nc3 Nc6 609. Nxe4 Bxe4 610. Qd2 Nc6 611. Rfd1 Nf6 612. Nc3 Nc6 613. Nxe4 Bxe4 614. Qd2 Nc6 615. Rfd1 Nf6 616. Nc3 Nc6 617. Nxe4 Bxe4 618. Qd2 Nc6 619. Rfd1 Nf6 620. Nc3 Nc6 621. Nxe4 Bxe4 622. Qd2 Nc6 623. Rfd1 Nf6 624. Nc3 Nc6 625. Nxe4 Bxe4 626. Qd2 Nc6 627. Rfd1 Nf6 628. Nc3 Nc6 629. Nxe4 Bxe4 630. Qd2 Nc6 631. Rfd1 Nf6 632. Nc3 Nc6 633. Nxe4 Bxe4 634. Qd2 Nc6 635. Rfd1 Nf6 636. Nc3 Nc6 637. Nxe4 Bxe4 638. Qd2 Nc6 639. Rfd1 Nf6 640. Nc3 Nc6 641. Nxe4 Bxe4 642. Qd2 Nc6 643. Rfd1 Nf6 644. Nc3 Nc6 645. Nxe4 Bxe4 646. Qd2 Nc6 647. Rfd1 Nf6 648. Nc3 Nc6 649. Nxe4 Bxe4 650. Qd2 Nc6 651. Rfd1 Nf6 652. Nc3 Nc6 653. Nxe4 Bxe4 654. Qd2 Nc6 655. Rfd1 Nf6 656. Nc3 Nc6 657. Nxe4 Bxe4 658. Qd2 Nc6 659. Rfd1 Nf6 660. Nc3 Nc6 661. Nxe4 Bxe4 662. Qd2 Nc6 663. Rfd1 Nf6 664. Nc3 Nc6 665. Nxe4 Bxe4 666. Qd2 Nc6 667. Rfd1 Nf6 668. Nc3 Nc6 669. Nxe4 Bxe4 670. Qd2 Nc6 671. Rfd1 Nf6 672. Nc3 Nc6 673. Nxe4 Bxe4 674. Qd2 Nc6 675. Rfd1 Nf6 676. Nc3 Nc6 677. Nxe4 Bxe4 678. Qd2 Nc6 679. Rfd1 Nf6 680. Nc3 Nc6 681. Nxe4 Bxe4 682. Qd2 Nc6 683. Rfd1 Nf6 684. Nc3 Nc6 685. Nxe4 Bxe4 686. Qd2 Nc6 687. Rfd1 Nf6 688. Nc3 Nc6 689. Nxe4 Bxe4 690. Qd2 Nc6 691. Rfd1 Nf6 692. Nc3 Nc6 693. Nxe4 Bxe4 694. Qd2 Nc6 695. Rfd1 Nf6 696. Nc3 Nc6 697. Nxe4 Bxe4 698. Qd2 Nc6 699. Rfd1 Nf6 700. Nc3 Nc6 701. Nxe4 Bxe4 702. Qd2 Nc6 703. Rfd1 Nf6 704. Nc3 Nc6 705. Nxe4 Bxe4 706. Qd2 Nc6 707. Rfd1 Nf6 708. Nc3 Nc6 709. Nxe4 Bxe4 710. Qd2 Nc6 711. Rfd1 Nf6 712. Nc3 Nc6 713. Nxe4 Bxe4 714. Qd2 Nc6 715. Rfd1 Nf6 716. Nc3 Nc6 717. Nxe4 Bxe4 718. Qd2 Nc6 719. Rfd1 Nf6 720. Nc3 Nc6 721. Nxe4 Bxe4 722. Qd2 Nc6 723. Rfd1 Nf6 724. Nc3 Nc6 725. Nxe4 Bxe4 726. Qd2 Nc6 727. Rfd1 Nf6 728. Nc3 Nc6 729. Nxe4 Bxe4 730. Qd2 Nc6 731. Rfd1 Nf6 732. Nc3 Nc6 733. Nxe4 Bxe4 734. Qd2 Nc6 735. Rfd1 Nf6 736. Nc3 Nc6 737. Nxe4 Bxe4 738. Qd2 Nc6 739. Rfd1 Nf6 740. Nc3 Nc6 741. Nxe4 Bxe4 742. Qd2 Nc6 743. Rfd1 Nf6 744. Nc3 Nc6 745. Nxe4 Bxe4 746. Qd2 Nc6 747. Rfd1 Nf6 748. Nc3 Nc6 749. Nxe4 Bxe4 750. Qd2 Nc6 751. Rfd1 Nf6 752. Nc3 Nc6 753. Nxe4 Bxe4 754. Qd2 Nc6 755. Rfd1 Nf6 756. Nc3 Nc6 757. Nxe4 Bxe4 758. Qd2 Nc6 759. Rfd1 Nf6 760. Nc3 Nc6 761. Nxe4 Bxe4 762. Qd2 Nc6 763. Rfd1 Nf6 764. Nc3 Nc6 765. Nxe4 Bxe4 766. Qd2 Nc6 767. Rfd1 Nf6 768. Nc3 Nc6 769. Nxe4 Bxe4 770. Qd2 Nc6 771. Rfd1 Nf6 772. Nc3 Nc6 773. Nxe4 Bxe4 774. Qd2 Nc6 775. Rfd1 Nf6 776. Nc3 Nc6 777. Nxe4 Bxe4 778. Qd2 Nc6 779. Rfd1 Nf6 780. Nc3 Nc6 781. Nxe4 Bxe4 782. Qd2 Nc6 783. Rfd1 Nf6 784. Nc3 Nc6 785. Nxe4 Bxe4 786. Qd2 Nc6 787. Rfd1 Nf6 788. Nc3 Nc6 789. Nxe4 Bxe4 790. Qd2 Nc6 791. Rfd1 Nf6 792. Nc3 Nc6 793. Nxe4 Bxe4 794. Qd2 Nc6 795. Rfd1 Nf6 796. Nc3 Nc6 797. Nxe4 Bxe4 798. Qd2 Nc6 799. Rfd1 Nf6 800. Nc3 Nc6 801. Nxe4 Bxe4 802. Qd2 Nc6 803. Rfd1 Nf6 804. Nc3 Nc6 805. Nxe4 Bxe4 806. Qd2 Nc6 807. Rfd1 Nf6 808. Nc3 Nc6 809. Nxe4 Bxe4 810. Qd2 Nc6 811. Rfd1 Nf6 812. Nc3 Nc6 813. Nxe4 Bxe4 814. Qd2 Nc6 815. Rfd1 Nf6 816. Nc3 Nc6 817. Nxe4 Bxe4 818. Qd2 Nc6 819. Rfd1 Nf6 820. Nc3 Nc6 821. Nxe4 Bxe4 822. Qd2 Nc6 823. Rfd1 Nf6 824. Nc3 Nc6 825. Nxe4 Bxe4 826. Qd2 Nc6 827. Rfd1 Nf6 828. Nc3 Nc6 829. Nxe4 Bxe4 830. Qd2 Nc6 831. Rfd1 Nf6 832. Nc3 Nc6 833. Nxe4 Bxe4 834. Qd2 Nc6 835. Rfd1 Nf6 836. Nc3 Nc6 837. Nxe4 Bxe4 838. Qd2 Nc6 839. Rfd1 Nf6 840. Nc3 Nc6 841. Nxe4 Bxe4 842. Qd2 Nc6 843. Rfd1 Nf6 844. Nc3 Nc6 845. Nxe4 Bxe4 846. Qd2 Nc6 847. Rfd1 Nf6 848. Nc3 Nc6 849. Nxe4 Bxe4 850. Qd2 Nc6 851. Rfd1 Nf6 852. Nc3 Nc6 853. Nxe4 Bxe4 854. Qd2 Nc6 855. Rfd1 Nf6 856. Nc3 Nc6 857. Nxe4 Bxe4 858. Qd2 Nc6 859. Rfd1 Nf6 860. Nc3 Nc6 861. Nxe4 Bxe4 862. Qd2 Nc6 863. Rfd1 Nf6 864. Nc3 Nc6 865. Nxe4 Bxe4 866. Qd2 Nc6 867. Rfd1 Nf6 868. Nc3 Nc6 869. Nxe4 Bxe4 870. Qd2 Nc6 871. Rfd1 Nf6 872. Nc3 Nc6 873. Nxe4 Bxe4 874. Qd2 Nc6 875. Rfd1 Nf6 876. Nc3 Nc6 877. Nxe4 Bxe4 878. Qd2 Nc6 879. Rfd1 Nf6 880. Nc3 Nc6 881. Nxe4 Bxe4 882. Qd2 Nc6 883. Rfd1 Nf6 884. Nc3 Nc6 885. Nxe4 Bxe4 886. Qd2 Nc6 887. Rfd1 Nf6 888. Nc3 Nc6 889. Nxe4 Bxe4 890. Qd2 Nc6 891. Rfd1 Nf6 892. Nc3 Nc6 893. Nxe4 Bxe4 894. Qd2 Nc6 895. Rfd1 Nf6 896. Nc3 Nc6 897. Nxe4 Bxe4 898. Qd2 Nc6 899. Rfd1 Nf6 900. Nc3 Nc6 901. Nxe4 Bxe4 902. Qd2 Nc6 903. Rfd1 Nf6 904. Nc3 Nc6 905. Nxe4 Bxe4 906. Qd2 Nc6 907. Rfd1 Nf6 908. Nc3 Nc6 909. Nxe4 Bxe4 910. Qd2 Nc6 911. Rfd1 Nf6 912. Nc3 Nc6 913. Nxe4 Bxe4 914. Qd2 Nc6 915. Rfd1 Nf6 916. Nc3 Nc6 917. Nxe4 Bxe4 918. Qd2 Nc6 919. Rfd1 Nf6 920. Nc3 Nc6 921. Nxe4 Bxe4 922. Qd2 Nc6 923. Rfd1 Nf6 924. Nc3 Nc6 925. Nxe4 Bxe4 926. Qd2 Nc6 927. Rfd1 Nf6 928. Nc

38)

**TEST 38 (Passed) :**

When then question form is opened, the player's timer label STOPS counting down and pauses correctly. In this case, the question was answered wrong so time was deducted and then player 2's timer starts counting down.

39/40)

**TEST 39 /40(Passed) :**

When the correct username and password is entered, the admin page form loads up and is given focus (shown as it is at the front, ontop of the login screen). Also as seen from the screenshot, when the admin user types the password, it is concealed by symbol characters but when processing, the data input is still intact. This test shows the login process working correctly and the password security feature also working as it should.

41)

**TEST 41 (Passed) :**

When an incorrect combination was entered, a message box appears warning that the data input is incorrect. Once the OK button is clicked, the message box closes and the form remains open so the password or username can be changed and the attempt to login can happen again. Access to the admin page is not given – which is how the system should work.

42)

**TEST 42 (Passed) :**

Like some others, this test is hard to show by means of screenshots but as shown, the newly opened forms are at the front are in focus for the user. When the user clicks off of the form in an area around the form – usually the window would be sent to the back and focus would be lost. But when this does happen, the form simply flashes and the windows beep sound plays and the form stays in focus at the front, indicating to the user that this form has to be dealt with or closed before they can perform other tasks.

43)

```

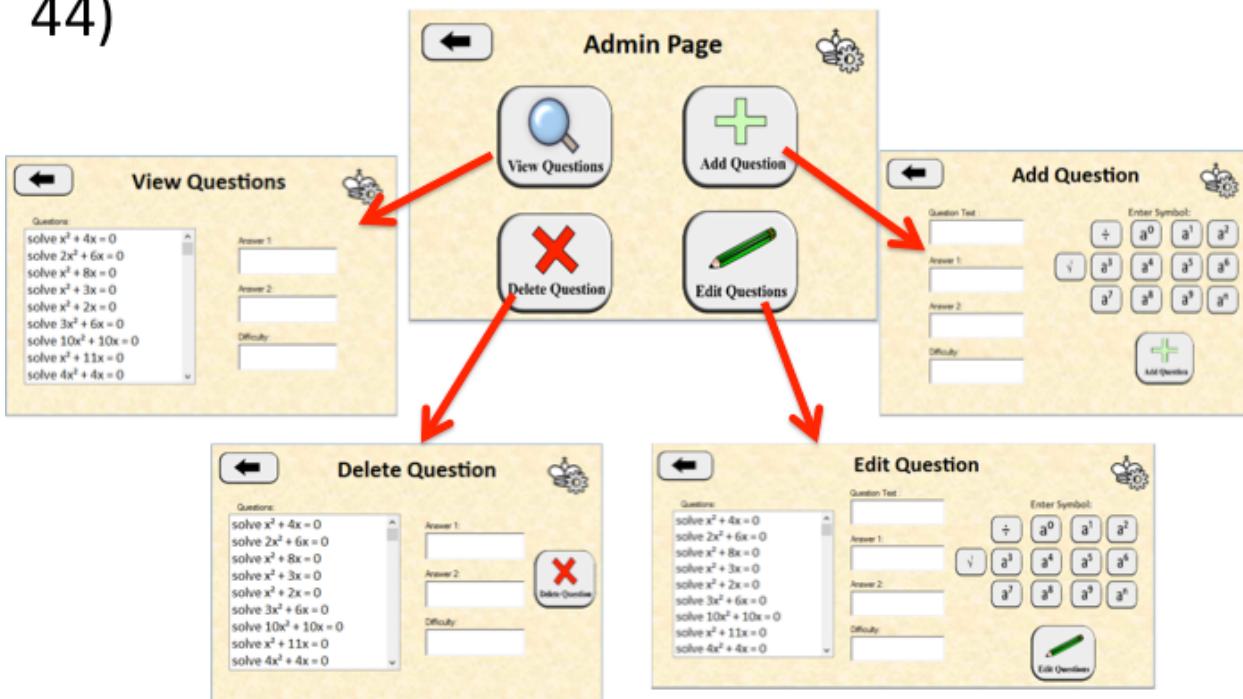
BoardAdmin.questions = new List<QuestionBank>();
while (aSM.EndOfS
{
    string[] a =
        // adds a que
    BoardAdmin.qu
}
aSM.Close();
MessageBox.Show("as popu
ate void AdminBut
// creates the ad

```

**TEST 43 (Passed) :**

For this test, a code break was used in the form of a message box, to allow the debugging to be paused and the current state of the system to be analysed. This message box was added after the content of the file was retrieved and the data was imported into the question object list in the **gatherQuestionsFromFile** routine. Firstly, the number of question objects in the list was equal to the number of questions and hence lines in the text file. Then as shown, each question object correctly had properties for each part of the question. Having the question objects created at the same time as their properties are assigned proved to be very efficient here.

44)

**TEST 44 (Passed) :**

All navigation works fine and opens the forms that are meant to open. The back button also works correctly by closing the form that is currently open to reveal the main admin page again. All buttons work and append the symbol to the current textbox that has the cursor in.

45)

**View Questions**

Questions:

- solve  $x^2 + 4x = 0$
- solve  $2x^2 + 6x = 0$
- solve  $x^2 + 8x = 0$
- solve  $x^2 + 3x = 0$
- solve  $x^2 + 2x = 0$
- solve  $3x^2 + 6x = 0$
- solve  $10x^2 + 10x = 0$
- solve  $x^2 + 11x = 0$
- solve  $4x^2 + 4x = 0$

Answer 1:

Answer 2:

Difficulty:

**View Questions**

Questions:

- solve  $x^2 + 4x = 0$
- solve  $2x^2 + 6x = 0$
- solve  $x^2 + 8x = 0$**
- solve  $x^2 + 3x = 0$
- solve  $x^2 + 2x = 0$
- solve  $3x^2 + 6x = 0$
- solve  $10x^2 + 10x = 0$
- solve  $x^2 + 11x = 0$
- solve  $4x^2 + 4x = 0$

Answer 1:  0

Answer 2:  -8

Difficulty:  1

**View Questions**

Questions:

- simplify  $3\sqrt{18} + 5\sqrt{18}$
- Find  $n: \sqrt{a^2 \div a^{10}} = a^n$
- Find  $n: \sqrt{a^6 \div a^{11}} = a^n$
- Find  $n: \sqrt{a^{64} \div a^{22}} = a^n$
- Find  $n: \sqrt{a^{25} \div a^8} = a^n$
- Find  $n: \sqrt{a^{10} \div a^{10}} = a^n$
- Find  $n: \sqrt{a^3 \times a^{10}} = a^n$**
- Find  $n: \sqrt{a^3 \times a^{10}} = a^n$
- Find  $n: \sqrt{a^{64} \times a^{22}} = a^n$

Answer 1:  13/2

Answer 2:

Difficulty:  3

**TEST 45 (Passed) :**

The question list box was populated correctly when the view questions form loads. The question text property of each question object was correctly viewable in the list. When a question is selected, its corresponding properties show in the adjacent text boxes and the information displayed is correct and of the same question object, thus working as it should do. Questions with only one answer have the second answer box greyed out as shown.

46)

Question Text:  8 ÷  

Enter Symbol: ÷  $a^0$   $a^1$   $a^2$

Answer 1:

Answer 2:

Difficulty:

**Add Question**

Question Text:  8 ÷  

Enter Symbol: ÷  $a^0$   $a^1$   $a^2$

Answer 1:  3<sup>4</sup>  

Answer 2:

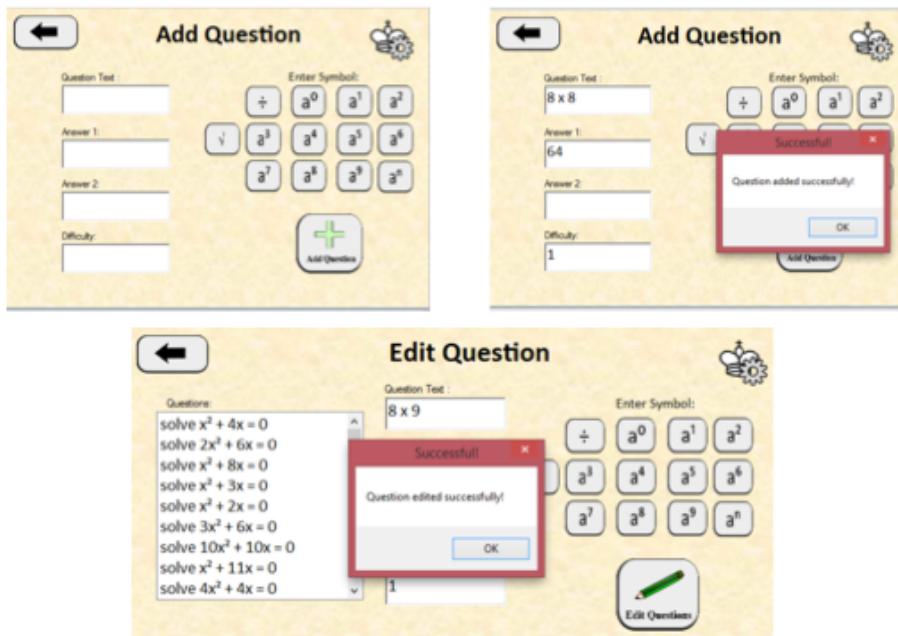
Difficulty:

**Add Question**

**TEST 46 (Passed) :**

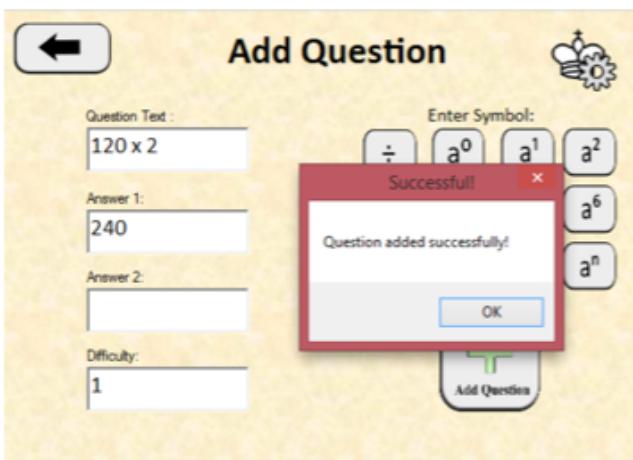
All buttons append text correctly. Whatever button is clicked, its corresponding symbol is appended to the text box that has the current focus, the one with which the cursor is in currently.

47)

**TEST 47 (Passed) :**

When there is no input or missing field and the edit/add button is clicked, nothing happens. The list box of questions is not updated and no changes are made. But if the correct fields have input and the buttons are clicked then a confirmation appears letting the user know that the changes have been made.

48)

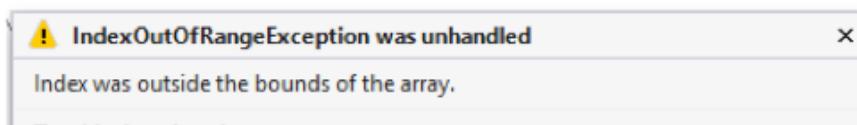


Find n:  $\sqrt{(a^{n_0} \times a^{n_1})} = a^{n_2}$ , 45,,3  
 $solve 10x^2 + 80x + 120 = 0, -6, -2, 3$   
 $solve 20x^2 + 60x + 40 = 0, -2, -1, 3$   
 $solve 25x^2 + 100x + 100 = 0, -2, , 3$   
 $solve 2x^2 - 16x + 24 = 0, 6, 2, 3$   
 $solve 10x^2 + 30x - 400 = 0, -8, 5, 3$   
 $solve 100x^2 + 500x + 600 = 0, -3, -2, 3$   
 $120 \times 2, 240, , 1$

**TEST 48 (Passed) :**

When the question is added, the question object and its properties are appended to the text file and should be in the correct format to allow reading again in another process. As shown, adding the question works correctly and in the text file, the line is correctly separated and the question property are ready to be read from.

49)



```

Find n: √(a25 × a8) = an,33/2,,3
Find n: √(a80 × a10) = an,45,,3
solve 10x2 + 80x + 120 = 0,-6,-2,3
solve 20x2 + 60x + 40 = 0,-2,-1,3
solve 25x2 + 100x + 100 = 0,-2,,3
solve 2x2 - 16x + 24 = 0,6,2,3
solve 10x2 + 30x - 400 = 0,-8,5,3
solve 100x2 + 500x + 600 = 0,-3,-2,3
solve 6x2 + 6x = 0,0,-1,1
9 x 9,72,,1
9 x 9,81,,1

```

**TEST 49 (Failed) :**

Editing a question works correctly – a question that is edited is removed from the list of questions and the updated question is simply appended to the file in a similar manor to adding a question. This process works correctly – however a problem occurred when a question was ADDED after a question being edited – a blank line was present in the file and this caused an error when reading in the contents of the file in a new run of the program (as the blank line is not in the correct format to be read – comma separated).

SO although the editing process works, it causes problems for subsequent manipulation of the file.

```

StreamWriter aSW = new StreamWriter(File.Open("QuestionBank 2.csv", FileMode.Open));
foreach (QuestionBank aQuestion in BoardAdmin.questions)
{
    aSW.WriteLine(aQuestion.questiontext + "," + aQuestion.answer + "," + aQuestion.answer2 + "," + aQuestion.difficulty);
}
aSW.Close();

```

**TEST 49 (Solution) :**

The problem lies with the code that writes the new data to the file under the edit question module. In the module, the list of question objects is simply looped through and the properties of each questions are put into a correct string line, separated by commas and this line is written to the file. The problem with this however, is this function adds a line terminator to the end of the line so that each question is on a separate line – but this means a blank line will be added for the last question line in the loop. So a new method of writing to the file is needed.

To solve this problem, the blank line at the end of the stream needs to be removed and to do this I can read in the whole file as one big text variable, trim that string and then write the string back to the file ( which overwrites the current file). This fixes the problem.

```

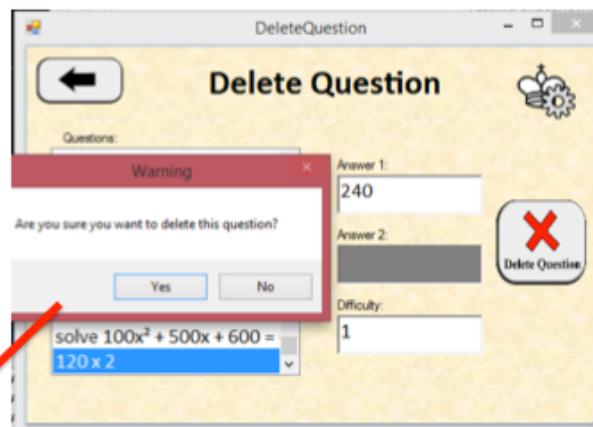
// the following takes the text from the file and removes all blank lines after the content to ensure no errors occur
// when reading in from the file on another process
string temptext = File.ReadAllText("QuestionBank 2.csv");
temptext = temptext.Trim();
File.WriteAllText("QuestionBank 2.csv", temptext); // overwrites

```

50)

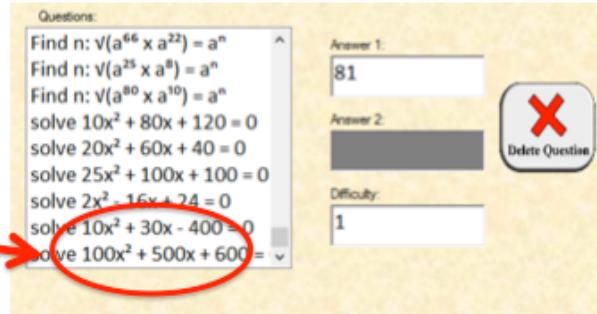
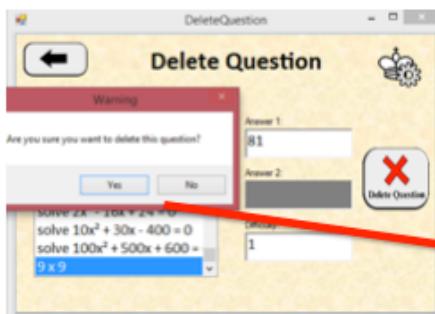
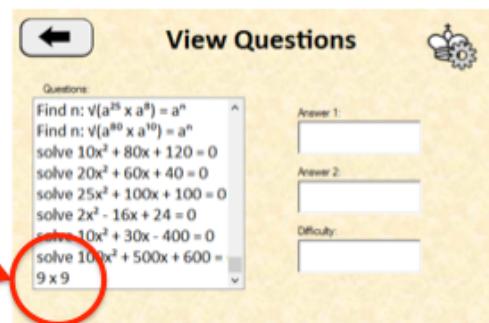
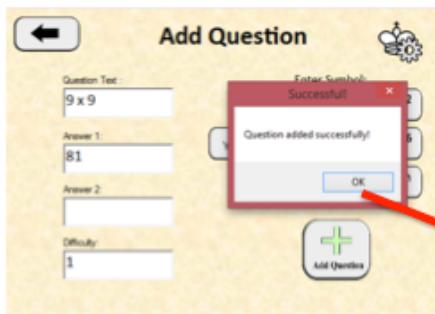
```
solve 20x2 + 60x + 40 = 0,-2,-1,3
solve 25x2 + 100x + 100 = 0,-2,,3
solve 2x2 - 16x + 24 = 0,6,2,3
solve 10x2 + 30x - 400 = 0,-8,5,3
solve 100x2 + 500x + 600 = 0,-3,-2,3
120 x 2,240,,1
```

```
solve 10x2 + 80x + 120 = 0,-6,-2,3
solve 20x2 + 60x + 40 = 0,-2,-1,3
solve 25x2 + 100x + 100 = 0,-2,,3
solve 2x2 - 16x + 24 = 0,6,2,3
solve 10x2 + 30x - 400 = 0,-8,5,3
solve 100x2 + 500x + 600 = 0,-3,-2,3
```

**TEST 50 (Passed) :**

Showing the question at the bottom of the text file, this is the question to be deleted. In the program, the question is selected from the question list box and the delete button is clicked. A warning is given about deleting but when the yes button is clicked, the list box is updated and the file has also been updated as shown.

51)

**TEST 51 (Passed) :**

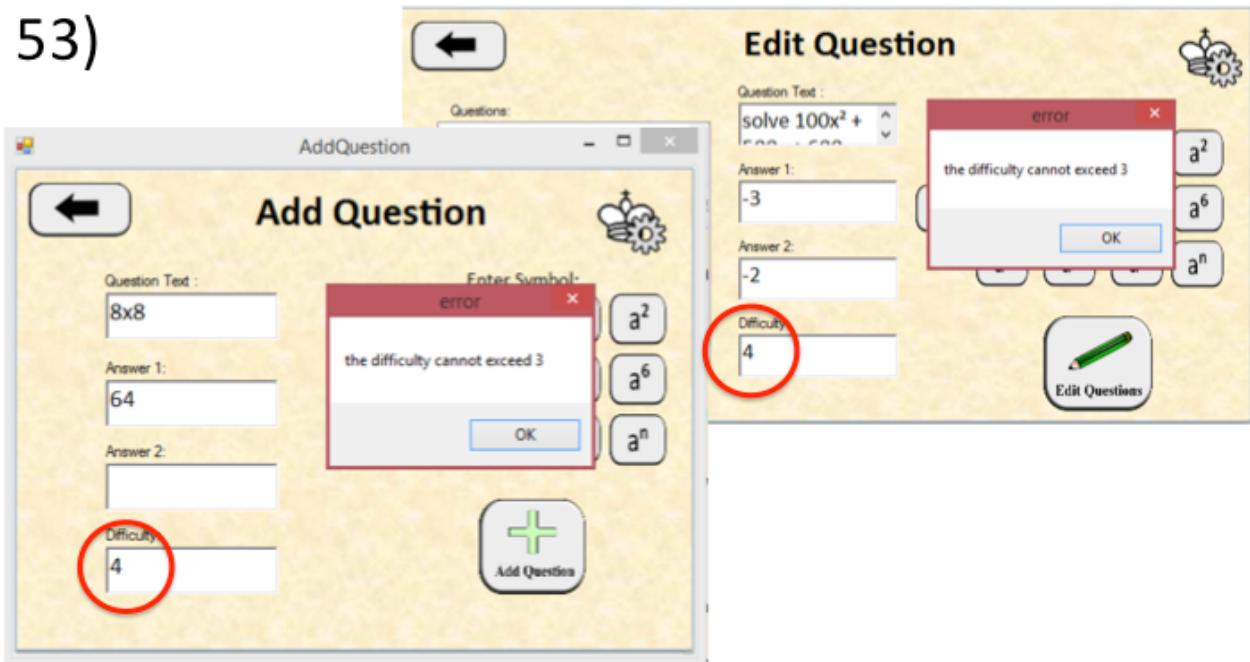
Updates made to the file are made immediately both internally and externally. The file itself is changed as shown in previous test but the contents of the list box on the form itself is also updated immediately so the user can see the changes – as the user never sees the text file. This list box is updated wherever it is present using the **populateListBox** procedure.

52)

**TEST 52 (Passed) :**

When the admin page is exited, the forms close and the main menu is back in focus. Then to get back to the admin section, the admin button on the main menu must be clicked and this then generates the login process again – for security. This is working correctly.

53)

**TEST 53 (Passed) :**

When the difficulty is entered with a number outside the range, an error message occurs and the processing does not take place – as expected. Other validation here includes erroneous data types (such as entering strings) all have their own error message.

# System Maintenance

**System Overview**

**Form sizes and purposes:**

- AddQuestion.cs - 5KB  
The add question form is part of the admin section of the project which allows an admin user to add a question to the question bank file, which is stored on a central server. The form includes input mediums (e.g text boxes) for each part of a question: text, answer1, answer2 and difficulty. There are various buttons with symbols that allow the user to add unicode symbols that are not available on the keyboard. The data is processed under the addQuestion() method.
- AddQuestion.cs - 5KB  
The design of this form is to allow the admin user to add a question to the question bank file. It contains fields for question text, two answers, and difficulty level, along with a button to save the new question.
- AdminLogin.cs - 2KB  
The admin login form is used by administrators to log in to the system. It has fields for username and password, and a login button.
- GameBoard.cs - 10KB  
The game board form displays the current state of the board and allows players to make moves. It includes a grid of squares and buttons for selecting pieces and performing actions like promotion.

```

graph TD
    MM[Main Menu] --> AS[Admin Section]
    MM --> GB[GameBoard]
    AS --> AL[Admin Login]
    GB --> AQ[Ask Question]
    GB --> PP[Promote Pawn]
  
```

**GameBoard**

**GameBoard (BoardUI)**

Procedure/Function Name	Parameters	Purpose
<code>public BoardUI()</code>	N/A	<p>This procedure is run when the form is loaded. The first thing this procedure does is create the game using the appropriate methods in the game class, which will override depending on what game mode has been selected.</p> <p>The next thing it does, it manipulates controls on the form, corresponding to the game mode (for example, the procedure makes timer labels visible if the game mode is a timed game). This procedure also enables the <code>updateViewTimer</code> which is a timer used to time when the UI is updated. The procedure <code>assignBoardSquares</code> is also invoked here.</p>

## ● System Overview

### **Form sizes and purposes:**

- AddQuestion.cs - 5KB

The add question form is part of the admin section of the project which allows an admin user to add a question to the question bank file, which is stored on a central server.

The form includes input mediums (e.g text boxes) for each part of a question: text, answer1, answer 2 and difficulty. There are various buttons with symbols that allow the user to add unicode symbols that are not available on the keyboard. The data is processed under the action of an “ADD” button.

- AddQuestion.Designer.cs - 27KB

The designer file associated with the form.

- AddQuestion.resx - 6KB

The resource file associated with the form.

- AdminLogin.cs - 2KB

The admin login form is the interface that the admin user is presented with when trying to access the admin section of the system. The form includes a username and password input means and a button to submit the data. The username and password combination entered determines whether the admin page form is opened for the user.

- AdminLogin.Designer.cs - 9KB

The designer file associated with the form.

- AdminLogin.resx - 6KB

The resource file associated with the form.

- AdminPage.cs - 3KB

The admin page form is the interface that the admin user is presented with after a successful login in the login page. This form consists of four main buttons which correspond to the different areas in the admin section: adding a question, deleting a question, editing a question and viewing the file of questions.

- AdminPage.Designer.cs - 11KB

The designer file associated with the form.

- AdminPage.resx - 6KB

The resource file associated with the form.

- AskQuestion.cs - 12KB

The ask question form is the interface that is presented to the user in gameplay, during a maths mode game. With each move made in the chess game, this form pops up with a new randomly generated question from the question bank file which corresponds to the associated difficulty level the current player chose. The form has means of input to answer the question, the behaviour of which depends on how many answers to the question there are. There are buttons to input symbols and to submit the user answers.

- AskQuestion.Designer.cs - 12KB

The designer file associated with the form.

- AskQuestion.resx - 6KB

The resource file associated with the form.

- Bishop.cs - 5KB

The bishop class is a class that inherits the class piece. Bishop is a type of piece on the chess board and this class handles all the specific rules associated with the bishop piece. This is within overridden methods from the piece class.

- Board.cs - 4KB

The board class is a class that is used to create an object that represents the game board. The board class therefore handles the sets of pieces on the board and also the squares on the board. The board object created is a property of the game class.

- BoardAdmin.cs - 1KB

The boardAdmin class is a static class which stores statics variables that are used frequently over the program. Instead of passing variables all the time, a tedious process, the most important variables are stored here for easy access over the system. This includes the game object - which contains all the relevant properties for the game and variables such as **isTempMoving** to quickly tell whether the move being made is temporary or not.

- BoardUI.cs - 48KB

The boardUI form is the main form in use in the system. This is the interface the user interacts with when playing the actual game of chess. This form is loaded after the

MainMenu form is dealt with and closed. This form contains the board itself with all the pieces that can be moved under a strict combination of click events. It also contains other properties of the game such as player timers, maths scores, move history etc.

- BoardUI.Designer.cs - 94KB

The designer file associated with the form.

- BoardUI.resx - 4.5KB

The resource file associated with the form.

- Comp4\_Chess.csproj - 16KB

The project file associated with the program.

- Comp4\_Chess.csproj.user - 0.14KB

The project file associated with the program.

- DeleteQuestion.cs - 5KB

The delete question form is a form that branches from the admin page form and is within the admin section of the project. This form allows the user to view the questions stored in the question bank and select one to delete. Clicking a delete button processed the relevant information and makes changes to the text file on the server.

- DeleteQuestion.Designer.cs - 12KB

The designer file associated with the form.

- DeleteQuestion.resx - 6KB

The resource file associated with the form.

- EditQuestion.cs - 6KB

The edit question form is a form that branches from the admin page form and is within the admin section of the project. This form allows the user to view questions from the file and edit the properties of the question which in turn updates the file itself.

- EditQuestion.Designer.cs - 28KB

The designer file associated with the form.

- EditQuestion.resx - 6KB

The resource file associated with the form.

- Game.cs - 9KB

The game class is a class that contains all the relevant information for a game of chess. The main properties involved with a game of chess are the players, who's turn is it? , the history of moves made, the state of the game, the game board and the game result. All these things are dealt with in the game class. The game class however, is abstract, as a certain type of game will be played - timed, maths etc.

- King.cs - 8KB

The king class is a class that inherits the piece class and contains all the relevant code and rules specific to a piece that is a king. This applies to the movement of the king, whether the king can castle and whether the king is in check or not. The king contains overridden methods from the piece class with adaptions that are specific.

- Knight.cs - 2KB

The knight class is a class that inherits the piece class and contains all the relevant code and rules specific to a piece that is a knight. This class overrides methods from the piece class and applies rules that correspond to the knight piece.

- MainMenu.cs - 13KB

The Main menu form is the first interface the user is presented with upon running the program. This is where all important selections are made for the game ( piece colours, game board style, timer values etc) and from here the user can play the game they have selected from the options available. Also from the main menu form, an admin user can access an admin section of the program.

- MainMenu.Designer.cs - 29KB

The designer file associated with the form.

- MainMenu.resx - 6KB

The resource file associated with the form.

- MathsGame.cs - 1KB

The maths game class inherits the game class and contains properties and rules specific to a game that is a maths type. A maths game specifically deals with current scoring of maths questions and also the process of presenting the question to the user.

- Move.cs - 1KB

The move class allows for the creation of move objects which are stored in the game's history as a list of moves. Each move object contains properties that detail what happened within each move of the game (i.e which piece moves to where, any captured pieces and whether castling took place).

- **Pawn.cs - 12KB**

The pawn class is a class that inherits the piece class and contains all the relevant code and rules specific to a piece that is a pawn. This class overrides methods from the piece class and applies rules that correspond to the pawn piece. Also in this class is the process of promoting a pawn piece when it reaches the other side of the game board. This is linked with the promote pawn form, which is created and opened in this class.

- **Piece.cs - 7KB**

The piece class is an abstract class that contains all information about a piece that is generic (i.e type, colour, position on board etc) and also contains methods to do with moving of a piece, capturing and other move related processes such as getting the possible moves for a particular piece.

- **PieceSet.cs - 9KB**

The piece set class is a class that stores the pieces in lists determined by their colour. This is a logical grouping of piece objects and this class can then be stored as a single property.

- **Player.cs - 9KB**

The player class is a class that represents a player in the game. This class deals with making the moves of the game (as a player makes a move) This could be a real move or a temporary move depending on what is currently being processed. In the program, each possible move is made temporarily to evaluate whether a player is in check etc. The player class has a colour and a difficulty or timer (for maths and timed games).

- **Program.cs - 0.52KB**

The class that deals with the entry point and execution of the program system.

- **PromotePawn.cs - 3KB**

The promote pawn form is the interface that is presented to the user when a pawn piece reaches the other side of the game board. The player may promote a pawn piece to a piece of a better rank and this form gives the user that option.

- **PromotePawn.Designer.cs - 8KB**

The designer file associated with the form.

- **PromotePawn.resx - 6KB**

The resource file associated with the form.

- **Queen.cs - 11KB**

The queen class is a class that inherits the piece class and contains all overridden code relevant for the queen type piece.

- QuestionBank.cs - 1KB

The question bank class represents A question in the program/ question file. When the program loads a list of these question bank objects are populated with information from the text file stored on the server. These objects can then be easily referenced in the system when asking questions in the maths mode. A question bank object has properties which correspond to that of a question - the question text, the answers and the difficulty.

- Rook.cs - 5KB

The rook class is a class that inherits the piece class and contains all overridden code relevant for the rook type piece.

- Square.cs - 0.71KB

The square class is a class that represents a square on the game board. Therefore this class is used to store the x and y coordinates of the position on the board as well as whether the square is occupied or not (useful for reference in code calculations). The square objects are stored in a two dimensional array which represents the board and this is a property of the board class and hence game class.

- StandardGame.cs - 0.65KB

The standard game class inherits the game class and allows the user to play a normal game of chess without any maths or timers.

- TimedGame.cs - 1KB

The timed game class is a class that inherits the game class and allows the user to play a timed game of chess.

- ViewQuestions.cs - 2KB

The view questions form is a form that is available in the admin section of the system. It allows the admin user to view all questions in the file stored on the sever in a user-friendly format. The properties of each question (answers etc) are also viewable in correspondence to question viewing.

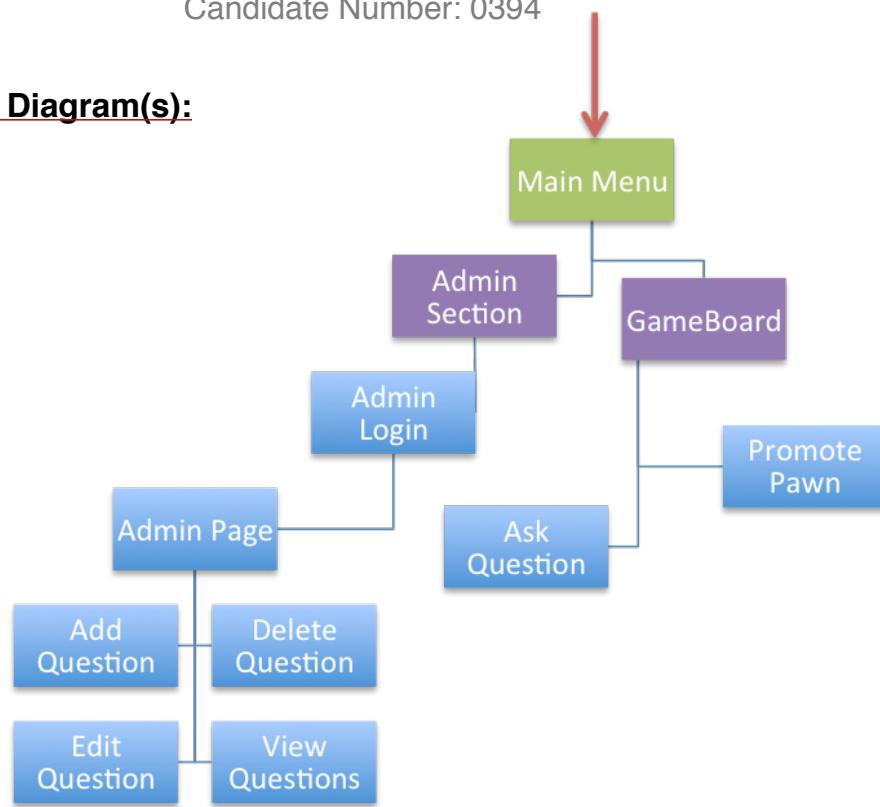
- ViewQuestions.Designer.cs - 10KB

The designer file associated with the form.

- ViewQuestions.resx - 6KB

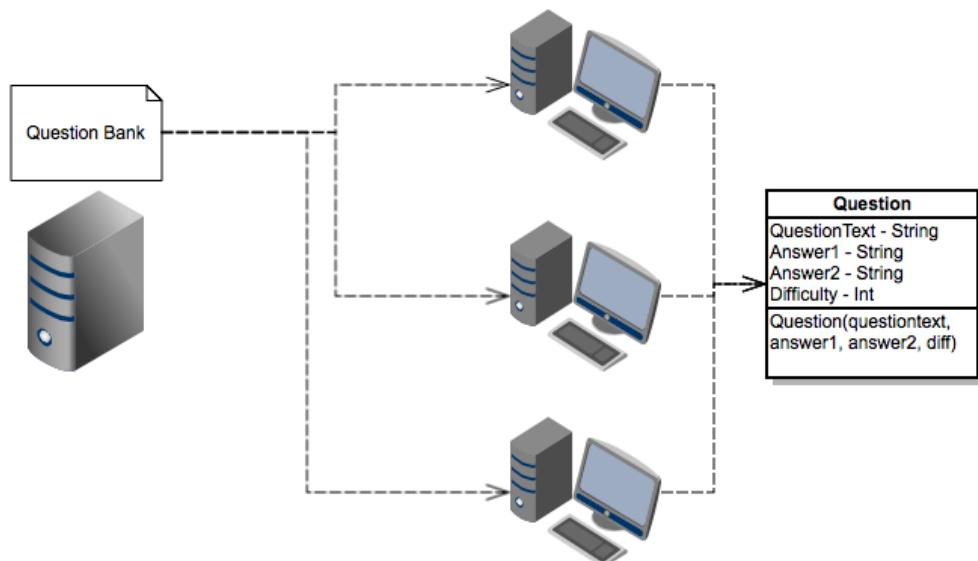
The resource file associated with the form.

**TOTAL SIZE = 492.52KB**

**System Overview Diagram(s):**

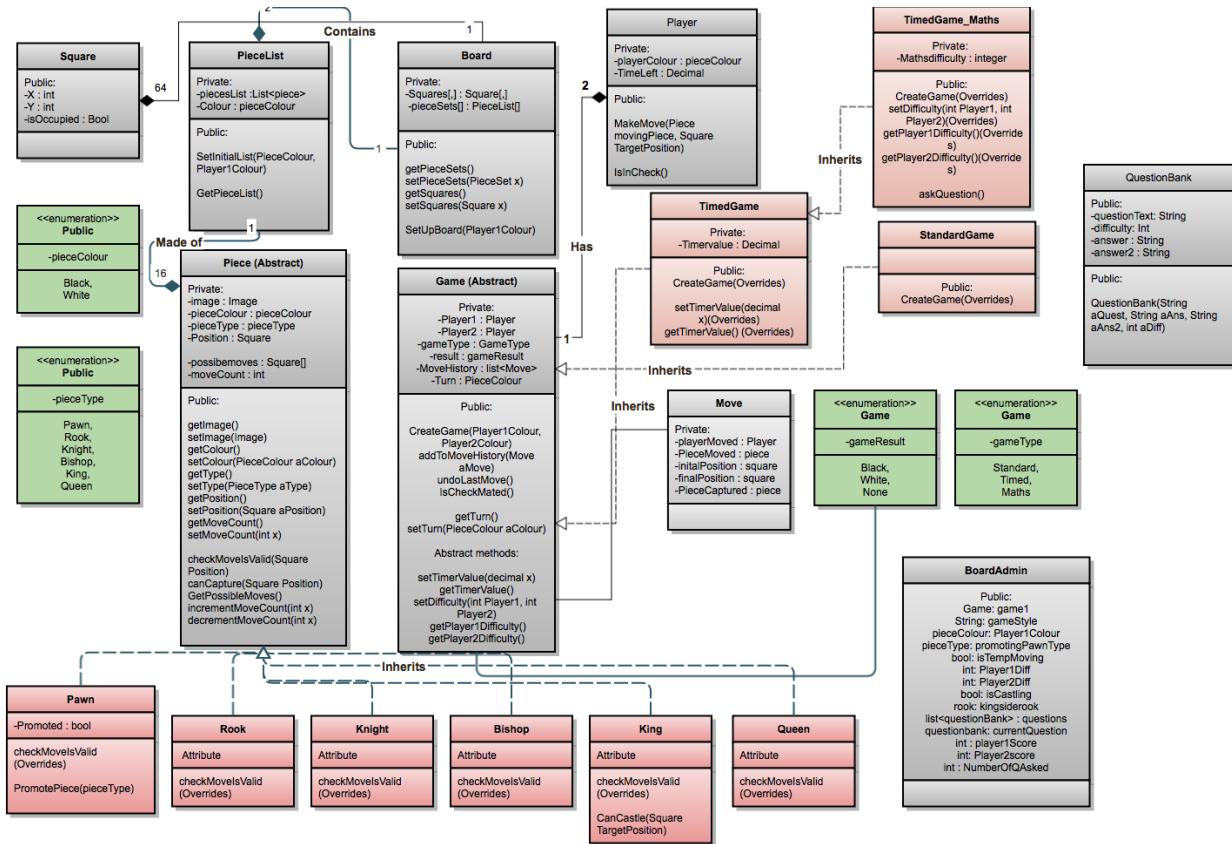
The diagram shows an overview of the system in modular form and represents the association of component within the system. For details of the specific modules, refer to the design section of the project.

Here is the diagram representing the interaction between the server and the text file containing the questions. Again, for more detail, see the design section of the project, under file organisation and data processing.



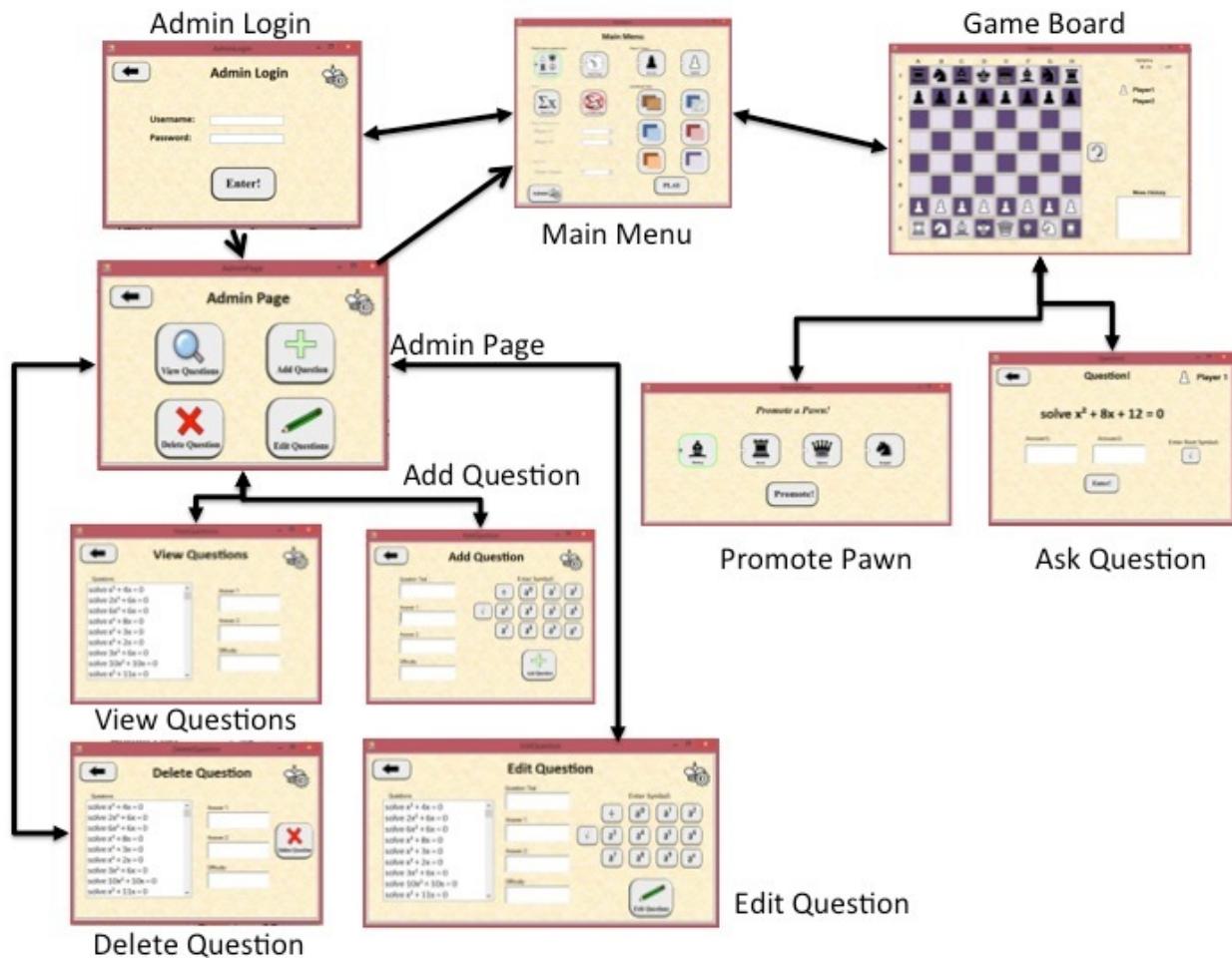
Object Orientated structure of system:

This diagram represents the final object structure of the system, but for further information on each class and individuals methods, see the design section and for further information on variables, see the variable list in this section.



**Form Navigation Diagram:**

Below shows a visual representation of the navigation between the forms in the system and how they are linked together.



## ● Procedure and Functions list

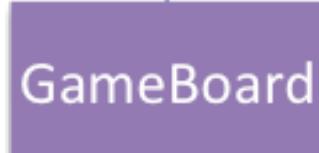


### Main Menu

Procedure/Function Name	Parameters	Purpose
<code>public MainMenu()</code>	N/A	This procedure is the standard routine run when the form is opened. Within this procedure, standard controls are set certain properties (i.e text boxes and buttons) and also runs the GatherQuestionsFromFile procedure, which retrieves the contents of the question bank file stored on the server.
<code>public void GatherQuestionsInputFromFile()</code>	N/A	This procedure gathers the questions stored in the text file stored on the server, so the program has them readily available in memory and so that communications with the text file is minimised. The procedure reads in each line of the text file and then splits the line (which is separated by commas) and stored each part of the line in the corresponding property variable of a question object. This happens which each line and a list of question objects is populated (questions list from boardAdmin class)
<code>private bool ValidateSelection(<b>ref</b> Piece.Piececolour player1Colour, <b>ref</b> String gameStyle)</code>	player1Colour, gameStyle	This procedure ensures that all appropriate selections have been made on the form. This routine also assigns the selections made for player 1 colour and game board style, by determining which selection has been made and assigning the appropriate values to the variables.

Procedure/Function Name	Parameters	Purpose
<pre>private void PlayButton_Click(object sender, EventArgs e)</pre>	Sender, e	This procedure is run when the play button on the main menu is clicked. The routine carries out some validation to ensure some options have been selected and also runs the validateSelection procedure to check other selections and to assign some variables. Then, a game object is created and its properties are populated with the appropriate data. The game board form is then opened if all the data entered is successful.
<pre>private void StandardGame_CheckedChanged_1 (object sender, EventArgs e)</pre>	Sender, e	This procedure handles the changing of the button's border, highlighting green if selected and reverting back to no highlight if the selection has changed.
<pre>private void TimedGame_CheckedChanged(object sender, EventArgs e)</pre>	Sender, e	This procedure handles the changing of the button's border, highlighting green if selected and reverting back to no highlight if the selection has changed.
<pre>private void MathsGame_CheckedChanged(object sender, EventArgs e)</pre>	Sender, e	This procedure handles the changing of the button's border, highlighting green if selected and reverting back to no highlight if the selection has changed.
<pre>private void NoMathsGame_CheckedChanged(object sender, EventArgs e)</pre>	Sender, e	This procedure handles the changing of the button's border, highlighting green if selected and reverting back to no highlight if the selection has changed.
<pre>private void Black_CheckedChanged(object sender, EventArgs e)</pre>	Sender, e	This procedure handles the changing of the button's border, highlighting green if selected and reverting back to no highlight if the selection has changed.
<pre>private void White_CheckedChanged(object sender, EventArgs e)</pre>	Sender, e	This procedure handles the changing of the button's border, highlighting green if selected and reverting back to no highlight if the selection has changed.
<pre>private void WoodStyle_CheckedChanged(object sender, EventArgs e)</pre>	Sender, e	This procedure handles the changing of the button's border, highlighting green if selected and reverting back to no highlight if the selection has changed.

Procedure/Function Name	Parameters	Purpose
<pre>private void PlainBlueStyle_CheckedChanged( object sender, EventArgs e)</pre>	Sender, e	This procedure handles the changing of the button's border, highlighting green if selected and reverting back to no highlight if the selection has changed.
<pre>private void PlainRedStyle_CheckedChanged( object sender, EventArgs e)</pre>	Sender, e	This procedure handles the changing of the button's border, highlighting green if selected and reverting back to no highlight if the selection has changed.
<pre>private void PlainBrownStyle_CheckedChanged( object sender, EventArgs e)</pre>	Sender, e	This procedure handles the changing of the button's border, highlighting green if selected and reverting back to no highlight if the selection has changed.
<pre>private void PlainPurpleStyle_CheckedChanged( object sender, EventArgs e)</pre>	Sender, e	This procedure handles the changing of the button's border, highlighting green if selected and reverting back to no highlight if the selection has changed.
<pre>private void AdminButton_Click(object sender, EventArgs e)</pre>	Sender, e	This procedure opens the admin login form, for a bridge between the main menu and the admin section of the system.



**GameBoard (BoardUI)**

Procedure/Function Name	Parameters	Purpose
<code>public BoardUI()</code>	N/A	<p>This procedure is run when the form is loaded. The first thing this procedure does is create the game using the appropriate methods in the game class, which will override depending on what game mode has been selected.</p> <p>The next thing it does, it manipulate controls on the form, corresponding to the game mode (for example, the procedure makes timer labels visible if the game mode is a timed game). This procedure also enables the updateViewTimer which is a timer used to time when the UI is updated. The procedure <b>assignBoardSquares</b> is also invoked here.</p>
<code>public void updateView()</code>	N/A	<p>This procedure is run with every timer tick of the timer updateViewTimer. The procedure loops through the array of picture boxes and sets their image to empty, then the pieces of the game are looped through and the picture boxes corresponding to their positions are set with images of that piece. This procedure also deals with the timers (if its a timed game) and updates the timer labels with the current timer values stored as properties in the player objects.</p>
<code>public void assignBoardsquares()</code>	N/A	<p>This procedure creates a link between the picture boxes on the form and an array of picture boxes. The <b>generateBoardPositions</b> procedure is invoked and then the controls on the form are looped through and the picture boxes with names corresponding the board positions array, are added to the same index in the picture box array (parallel).</p>
<code>public String[,] generateBoardPositions()</code>	N/A	<p>The Function returns a two dimensional array containing strings that represent the names and positions of squares on the board (A1,A2... C5 etc). This can then be used for reference in the system.</p>

Procedure/Function Name	Parameters	Purpose
<pre>private void Form1_Load(object sender, EventArgs e)</pre>	Sender, e	This procedure runs when the form is loaded, it combines all the click events for each picture box, into one event which is the same for all picture boxes (squares on the board).
<pre>private void UpdateViewTimer_Tick(object sender, EventArgs e)</pre>	Sender, e	This is what runs when the timer ticks. It updates the current move icon, to represent who's turn it is and it also calls the <b>updateView</b> procedure.
<pre>private void clickEvent(object sender, EventArgs e)</pre>	Sender, e	This procedure is a generic click event routine for all squares on the form. This procedure assigns the variable <b>currentPicBox</b> to store the picture box clicked and also a variable <b>clickedPiece</b> , containing the piece clicked. This is determined by invoking another procedure, <b>getPieceOnSquare</b> . Then depending on the properties of this piece and whether a piece was clicked, either the <b>pieceMovingClickEvent</b> or the <b>standardClickEvent</b> procedures will happen.
<pre>private Piece getPieceOnSquare(PictureBox clickedPicBox)</pre>	clickedPicBox	This procedure gets the piece on the square (picture box) that has been clicked by the user. The procedure starts by assigning x and y variables corresponding to the location of the picture box in the squares array. Then it does a cross reference to the same position on the game board, comparing whether a piece has the same location. If a comparison returns true, then the piece must be on the clicked square. The <b>clickedPiece</b> variable is assigned to.
<pre>private void standardClickEvent(PictureBox currentPicBox, Piece clickedPieceTemp)</pre>	currentPicBox, clickedPieceTemp	This procedure is run if a piece is clicked and it is the current player's piece. Firstly, the possible moves for that piece are generated and a possible moves array is populated. The procedure then highlights squares accordingly, depending on the option of highlighting being on or off.

Procedure/Function Name	Parameters	Purpose
<pre>private void PieceMovingClickEvent(PictureBox currentPicBox)</pre>	currentPicBox	<p>This procedure is run when a square has been clicked to move a piece to, or just a standard click. The procedure checks whether the clicked square is in the possible moves array for the piece clicked previously (if there was one) and if this is true then the move is made (invokes method from player class). The procedure also creates an askQuestion form which is presented to the user in a maths game mode. The procedure then checks whether checkmate has been reached and if so, the game result and actions are carried out accordingly.</p>
<pre>private bool isCheckMated()</pre>	N/A	<p>This function determines whether checkmate has been reached after a particular move in the game. Two null counters are used to keep track of “no possible moves” for each piece on the defending players list and if all pieces cannot move then it is check mate. This process is done by making each move in the pieces possible moves temporarily using the <b>makeTempMove</b> procedure and then evaluates whether the “check” status is still the case.</p>
<pre>private void makeTempMove(Square possiblemove)</pre>	possiblemove	<p>This procedure is used to make moves temporarily for the purposes of evaluation of that move. The procedure takes a parameter of a square object representing the position to move to. The procedure then makes the move and determines whether the player is in check. If so, then the possible move is removed from the array of possible moves or made null. The procedure then invokes the <b>undoLastMove</b> method from the game class to reverse the move made.</p>
<pre>private void resetSquarebackgrounds()</pre>	N/A	<p>This procedure quickly sets all the square background to those without green borders.</p>

Procedure/Function Name	Parameters	Purpose
<pre>private void ChangeStaticHighlighting()</pre>	N/A	This procedure is used for when the option of highlighting is changed during the game, it makes the changes to the board immediately to make the toggling of the option and its effects, dynamic. So if moves are currently highlighted, and the option is turned off, the board should change to correspond to this. If turned back on, the highlighting should reappear.
<pre>private void HighlightingON_CheckedChanged (object sender, EventArgs e)</pre>	Sender, e	Invokes the <b>changeStaticHighlighting</b> procedure.
<pre>private void HighlightingOFF_CheckedChanged (object sender, EventArgs e)</pre>	Sender, e	Invokes the <b>changeStaticHighlighting</b> procedure.
<pre>private void setDarkStyleSquares(PictureBox aPictureBox)</pre>	aPictureBox	This procedure takes a picture box parameter and then assigns its background image according to which game style has been chosen. This procedure assigns the dark squares without any highlight.
<pre>private void setLightStyleSquares(PictureBox aPictureBox)</pre>	aPictureBox	This procedure takes a picture box parameter and then assigns its background image according to which game style has been chosen. This procedure assigns the light squares without any highlight.
<pre>private void setDarkStyleSquares_Green(Pic tureBox aPictureBox)</pre>	aPictureBox	This procedure takes a picture box parameter and then assigns its background image according to which game style has been chosen. This procedure assigns the dark squares with highlight.
<pre>private void setLightStyleSquares_Green(Pi ctureBox aPictureBox)</pre>	aPictureBox	This procedure takes a picture box parameter and then assigns its background image according to which game style has been chosen. This procedure assigns the light squares with highlight.

Procedure/Function Name	Parameters	Purpose
<pre>private void updateUnicodeMoveBox(Move aMove)</pre>	aMove	This procedure updates the move history text box on the form with the details of the move that was most recently made. The Move object parameter is used and its properties are read, to determine what to append to the text box.
<pre>private void undoLastMove()</pre>	N/A	This procedure invokes the undo method in the game class but also runs the procedure updateUnicodeText_afterRemoval to update the move history box.
<pre>private void updateUnicodeText_afterRemova l()</pre>	N/A	This procedure updates the move history text box by removing the first line in the text box and then adjusting the remaining text to remove any blank lines etc.
<pre>private void undoMoveButton_Click(object sender, EventArgs e)</pre>	Sender, e	This procedure calls the undoLastMove routine when the undo buttons clicked on the game board form.

## Ask Question

### Ask Question Form

Procedure/Function Name	Parameters	Purpose
<code>public AskQuestion()</code>	N/A	Generic procedure that runs when the form is loaded. This procedure invokes the <b>gatherQuestion</b> procedure.
<code>public void gatherQuestion()</code>	N/A	This procedure gathers the question from the bank of questions to present to the user during gameplay. This procedure takes the difficulty level of the current player and creates a new questions array filled with questions only of the target difficulty. Then a randomly selected question is set as the current question and form adjusts labels to show this question
<code>private void enterButton_Click(object sender, EventArgs e)</code>	Sender, e	This procedure runs when the enter button is clicked on the form. The users answers are compared with the actual question answers and depending on the outcome, the labels of the form are updated to give feedback on the question.
<code>private void subtractTime()</code>	N/A	This procedure subtracts time from the players timer value after an incorrect answer or they have exited the question form without an answer. This is based on the level of difficulty and this procedure handles this.
<code>private void SquareRootButton_Click(object sender, EventArgs e)</code>	Sender, e	The square root button is clicked and this procedure appends the symbol to the current textbox that has focus.
<code>private void BackButton_Click(object sender, EventArgs e)</code>	Sender, e	When the back button is clicked, the question form is closed.

Procedure/Function Name	Parameters	Purpose
<pre>private void AskQuestion_FormClosing(object sender, EventArgs e)</pre>	Sender, e	Occurs when the form is closed manually. This procedure invokes the subtractTime procedure.

## Promote Pawn

### Promote Pawn

Procedure/Function Name	Parameters	Purpose
<pre>private void KnightButton_CheckedChanged(object sender, EventArgs e)</pre>	Sender, e	This procedure changes the button of the knight selection by adding highlighting to the button if it has been selected.
<pre>private void QueenButton_CheckedChanged(object sender, EventArgs e)</pre>	Sender, e	This procedure changes the button of the knight selection by adding highlighting to the button if it has been selected.
<pre>private void RookButton_CheckedChanged(object sender, EventArgs e)</pre>	Sender, e	This procedure changes the button of the knight selection by adding highlighting to the button if it has been selected.
<pre>private void BishopButton_CheckedChanged(object sender, EventArgs e)</pre>	Sender, e	This procedure changes the button of the knight selection by adding highlighting to the button if it has been selected.
<pre>private void PromoteButton_Click(object sender, EventArgs e)</pre>	Sender, e	This procedure takes the choice of piece to promote to and stores the type in the promotingPawnType variable in the boardAdmin class. This is then available to the rest of the system to process the promotion.

## Admin Login

### Admin Login

Procedure/Function Name	Parameters	Purpose
<code>public AdminLogin()</code>	N/A	This procedure initialises the form
<code>private void BackButton_Click(object sender, EventArgs e)</code>	Sender, e	This occurs when the back button is pressed, the form closes and the main menu is back in focus.
<code>private void enterButton_Click(object sender, EventArgs e)</code>	Sender, e	This invokes the <b>enterEvent</b> Procedure.
<code>private void enterEvent()</code>	N/A	This procedure checks the user input for username and password against the correct values. If correct then the admin page form is opened and given focus and if incorrect, feedback is given to the user detailing this.

## Admin Page

### Admin Page

Procedure/Function Name	Parameters	Purpose
<code>public AdminPage()</code>	N/A	This procedure initialises the form and also brings it to the front and gives it focus.

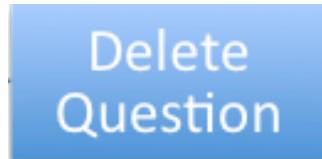
Procedure/Function Name	Parameters	Purpose
<pre>private void ViewQuestionsButton_Click(object sender, EventArgs e)</pre>	Sender, e	This procedure is run when the view questions button is clicked. It opens a new view questions form and waits for the form to be dealt with.
<pre>private void DeleteQuestionButton_Click(object sender, EventArgs e)</pre>	Sender, e	This procedure is run when the delete question button is clicked. It opens a new delete question form and waits for the dialog to be dealt with.
<pre>private void EditQuestionsButton_Click(object sender, EventArgs e)</pre>	Sender, e	This procedure is run when the edit question button is clicked. It opens a new edit question form and waits for the dialog to be dealt with.
<pre>private void AddQuestionButton_Click(object sender, EventArgs e)</pre>	Sender, e	This procedure is run when the add question button is clicked. It opens a new add question form and waits for the dialog to be dealt with.
<pre>private void ViewQuestionsButton_MouseEnter(object sender, EventArgs e)</pre>	Sender, e	This procedure changes the cursor icon to a hand icon when the mouse is over the button - the same event happens for all main buttons on the form.
<pre>private void ViewQuestionsButton_MouseLeave(object sender, EventArgs e)</pre>	Sender, e	This procedure changes the cursor icon to the default icon when the mouse leaves the button - the same event happens for all main buttons on the form.

## Add Question

### Add Question

Procedure/Function Name	Parameters	Purpose
<pre>public AddQuestion()</pre>	N/A	This procedure initialises the form but also combines the “enter” event for all text boxes on the form so they all run the same code. This is used to help append symbols to the current textbox.
<pre>private void QuestionTextBox_Enter(object sender, EventArgs e)</pre>	Sender, e	This is the procedure run for all text boxes on the form. A global variable is assigned to, to store the current textbox that has focus.

Procedure/Function Name	Parameters	Purpose
<code>private void addQuestionToFile()</code>	N/A	This procedure adds the new question to the text file stored on the server.
<code>private void AddQuestionButton_Click(object sender, EventArgs e)</code>	Sender, e	This procedure event invokes the addQuestionToFile procedure, when the add question button is clicked.
<code>private void SquareRootButton_Click(object sender, EventArgs e)</code>	Sender, e	This routine appends a square root symbol to the current focused text box on the form.
<code>private void BackButton_Click(object sender, EventArgs e)</code>	Sender, e	This procedure closes the form when the back button is clicked.
<code>private void DivideButton_Click(object sender, EventArgs e)</code>	Sender, e	This procedure appends a divide symbol to the textbox that has the current focus on the form
<code>private void powerButton_0_Click(object sender, EventArgs e)</code>	Sender, e	This procedure appends a superscript to the current text box that has focus on the form. Same procedures but for numbers up to 9 and "n" are also available.



### Delete Question

Procedure/Function Name	Parameters	Purpose
<code>public DeleteQuestion()</code>	N/A	This procedure initialises the form and also sets properties of certain controls on the form.
<code>private void populateListBox()</code>	N/A	This procedure retrieves the questions from the list of question objects and populates a list box with the question text. So that the user can select a question.

Procedure/Function Name	Parameters	Purpose
<pre>private void QuestionTextList_SelectedIndexChangedValueChanged(object sender, EventArgs e)</pre>	Sender, e	This happens when the question selected by the user in the list box control changes. The procedure updates the other text boxes that are view on the form by changing their contents to match the properties of the newly selected question.
<pre>private void BackButton_Click(object sender, EventArgs e)</pre>	Sender, e	This procedure closes the form when the back button is clicked by the user.
<pre>private void deleteQuestionFromFile()</pre>	N/A	This procedure carries out the deletion of the question from the question bank file on the server. This is done by removing the question from the list of questions in memory then overwriting the file with the updated list.
<pre>private void DeleteQuestionButton_Click(object sender, EventArgs e)</pre>	Sender, e	This procedure invokes the deleteQuestionFromFile procedure when the delete question button is clicked. A warning is given to the user in the form of a message box YES/NO.

## Edit Question

### Edit Question

Procedure/Function Name	Parameters	Purpose
<pre>public EditQuestion()</pre>	N/A	This procedure initialises the form and also combines the textbox “enter” event so that they all run the same code.
<pre>private void populateListBox()</pre>	N/A	This procedure retrieves the questions from the list of question objects and populates a list box with the question text. So that the user can select a question.

Procedure/Function Name	Parameters	Purpose
<pre>private void SquareRootButton_Click(object sender, EventArgs e)</pre>	Sender, e	This routine appends a square root symbol to the current focused text box on the form.
<pre>private void BackButton_Click(object sender, EventArgs e)</pre>	Sender, e	This procedure closes the form when the back button is clicked.
<pre>private void DivideButton_Click(object sender, EventArgs e)</pre>	Sender, e	This procedure appends a divide symbol to the textbox that has the current focus on the form
<pre>private void powerButton_0_Click(object sender, EventArgs e)</pre>	Sender, e	This procedure appends a superscript to the current text box that has focus on the form. Same procedures but for numbers up to 9 and "n" are also available.
<pre>private void EditQuestionsButton_Click(object sender, EventArgs e)</pre>	Sender, e	This performs the editing of the question and makes changes to the question list in memory and also to the file on the server. This happens when the edit button is clicked.
<pre>private void QuestionTextList_SelectedIndexChanged(object sender, EventArgs e)</pre>	Sender, e	This happens when the question selected by the user in the list box control changes. The procedure updates the other text boxes that are view on the form by changing their contents to match the properties of the newly selected question.

## View Questions

### **View Questions**

Procedure/Function Name	Parameters	Purpose
<pre>public ViewQuestions()</pre>	N/A	This procedure initialise the form and also sets certain properties of some controls on the form. This also invokes the populate list box procedure.
<pre>private void populateListBox()</pre>	N/A	This procedure retrieves the questions from the list of question objects and populates a list box with the question text. So that the user can select a question.
<pre>private void QuestionTextList_SelectedIndexChanged(<b>object</b> sender, <b>EventArgs</b> e)</pre>	Sender, e	This happens when the question selected by the user in the list box control changes. The procedure updates the other text boxes that are view on the form by changing their contents to match the properties of the newly selected question.
<pre>private void BackButton_Click(<b>object</b> sender, <b>EventArgs</b> e)</pre>	Sender, e	This procedure closes the form when the back button is clicked.

**Note:**

For details on procedures and functions within the object orientation in the system - mainly associated with the chess engine - see the design section of the project where methods and private variables are explained in detail.

- **Applications used in my project**

- **Microsoft Visual Studio (2012)**



Visual studio was the application mainly used during the project. Microsoft's development environment was used as the platform for me to create the project and code the software in visual c#.

Visual studio was used to write the code for the program but it was also used to create the user interfaces as well. Using windows forms in visual studio accompanied the code and combined to make my system. I made use of controls available in the environment, such as: rich text boxes, list boxes, numerical up and downs, combo boxes and various other standard controls.

Using visual studio as a platform actually increased efficiency, as i was familiar with the software but also because visual studio is very clean and easy to use. The automatic indentation kept my code clear while programming and clear layout of tools and components in the system made the project easy to manage.

- **Apple Pages (v5.0)**



Apple pages was the software i used to document the project and showcase all details and designs of the project as i worked on it.

The reasoning for using apples words processor over microsofts was due to the extremely ease of placing objects and images and diagrams around text, a much more flexible design choice over microsoft word which is very strict and you cannot move around objects as freely.

A disadvantage i found however, i had to sacrifice some formatting and had some problems combining documents as the contents of the document shifted and changed if pages were added between other pages for example, which meant spending more time sorting out the document layout than i should have.

However, i stick with my original choice for using apple pages as the pro's of being able to place objects freely made the documentation easier to showcase, despite any problems i had with the software.

- **Microsoft Powerpoint**



I used Microsoft powerpoint over apple keynote, however, as apple haven't quite perfected their vector drawing program just yet and i was more familiar with Microsoft's option. Some of the diagrams and flowcharts i have in my project were thanks to powerpoint and its easy to create charts and simple vector graphics.

I also used Microsoft powerpoint, quite heavily, for annotations to screenshots - creating documents of over 50 slides - to annotate and explain UI designs, test screenshots and other documentations. Powerpoint made easy work of the task with

its quick shapes, allowing me to annotate effectively. An extremely helpful feature of exporting slides as PNG pictures was also a deal breaker as i could easily place the slides as pictures onto Apple pages for my documentation.

- **Serif DrawPlus (x3)**



To accompany Microsoft powerpoint and its vector graphics processing, i also made use of Serif DrawPlus - a graphics studio software. I used DrawPlus to easily create all my buttons used in the project effectively.

DrawPlus proved effective when it came to slightly more fiddly vector graphics tasks such as making buttons for the project. I could cut out portions of an image easily or combine images with no fuss. I could also add some subtle, but effective bevels and filter effects on the button designs.

Serif was also very useful for testing button and form designs with its easy to use graphics pages.

- **Apple Text Edit / Windows NotePad**



I used both text edit and notepad for the creation of the question bank text file. As i worked across multiple operating systems, both applications were used. I used the simple text editors to create the lines of text which represented the questions to be communicated with by the program.

I created a comma separated values file (CSV) so i could store information and data logically and read/write from the file effectively.

- **Gliffy online diagram creator**



I used gliffy to create slightly more complex diagrams, that powerpoint could not cope with, such as the system object diagram and also the system flowchart and network diagrams. This free online application was extremely easy to use and provided many easy diagram tools and shapes which i could combine and attach together extremely efficiently. I

could then export diagrams as PNG's.

The main diagrams made with gliffy were the object orientated diagrams, the network diagrams and the system flowchart.

## ● Annotated Code Reference

I annotated my code as i wrote it, to make sure that each routine was clear and also to make things easier for me because when the size of the project started to grow, it became clear that it become harder to keep track of everything, so annotating with comments as i went along proved an effective technical solution skill.

Please reference the code given in the ***Technical solution*** of the project to see the commentary of the code. Please note that there are routines that repeat in other modules of the system so i have not made the same commentary for all of them, only the once was sufficient.

## ● Any code not written by myself

I wrote all code used in the project. I did however, use built in functions such as the “split” command for arrays or the “clear” function for text boxes etc but other than the fundamental built in functions of the programming language, all code was written by myself.

## ● Detailed Algorithm Annotation

### MakeTempMove Algorithm:

```
// the procedure to make a move temporarily to evaluate something then retract that move
private void makeTempMove(Square possiblemove)
{
    BoardAdmin.TempMoving = true;
// sets the global variable stored in the boardAdmin class the true, so the other routines
// know that the move is temporary

    if (BoardAdmin.game1.getTurn() ==
BoardAdmin.game1.getPlayer1().getPlayerColour())
// checks whether the current turn is player 1's
    {

        BoardAdmin.game1.getPlayer1().makeMove(possiblemove, gameBoard,
clickedPiece);
// player 1 makes the temporary move using the makeMove method

        if (BoardAdmin.game1.getPlayer1().isInCheck(gameBoard) == true)
// the isInCheck method is invoked to determine whether the player is in check
        {
            int index = Array.IndexOf(possiblemoves, possiblemove);
// gets the index of the possible move which has just been made temporarily
            possiblemoves[index] = null;
// makes that possible move null as the move is invalid -- results in check situation
        }

        BoardAdmin.game1.undoLastMove();
// invokes the undoLastMove method of the game class to undo the move that has just been
made
    }
}
```

```

        else // otherwise it is player 2's turn
        {

            BoardAdmin.game1.getPlayer2().makeMove(possiblemove, gameBoard,
clickedPiece);
// player 2 makes the temporary move using the makeMove method

            if (BoardAdmin.game1.getPlayer2().isInCheck(gameBoard) == true)
// the isInCheck method is invoked to determine whether the player is in check
            {
                int index = Array.IndexOf(possiblemoves, possiblemove);
// gets the index of the possible move which has just been made temporarily
                possiblemoves[index] = null;
// makes that possible move null as the move is invalid -- results in check situation
            }

            BoardAdmin.game1.undoLastMove();
// invokes the undoLastMove method of the game class to undo the move that has just been
made

        }

        BoardAdmin.TempMoving = false;
// After the temporary move is made, the global variable is set to false, to let other
routines
// know that any move being made is no longer temporary
    }
}

```

**IsCheckMated Algorithm:**

```

// this checks whether the current player is in the checkmate game state
    private bool isCheckMated()
    {
//the clicked piece variable is going to change as a result of a temporary move
// so it is stored temporarily to return to the current game state
        Piece tempPieceHolder = clickedPiece;
        int nullCounter = 0;
// a null counter to count all the pieces that have no possible moves

        if (BoardAdmin.game1.getTurn() == Piece.Piececolour.WHITE)
// runs to check whether WHITE is in checkmate
        {
            PieceSet WhitePieces = gameBoard.getPieceSets()[1];
// creates a pieceset variable for white pieces.
            foreach (Piece aPiece in WhitePieces.getPieceSet())
// uses variable in loop to evaluate each piece.
            {
                possiblemoves = aPiece.getPossibleMoves(gameBoard);
// assigns to the possible moves array, the possible moves of the current piece being
evaluated
                clickedPiece = aPiece;
// assigned the clicked piece variable to the piece currently being evaluated.

//for every piece in the players piece list, the move is made and it is evaluated whether
// the player is still in check
                foreach (Square possiblemove in possiblemoves)
// loop through each possible move to make a temporary move from that possible move.
                {
                    if (possiblemove != aPiece.getPosition())
// if the possible move is not equal to the pieces current position then true
                    {
                        makeTempMove(possiblemove);
// makes the temporary move from that possible move in the list, the makeTempMove algorithm

```

```

will make any non-valid moves null.
        }
        else
    // otherwise, make the move null as you cant move to the current square.
    {
        int index = Array.IndexOf(possiblemoves, possiblemove);
    // gets index of the position of the possible move
        possiblemoves[index] = null;
    // makes the move null

    }

bool a = false;
// a boolean flag to store a simple result of whether all moves are invalid.
int nullcounter2 = 0;

// a counter to count how many possible moves are null for an individual piece
// for each possible move of the current piece, if the move is null the update the second
null counter
foreach (Square possiblemove in possiblemoves)
{
    if (possiblemove == null)
// if the possible move is null
    {
        nullcounter2++;
// increment the second counter to say that there is one more null move.
    }
}
// if ALL possible moves are null then set the boolean flag to true
if (nullcounter2 == possiblemoves.Length)
{
    a = true;
}

// otherwise set the boolean flag to false
if (nullcounter2 != possiblemoves.Length)
{
    a = false;
}
// if the boolean flag is true then that piece has no possible moves, so update the first
null counter
if (a == true)
{
    nullCounter++;
}

//clear the possible moves array for the next piece in the for loop
Array.Clear(possiblemoves, 0, possiblemoves.Length);

}

clickedPiece = tempPieceHolder;
// returns the original clicked piece

// if the first null counter is equal to the number of pieces in the player's list
// then that means NONE of the pieces have any possible moves and so this is checkmate.
if (nullCounter == WhitePieces.getPieceSet().Count)
{
    return true;
}
else
{
    return false;
}
}

```

```

        else
// runs to check whether BLACK is in check mate then runs the same code but for black pieces
specifically.
{
    PieceSet BlackPieces = gameBoard.getPieceSets()[0];
    foreach (Piece aPiece in BlackPieces.getPieceSet())
    {
        possiblemoves = aPiece.getPossibleMoves(gameBoard);
        clickedPiece = aPiece;

        foreach (Square possiblemove in possiblemoves)
        {
            if (possiblemove != aPiece.getPosition())
            {
                makeTempMove(possiblemove);
            }
            else
            {
                int index = Array.IndexOf(possiblemoves, possiblemove);
                possiblemoves[index] = null;
            }
        }
        bool a = false;
        int nullcounter2 = 0;

        foreach (Square possiblemove in possiblemoves)
        {
            if (possiblemove == null)
            {
                nullcounter2++;
            }
        }

        if (nullcounter2 == possiblemoves.Length)
        {
            a = true;
        }
        if (nullcounter2 != possiblemoves.Length)
        {
            a = false;
        }

        if (a == true)
        {
            nullCounter++;
        }

        Array.Clear(possiblemoves, 0, possiblemoves.Length);
    }
    clickedPiece = tempPieceHolder;
    if (nullCounter == BlackPieces.getPieceSet().Count)
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

}

**IsInCheck Algorithm:**

```

// this is an algorithm to determine whether the player is in check
// this is achieved by searching the other players pieces and seeing whether the friendly
king
// is occupying one of the other players piece's possible moves
    public bool isInCheck(Board gameBoard)
    {
        PieceSet[] pieceSets = gameBoard.getPieceSets();
        // gets the piece sets from the gameboard property of the game class

        PieceSet BlackPieces = pieceSets[0];
        // assigns the black pieces into a seperate variable for ease

        PieceSet WhitePieces = pieceSets[1];
        // assigns the white pieces into a seperate variable for ease

        Piece kingToCheck = null;
        // creates a piece variable to store the king piece for use in the routine

        if (this.getPlayerColour() == Piece.Piececolour.BLACK)
        // if the player is using black pieces
        {
            foreach (Piece piece in BlackPieces.getPieceSet())
            // loop through the black pieces
            {
                if (piece.getType() == Piece.pieceType.KING)
                // if the piece is of type king then do the following
                {
                    kingToCheck = piece;
                    // assign the piece variable with the king piece found
                }
            }
        }

        if (this.getPlayerColour() == Piece.Piececolour.WHITE)
        // if the player is using white pieces
        {
            foreach (Piece piece in WhitePieces.getPieceSet())
            // loop through the white pieces
            {
                if (piece.getType() == Piece.pieceType.KING)
                // until a king piece is found
                {
                    kingToCheck = piece;
                    // set the piece variable with this king piece.
                }
            }
        }

        if (kingToCheck != null)
        // makes sure that the piece is not null.
        {
            if (this.getPlayerColour() == Piece.Piececolour.BLACK)
            // if the player is using black pieces
            {
                foreach(Piece piece in WhitePieces.getPieceSet())

```

```

// scan through the white pieces
{
    if
(piece.getPossibleMoves(gameBoard).Contains(kingToCheck.getPosition()))
// populates each pieces possible moves and then scans those possible moves to see whether
the kingToCheck has the same position as one of those moves

    {
        return true;
    // if so then the king is in check so return true
    }
}
return false;
// otherwise return false as no pieces have a move that has the position of the king
}
else // otherwise the player is using black pieces
{
    foreach (Piece piece in BlackPieces.getPieceSet())
//scan through the black pieces
{
    if
(piece.getPossibleMoves(gameBoard).Contains(kingToCheck.getPosition()))
{
// populates each pieces possible moves and then scans those possible moves to see whether
the kingToCheck has the same position as one of those moves

        return true;
    // if so then the king is in check so return true
    }
}
return false;
// otherwise return false as no pieces have a move that has the position of the king
}

}
return false;
// if the piece is null then return false as their is no comparisons made.
}

```

### CheckMoveIsValid Algorithm (Queen overriding):

```

// override for queen
public override bool checkMoveIsValid(Square targetPosition, Board gameBoard)
{
    Square[,] board = gameBoard.getSquares();
// creates an easy accessible variable of the board squares

    if (base.checkMoveIsValid(targetPosition, gameBoard) == false)
// performs the generic checkMoveIsValid procedure first
    {
        return false;
    }
    else
// otherwise the overriding for the queen happens
    {
        int targetYindex = targetPosition.Y, targetXindex = targetPosition.X,
currentYindex = this.getPosition().Y, currentXindex = this.getPosition().X;
// creates variables to store the current coordinates and the target coordinates
    }
}

```

```

// a queen's movement is both horizontal and diagonal so checks must be made for both cases
// for loops are used to check subsequent squares between the queen and the target position
    if (targetYindex == currentYindex || targetXindex == currentXindex ||
Math.Abs(targetXindex - currentXindex) == Math.Abs(targetYindex - currentYindex))
    {
// checks whether the target square is either horizontal or diagonal movement

        if (targetYindex == currentYindex || targetXindex == currentXindex)
        {
// then checks whether the movement is specifically horizontal
            if (targetYindex == currentYindex)
            {
// if the y coordinates are the same the horizontal movement is in the x-direction
                if ((targetXindex - currentXindex) < 0)
                {
// checks whether the movement is to the left
                    for (int i = currentXindex - 1; i >= targetXindex; i--)
                    {
// if so then loop through between the square above the piece and the target position
                        if (board[i, currentYindex ].occ == true)
                        {
// if a particular square is occupied then the move to the target square cannot be made
                            if (i == targetXindex)
                            {
// if the current square being assessed is the target square
                                if (canCapture(board[i, currentYindex ],
gameBoard) == true)
                                {
// invoke the canCapture procedure to determine whether capturing can take place
                                    return true;
                                }
                            }
                            return false;
                        }
// if false then capturing cannot take place
                    }
                }
            }
            else
// otherwise the movement is to the right
            {
                for (int i = currentXindex + 1; i <= targetXindex; i++)
{
// loop through, checking the squares between the current position, starting at the square
one to the right of the current position and then through to the target position

                    if (board[i, currentYindex ].occ == true)
                    {
//if a the current square being assessed is occupied
                        if (i == targetXindex)
                        {
// if the current square being assessed is the target square
                            if (canCapture(board[i, currentYindex ],
gameBoard) == true)
                            {
// invoke canCapture procedure to see whether the occupants of the square can be captured
                                return true;
                            }
                        }
                    }
                    return false;
// otherwise the current square is occupied so the target square cannot be reached.
                }
            }
        }
    }
}

```

```

        else
    // otherwise the movement is in the y direction
    {
        if ((targetYindex - currentYindex) < 0)
        {
    // checks whether the movement is upwards

            for (int i = currentYindex - 1; i >= targetYindex; i--)
            {

    // loop through the squares between the current square and the target square, starting at
    // the square one above the current square
                    if (board[currentXindex,i].occ == true)
                    {
    // checks whether the square is occupied
                        if (i == targetYindex)
                        {
    // checks whether the current square being assessed is the target square
                                if (canCapture(board[currentXindex, i],
gameBoard) == true)
    // if so then check whether the occupants can be captured
                                {
                                    return true;
                                }
                            }
                        }
                    }
                }
            }
        }
    }

// otherwise return false as the target square cannot be reached.
}

}

else
// otherwise the movement is downwards
{
    for (int i = currentYindex + 1; i <= targetYindex; i++)
    {

    // loop through the squares between the current square and the target square, starting at
    // the square one below the current square
        if (board[currentXindex, i].occ == true)
        {
    // checks whether the square is occupied
            if (i == targetYindex)
            {
    //checks whether the current square being assessed is the target square
                if (canCapture(board[currentXindex, i],
gameBoard) == true)
    // if so then check whether the occupants can be captured
                {
                    return true;
                }
            }
        }
    }
}

// otherwise return false as the target square cannot be reached.
}

}

}

if (Math.Abs(targetXindex - currentXindex) == Math.Abs(targetYindex -
currentYindex))
//checks whether the movement is diagonal (equal increments of y and x) by using the
absolute function
{
    if ((targetXindex - currentXindex) < 0 && (targetYindex -

```





```

        if (board[currentXindex, currentYindex].occ == true)
// checks whether the current square is occupied
{
    if (currentXindex == targetXindex && currentYindex
== targetYindex)
// checks whether the current square is the target square
{
    if (canCapture(board[currentXindex,
currentYindex], gameBoard) == true)
// can capturing take place on the target square?
{
    return true;
}
// return true is capturing can take place
}
}
return false;
// otherwise return false as the target square cannot be reached
}
}
}

return true;
// return true for the current position of the queen
}
else
{
    return false;
}
// return false as the target square is not horizontal or diagonal away from the current
piece.
}
}
}

```

**GatherQuestionsFromFile Algorithm:**

```

// retrieves the contents of the text file and splits each line up and populates a question
object
public void GatherQuestionsInputFromFile()
{
    const string filename = "QuestionBank 2.csv";
//FilePath of the question textfile.

    StreamReader aSW = new StreamReader(File.Open(filename, FileMode.Open));
// opens a new stream reader which can read from the file

    BoardAdmin.questions = new List<QuestionBank>();
// initiates the list of questions variable in the board admin class

    while (aSW.EndOfStream == false)
// happens until the end of the file has been reached
{
    string[] a = aSW.ReadLine().Split(',');
// read a line in from the file and split the data from that line based on the comma

```

separation, store each portion in a string array

```
// adds a question object at the same time as populating that object using the overridden
constructor
    BoardAdmin.questions.Add(new QuestionBank(a[0], a[1], a[2],
Convert.ToInt16(a[3])));
}

aSW.Close(); // closes the stream
}
```

### **MakeMove Algorithm:**

```
// this is where the moves of the game are made, and where the moves are added to history
public void makeMove(Square targetposition, Board gameBoard, Piece movingPiece)
{

    PieceSet[] pieceSets = gameBoard.getPieceSets();
// retrieves piece sets from the gameboard
    PieceSet BlackPieces = pieceSets[0];
// stores black pieces in an easily accessible variable
    PieceSet WhitePieces = pieceSets[1];
// stores white pieces in an easily accessible variable

//Move object created and populated
    Move aMove = new Move();
// creates a new move object to add to the game history

    aMove.finalPosition = targetposition;
// sets the final position to the target position - the square being moved to.

    aMove.initialPosition = movingPiece.getPosition();
// sets the initial position to the current position of the piece being moved

    aMove.PieceMoved = movingPiece;
// sets the piece moved to the current piece.

    if (BoardAdmin.game1.getTurn() ==
BoardAdmin.game1.getPlayer1().getPlayerColour())
// checks whether it is player 1's turn
    {
        aMove.PlayerMoved = BoardAdmin.game1.getPlayer1();
// sets the playerMoved property to player 1

        if (BoardAdmin.game1.getPlayer1().getPlayerColour() ==
Piece.Piececolour.WHITE)
//checks whether player 1 is playing with white pieces.
        {
            foreach (Piece piece in gameBoard.getPieceSets()[0].getPieceSet())
// loops through the black pieces
            {
                if (piece.getPosition() == targetposition)
// checks whether a black piece is in the target position of the move being made
                {
                    aMove.pieceCaptured = piece;
// if so then set the pieceCaptured property to the piece occupying the square
                }
            }
        }
    }
}
```

```

        BlackPieces.getPieceSet().Remove(aMove.pieceCaptured);
// remove the piece on the target square as it is being captured

    }

    if (BoardAdmin.game1.getPlayer1().getPlayerColour() ==
Piece.Piececolour.BLACK)
// checks whether player 1 is playing with black pieces
{
    foreach (Piece piece in gameBoard.getPieceSets()[1].getPieceSet())
// loops through the white pieces
    {
        if (piece.getPosition() == targetposition)
// checks whether piece occupies the target square in the move
        {
            aMove.pieceCaptured = piece;
// if so then store this piece in the move object property
        }
    }

    WhitePieces.getPieceSet().Remove(aMove.pieceCaptured);
// remove the piece as it has been captured
}

else
// otherwise it is player 2's turn so do exactly the same but for player 2
{
    aMove.PlayerMoved = BoardAdmin.game1.getPlayer2();

    if (BoardAdmin.game1.getPlayer2().getPlayerColour() ==
Piece.Piececolour.WHITE)
    {
        foreach (Piece piece in gameBoard.getPieceSets()[0].getPieceSet())
        {
            if (piece.getPosition() == targetposition)
            {
                aMove.pieceCaptured = piece;
            }
        }
        BlackPieces.getPieceSet().Remove(aMove.pieceCaptured);
    }
    if (BoardAdmin.game1.getPlayer2().getPlayerColour() ==
Piece.Piececolour.BLACK)
    {
        foreach (Piece piece in gameBoard.getPieceSets()[1].getPieceSet())
        {
            if (piece.getPosition() == targetposition)
            {
                aMove.pieceCaptured = piece;
            }
        }
        WhitePieces.getPieceSet().Remove(aMove.pieceCaptured);
    }
}

if (movingPiece.getType() == Piece.pieceType.KING)
// if the moving piece is a king, castling may occur
{
    King movingKing = (King)movingPiece;
// creates a piece object to store the king temporarily
    if (movingKing.canCastle(targetposition, gameBoard) == true)

```

```

// invokes the canCastle method of the King class to check whether castling can occur
{

    BoardAdmin.isCastling = true;
// if so then set the boolean variable to true to let other modules know that castling is
happening.

    Rook movingRook= null;
// create a rook object to represent the rook being moved in the castling move.

    if(this.getPlayerColour() == Piece.Piececolour.WHITE)
//if the player is moving white pieces then loop through the white pieces for the kingside
rook
    {
        foreach(Piece piece in WhitePieces.getPieceSet())
        {
            if(piece.getType() == Piece.pieceType.ROOK &&
Math.Abs(piece.getPosition().X - movingKing.getPosition().X) == 3)
// the kingside rook will be 3 x coordinates away from the king
            {
                movingRook = (Rook)piece;
// set the rook object
            }
        }
    }
    if(this.getPlayerColour() == Piece.Piececolour.BLACK )
// if the player is moving black pieces then perform the same code but in terms of black
pieces.
    {
        foreach(Piece piece in BlackPieces.getPieceSet())
        {
            if(piece.getType() == Piece.pieceType.ROOK &&
Math.Abs(piece.getPosition().X - movingKing.getPosition().X) == 3)
            {
                movingRook = (Rook)piece;
            }
        }
    }
    BoardAdmin.KingsideRook = movingRook;
// sets the global variable with the kingside rook piece found.

    movingKing.PerformCastle(BoardAdmin.KingsideRook, movingKing,
targetposition, gameBoard);
// invokes the performCastle method in the King class to make the castling move.

    if (BoardAdmin.TempMoving == false)
// if temp moving is true, then the move needs to be retracted so the boolean representing
castling needs to remain true otherwise the kingside rook will not be moved back as well as
the king
    {
        BoardAdmin.isCastling = false;
// so if temp moving is false, the castling can be made false as well as the move is not
temporary.
    }
}

movingPiece.getPosition().occ = false;
// sets the current square to unoccupied

movingPiece.setPosition(targetposition);
// sets the position of the moving piece to the new square

movingPiece.getPosition().occ = true;

```

```

// sets that square to occupied as it is now occupied by the piece moving.

    movingPiece.IncrementMoveCount(1);
// increments the move count integer of that piece.

    if (movingPiece.getType() == Piece.pieceType.PAWN)
// if the moving piece is a pawn, promotion may occur
    {

        if (movingPiece.getPosition().Y == 0 || movingPiece.getPosition().Y == 7)
//if either the bottom or the top of the board is reached
        {
            if (BoardAdmin.TempMoving == false)
// only promotes pawn if it is not a temporary move
            {

                movingPiece.promotePawn();
// invokes the promotePawn method to perform the move.
            }
        }
    }

    BoardAdmin.game1.addTomoveHistory(aMove);
// adds the populated move object to the list of move objects representing the game's
history
}

```

### CanCastle Algorithm:

```

// evaluates whether the king can perform the castle procedure by finding the kingside rook
and determining
// whether both the king and rook havent moved yet and whether the spaces between them are
empty
    public bool canCastle(Square targetPosition, Board gameBoard)
    {
// if the target position is castling then the square will be 2 x - coordinates away
        if (Math.Abs(targetPosition.X - this.getPosition().X) == 2)
        {
            PieceSet blackpieces = gameBoard.getPieceSets()[0]; // stores black pieces
            PieceSet whitepieces = gameBoard.getPieceSets()[1]; // stores white pieces

            Rook kingsideRook = null; // creates a piece to store the kingside rook

            if(this.getColour() == Piececolour.BLACK)
// checks whether the king is black
            {
                foreach (Piece piece in blackpieces.getPieceSet())
// scans the black pieces to find the rook
                {
                    if (piece.getType() == pieceType.ROOK)
                    {
                        if (Math.Abs(this.getPosition().X - piece.getPosition().X) == 3)
// checks that the rook is 3 x-coordinates from the king (kingside rook)
                        {
                            kingsideRook = (Rook)piece; // stores the rook
                        }
                    }
                }
            }
        }
    }

```

```

        }

    }

    if (this.getColour() == Piececolour.WHITE)
// checks whether the king is white
{
    foreach (Piece piece in whitepieces.getPieceSet())
// scans white pieces for the whitepieces
    {
        if (piece.getType() == pieceType.ROOK)
// if the piece is a rook
        {
            if (Math.Abs(this.getPosition().X - piece.getPosition().X) == 3)
// check that the rook is 3 x-coordinates away
            {
                kingsideRook = (Rook)piece; // store the rook piece
            }
        }
    }

    if (kingsideRook != null)
// makes sure that the stored piece is not null
    {
        if (kingsideRook.getMoveCount() == 0)
// makes sure that the rook has not moved yet ( a condition of the move)
        {
            if (this.getMoveCount() == 0)
// makes sure that the king has not moved yet ( a condition of the move)
            {
                if (targetPosition.Y == this.getPosition().Y &&
targetPosition.occ != true)
// checks that the y coordinate of the target square is the same as the king's and also
that the target square is not occupied
                {

                    for (int i = this.getPosition().X - 1 ; i >
kingsideRook.getPosition().X; i--)
// loops through between the rook and the king, to check the squares in between to make sure
they are not occupied.
                    {
                        if (gameBoard.getSquares()[i, this.getPosition().Y].occ == true)
// if a square inbetween is occupied then castling can not occur
                        {
                            return false; // so return false
                        }
                    }
                    if (targetPosition.X < this.getPosition().X)
// ensures that the algorithm only returns true for the correct target square ( in between
rook and king ) and not the other side of the king
                    {
                        return true; // return true if this condition is met
                    }
                    else
                    {
                        return false; // otherwise return false
                    }
                }
            }
        }
    }
}

// otherwise, if the square is occupied or the y coordinates are not the same
{
    return false; // return false
}

```

```
        }
        else // if the king has already moved then return false
        {
            return false;
        }
    }
    else // if the rook has already moved before then return false
    {
        return false;
    }
}
else // if the kingside rook variable is null then return false
{
    return false;
}

}
else // if the target square is not 2 x-coordinates away then return false
{
    return false;
}
}
```

### **PerformCastle Algorithm:**

```
// performs the castling procedure
// moves the kingside rook as the standard makeMove algorithm in the player class handles
the king piece as that is the primary piece involved in the move and is the piece that is
clicked to perform the castle move.
    public void PerformCastle(Rook kingsideRook, King movingKing, Square targetPosition,
Board gameBoard)
    {
// sets the target position for the rook
        Square rookTargetPosition = gameBoard.getSquares()[targetPosition.X
+1,targetPosition.Y];
// the target position for the rook is 1 x-coordinate over the king's target position

//creates a move object for the rook and populates it
        Move aMove = new Move();

        aMove.initialPosition = kingsideRook.getPosition();
// the initial position is the rooks current position

        aMove.finalPosition = rookTargetPosition;
// the final position is the rooks target position

        aMove.PieceMoved = kingsideRook;
// the piece moving is the rook

        if (kingsideRook.getMoveCount() == 0)
// checks that the rook hasn't moved yet
        {
            aMove.castled = true;
// set the castled boolean to true for the move
        }
        if (BoardAdmin.game1.getTurn() ==
BoardAdmin.game1.getPlayer1().getPlayerColour())
// checks whether it is player 1's turn
        {
```

```

        aMove.PlayerMoved = BoardAdmin.game1.getPlayer1();
// sets playermoved property to player 1
    }
else
// otherwise player 2 is moving
{
    aMove.PlayerMoved = BoardAdmin.game1.getPlayer2();
// sets playerMoved Property to player2

}

BoardAdmin.game1.addTomoveHistory(aMove);
// adds the move object to the list of moves for the game history

kingsideRook.getPosition().occ = false;
// sets the initial position as unoccupied (false)

kingsideRook.setPosition(rookTargetPosition);
// sets the new position of the rook

kingsideRook.getPosition().occ = true;
// sets the new square as occupied (true)

kingsideRook.IncrementMoveCount(1);
// increments the move count for the rook

}

```

### undoLastMove Algorithm (Partially Recursive) :

```

// method to undo last move
// uses the top-most MOVE object in the game objects moveHistory list property to undo the
move
public void undoLastMove()
{

    if (_MoveHistory.Count != 0)
//Ensures that the move history list is not empty before trying to remove a previous move
object
    {

        Move lastMove = getMoveHistory().Last<Move>();
//Creates a move object variable to access the last move made in the game

        if (_turn == Piece.Piececolour.BLACK)
//check whether it is black's turn
        {
            foreach (Piece apiece in gameBoard.getPieceSets()[1].getPieceSet())
//Scan the pieces to check the piece that is now in the last moved position
            {
                if (apiece.getPosition() == lastMove.PieceMoved.getPosition())
// incase of pawn promotion, the new piece will have the same position
                {
                    if (apiece.getType() != lastMove.PieceMoved.getType())
//if the piece has the same position as the last moved piece then this could have been pawn
promotion
                    {
//Therefore remove the piece found and replace with the old piece in the move object
                        gameBoard.getPieceSets()[1].getPieceSet().Remove(apiece);
                        gameBoard.getPieceSets()
[1].getPieceSet().Add(lastMove.PieceMoved);

```

```

                break;
            }
        }
    }
}

if (m_turn == Piece.Piececolour.WHITE) // same for white pieces
{
    foreach (Piece apiece in gameBoard.getPieceSets()[0].getPieceSet())
//Scan the pieces to check the piece that is now in the last moved position
    {
        if (apiece.getPosition() == lastMove.PieceMoved.getPosition())
// incase of pawn promotion, the new piece will have the same position
        {
            if (apiece.getType() != lastMove.PieceMoved.getType())
//if the piece has the same position as the last moved piece then this could have been pawn
promotion
            {
//Therefore remove the piece found and replace with the old piece in the move object
                gameBoard.getPieceSets()[0].getPieceSet().Remove(apiece);
                gameBoard.getPieceSets()
[0].getPieceSet().Add(lastMove.PieceMoved);
            }
        }
    }
}

// uses properties of move object to reverse the effects of the move
lastMove.PieceMoved.setPosition(lastMove.initialPosition);

// sets the position of the piece to the initial position in that move
lastMove.PieceMoved.decrementMoveCount(1);

// decrements the move count for that piece
lastMove.initialPosition.occ = true;

// the initial square will now be occupied
lastMove.finalPosition.occ = false;

// the final square will now be unoccupied.

this.setTurn(lastMove.PlayerMoved.getPlayerColour());
// The turn of the game goes back to the other player

if (lastMove.pieceCaptured != null)
// if a piece was captured in the move then this piece needs to be returned

{
    Piece pieceToReturn = lastMove.pieceCaptured;
// stores this piece in a variable

    if (pieceToReturn.getColour() == Piece.Piececolour.BLACK)
// checks whether the piece is black
    {
        m_gameBoard.getPieceSets()[0].getPieceSet().Add(pieceToReturn);
        pieceToReturn.getPosition().occ = true;
// adds the piece back to the corresponding piece list and sets the square position to
occupied.

    }
    else
    {
// same for white piece list

```

```

        m_gameBoard.getPieceSets()[1].getPieceSet().Add(pieceToReturn);
        pieceToReturn.getPosition().occ = true;
    }

// if the previous move was a castling procedure then the rook needs to be moved back also
// this is only for temporary movement
    if (BoardAdmin.isCastling == true)
    {

        if (BoardAdmin.TempMoving == true)
// If the movement was temporary

        {
            if (BoardAdmin.KingsideRook != null)
// if the kingside rook variable is not empty then the rook needs to be moved back
            {
                BoardAdmin.KingsideRook.getPosition().occ = false;
// sets the current position of the rook to unoccupied as its moving back

                BoardAdmin.KingsideRook.setPosition(gameBoard.getSquares()
[BoardAdmin.KingsideRook.getPosition().X - 2, BoardAdmin.KingsideRook.getPosition().Y]);

// sets the position of the rook back to the initial position which is two x coordinates
lower than the current square
                BoardAdmin.KingsideRook.getPosition().occ = true;
// sets the new position to occupied
                BoardAdmin.KingsideRook.decrementMoveCount(1);
// decrement the move count as the move has been undone
                BoardAdmin.isCastling = false;
// the move is no longer castling

            }
        }
    }
// removes the move object from the list
m_MoveHistory.Remove(lastMove);

// checks that the list is not empty
    if (BoardAdmin.game1.getMoveHistory().Count != 0)
    {
// if the actual move was a castling procedure and not temporary then the rook has its own
move object corresponding to its own movement - separately from the king.
// therefore the top TWO moves in the moveHistory list needs to be removed
// Hence this is recursive as the method is called again for the rook to undo its move as
well as the king.

        if (BoardAdmin.game1.getMoveHistory().Last<Move>().castled == true)
// if the castled property of the move is true then this move needs to be removed as well.
        {

            if (BoardAdmin.TempMoving == false)
// only if the movement is not temporary and is user made, as during temp moves, the move
object for the rook is not added.
            {
                BoardAdmin.isCastling = false;
// sets the is castling boolean to false as the move has been undone.

                this.undoLastMove(); // call method again for rook
// This is the recursive step as the routine will now run again to remove the move object
with the rook inside.
            }
        }
    }
}

```

```
        }
    }
}

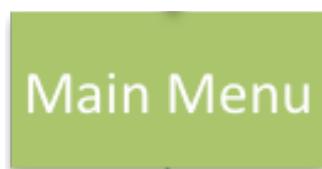
// if the movement is temporary then simply remove the move object added.
m_MoveHistory.Remove(m_MoveHistory.Last<Move>());
}
```

- Variable Lists

For a further reference of the system, a table of variables has been made up to show clearly the variables used, what data type they are and their purpose in the corresponding modules of the system.

The majority of code involves control properties such as “textbox.text” which is not included in these tables. These tables show variables that I have had to create for purposes of the system.

These lists are for the main modules in the system and do not include methods from objects in chess engine. For more information on the object orientation of the system - see the design stage of the documentation.



## Main Menu-Variables

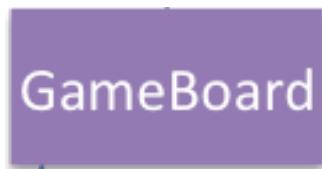
Variable Name	Data Type	Purpose
gameBoardForm	BoardUI (Class)	This is an un-instantiated object which is of the BoardUI class. This object represents the game board module which will be created and opened if appropriate selections have been made.

Variable Name	Data Type	Purpose
game1	Game (Class)	This is an un-instantiated object which is of the Game class. the object is un-instantiated because we don't know which type of game to create yet, this will be determined further down the code structure. This game object will be passed to the boardAdmin form so that it can be used on the game board form after the main menu has been dealt with.
gameStyle	String	This string variable is used to store the style of the game board by a keyword such as "WOOD". This can then be used in the game board form when opened, to determine which squares to populate the board with etc.
filename	String	This variable is used to store the path URL to the question bank file on the server. This is used in processes involving reading/writing from/to the file.
aSW	StreamReader	This variable is a streamReader object which is used to read from the file. The 'readline' method is used to read in each line of the file and populate question objects in memory.
a	string[]	This variable is a string array which is used to store the portions of a line in the text file after it is read using the stream reader object. The file is comma delimited and so using the split function, the parts of the questions are stored in an array temporarily.
player1Colour	Piece.Piececolour	This variable is used to store the users choice of which colour pieces player 1 is playing with. The data type is an enum used in the system.
aAdminLogin	AdminLogin	This variable is an adminLogin object and is created when the user clicks the admin section of the system. This form is then shown using methods of the variable.

**BoardAdmin Class Variable**

Variable Name	Data Type	Purpose
game1	Game	<p>This is a public static variable used to globally store the game object which is being used to currently play a game of chess. This makes the processing easier and removes tedious passing of arguments.</p> <p>This is given to the class by the main menu.</p>
gameStyle	String	<p>This string variable is used to store the style of the game board by a keyword such as "WOOD".</p> <p>This can then be used in the game board form when opened, to determine which squares to populate the board with etc.</p> <p>This is given to the class by the main menu.</p>
player1Colour	Piece.Piececolour	<p>This variable is used to store the users choice of which colour pieces player 1 is playing with.</p> <p>The data type is an enum used in the system.</p> <p>This is given to the class by the main menu.</p>
player1Diff	int	<p>This stores the difficulty level that player 1 is answering maths questions with. This variable is used to determine which questions are presented and how much time should be deducted.</p>
player2Diff	int	<p>This stores the difficulty level that player 2 is answering maths questions with. This variable is used to determine which questions are presented and how much time should be deducted.</p>
promotingPawnType	Piece.pieceType	<p>This variable stores the type of piece that the user has chosen to promote a pawn piece to. This is given to the class by the promotePawn form so the game board can perform the processes necessary to update the pawn piece.</p>

Variable Name	Data Type	Purpose
isCastling	bool	This variable is used to determine whether the move currently being made is a castling procedure or not. This is used by different modules in the system and so is stored in the board admin class.
KingsideRook	Rook	This variable stores the kingside rook piece when the game is performing the castling procedure
TempMoving	bool	This variable lets the other modules in the system know that the current move being made is only temporary, Which will stop certain processes from happening (such as adding the move to history).
questions	List<QuestionBank>	This list variable is used to store all the questions that the system is using for the mathematics mode. This list is populated at the beginning of runtime, to avoid any further communication with the text file during gameplay.
currentQuestion	QuestionBank	This variable is a question object which stores the current question being presented to the user.
player1Score	int	This integer variable stores the score of player 1 - the number of correct answers they have answered.
player2Score	int	This integer variable stores the score of player 2 - the number of correct answers they have answered.
NumOfQuestionsAsked1	int	Stores the total number of questions asked to player 1.
NumOfQuestionsAsked2	int	Stores the total number of questions asked to player 2.



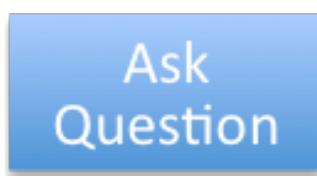
### GameBoard (BoardUI) - Variables

Variable Name	Data Type	Purpose
squares	PictureBox[, ]	This 2-dimensional array stores the 8 x 8 picture box controls on the game board form. This is used in parallel with the 2d array of square objects on the Board class of the system.
gameBoard	Board	The gameBoard variable is a board object and is referenced a lot in the module. The variable gives easy access to the squares array representing the board in the game.
pieceSets;	PieceSet[]	The pieceSets variable is the property of the board class but it is given its own memory space in this form as again, it is referenced a lot.
currentPicBox	PictureBox	This variable stored the current picture box that is being processed - the picture box that has been clicked by the user.
clickedPiece	Piece	This variable stores the current piece that has been clicked by the user. Knowledge of the clicked picture box is used to determine the current clicked piece.
possiblemoves	Square[]	This Square object array stores all the square objects that represent possible moving for the current piece. These squares are valid movements for the piece and are processed to determine other things such as occupied squares and capturing etc.

Variable Name	Data Type	Purpose
player2Colour	Piece.Piececolour	This variable is used to store player 2's colour so that the game class can be updated with this property. Player 1's colour is present from the main menu selection, so player 2's colour can be determined from this.
x	int	This is a variable used to represent an x coordinate of something within multiple procedures in the module.
y	int	This is a variable used to represent an y coordinate of something within multiple procedures in the module.
seconds	double	This variable is used to store the seconds part of the timer value when updating the timer labels on the form.
minutes	double	This variable is used to store the minutes part of the timer value when updating the timer labels on the form.
boardPositions	string[, ]	This 2d array is used when assigning the board squares to the picture boxes on the form and to do this, you need the name of each picture box - which corresponds to the square (A2, B3 etc) and the generateBoardPositions procedure is used to create this 2d string array.
positions	String[, ]	This is used as the return variable in the generateBoardPositions function.
clickedPieceTemp	Piece	This variable is used during click events to store the clicked piece temporarily. This is needed as well as the clicked piece variable because the click event could be an empty square representing a move making click. So the original piece clicked needs to be stored and the second click also needs to be dealt with.

Variable Name	Data Type	Purpose
tempPieceHolder	Piece	This variable is used to store the clicked piece temporarily, while the checkMate procedure carries out evaluations on other pieces and moves. The variable can then reassign the global clicked piece variable after processing is done.
nullCounter	int	This is the first of two null counters used in the checkMate procedure and this counts how many pieces have no possible moves.
a	bool	This boolean flag is also used in the check mate procedure to determine whether the current piece has no moves at all.
nullCounter2	int	This is the second null counter and counts how many pieces have no moves at all - hence if all pieces have no moves then it is check mate.
BlackPieces	PieceSet	This variable is used in multiple routines and is used as an easy access to the black pieces that are being processed.
WhitePieces	PieceSet	This variable is used in multiple routines and is used as an easy access to the white pieces that are being processed.
index	int	This variable is used in procedures to store the index of an object within an array - for instance, store the index of a possible move that has been evaluated to be null. The index can be used to remove it.

Variable Name	Data Type	Purpose
<code>clickedSquare</code>	<code>Square</code>	This square object variable stores the current square that has been clicked by the user in a piece moving click event.
<code>tempText</code>	<code>string</code>	This variable is used to store the text from the move history box temporarily when changes are being made to its contents.
<code>initialPosition</code>	<code>string</code>	This stores the name of the picture box and hence the position of the initial square involved in the move. This is used to add information to the move history text box.
<code>finalPosition</code>	<code>string</code>	This stores the name of the picture box and hence the position of the final square involved in the move. This is used to add information to the move history text box.



Ask  
Question

### Ask Question - Variables

Variable Name	Data Type	Purpose
<code>RND</code>	<code>Random</code>	This random object is used to create a random number which will retrieve a question from the list of questions at random.

Variable Name	Data Type	Purpose
questions2	List<QuestionBank>	This list variable is used to store a secondary list of question objects - those that have a difficulty corresponding to the current player. It is from this list that the question will be randomly selected.
targetDifficulty	int	This variable is used to store the difficulty level of the current player for use in processing of the list of questions and populating the secondary list.
index	int	This index variables is used to store the randomly generated number which can then be used as an index in the questions list to retrieve the question.
seconds	double	This variable is used to store the seconds part of the timer value when updating the timer labels on the form.
minutes	double	This variable is used to store the minutes part of the timer value when updating the timer labels on the form.
difference	double	This variable is used to store the difference between the double seconds and 1. As these are double values they count down from 0.99 and so the difference is needed to subtract that - to make it count from 0.60 like a timer.



Promote  
Pawn

The only variables use in the promote pawn module is the global variable **promotingPawnType** which is stored statically in the **boardAdmin** class.



## Admin Login

### Admin Login - Variables

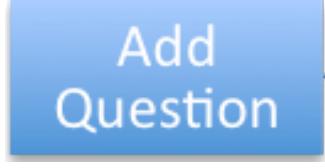
Variable Name	Data Type	Purpose
aAdminPage	AdminPage	This variable is the adminPage object which is opened and shown to the user as a form when the correct username and password are entered.



## Admin Page

### Admin Page - Variables

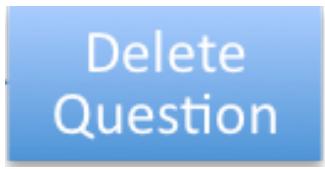
Variable Name	Data Type	Purpose
aViewQuestionsForm	ViewQuestions	This variable is the viewQuestions object which is opened and shown to the user as a form when the view questions button is clicked.
aDeleteQuestionForm	DeleteQuestion	This variable is the deleteQuestion form object which is opened and shown to the user when the delete question button is clicked by the user.
aEditQuestionForm	EditQuestion	This variable is the editQuestion form object which is opened and shown to the user when the edit question button is clicked by the user.
aAddQuestion	AddQuestion	This variable is the addQuestion form object which is opened and shown to the user when the add question button is clicked by the user.



## Add Question

### Add Question - Variables

Variable Name	Data Type	Purpose
focusedTextBox	RichTextBox	This variable stores the rich text box control which has the cursor in currently - i.e the text box that has focus. This is used for when symbols are being appended to the currently focused text box.
isint	bool	This variable is used to store the result of converting the contents of the difficulty text box to an integer value - as it has to be a number. This is part of the validation of the form.
num	int	This stores the result of the number converted when testing whether the difficulty box contains an integer.
newQuestion	QuestionBank	This variable stores the object which represents the new question to be added to the list of questions and the text file. The object is populated by the information the user has entered.



## Delete Question

### Delete Question - Variables

Variable Name	Data Type	Purpose
QToDelete	QuestionBank	This object variable stores the question to be deleted.

Variable Name	Data Type	Purpose
aSW	StreamWriter	A stream write object which is used to write to the text file. This is used to update the text file contents with the updated list of questions - without the question just deleted.
dialogResult	DialogResult	This is used to give a warning about the deletion of a question to the user. This is in the form of a yes/no message box.

## Edit Question

### Edit Question - Variables

Variable Name	Data Type	Purpose
focusedTextBox	RichTextBox	This variable stores the rich text box control which has the cursor in currently - i.e the text box that has focus. This is used for when symbols are being appended to the currently focused text box.
isint	bool	This variable is used to store the result of converting the contents of the difficulty text box to an integer value - as it has to be a number. This is part of the validation of the form.
num	int	This stores the result of the number converted when testing whether the difficulty box contains an integer.
QtoEdit	QuestionBank	This question object is used to store the question to be edited.

Variable Name	Data Type	Purpose
aSW	StreamWriter	A stream write object which is used to write to the text file. This is used to update the text file contents with the updated list of questions - with the question just updated
temptext	string	This is used to store the contents of the text file temporarily so the contents can be trimmed to remove any trailing blank lines and then the text is rewritten back to the file.

**View  
Questions**

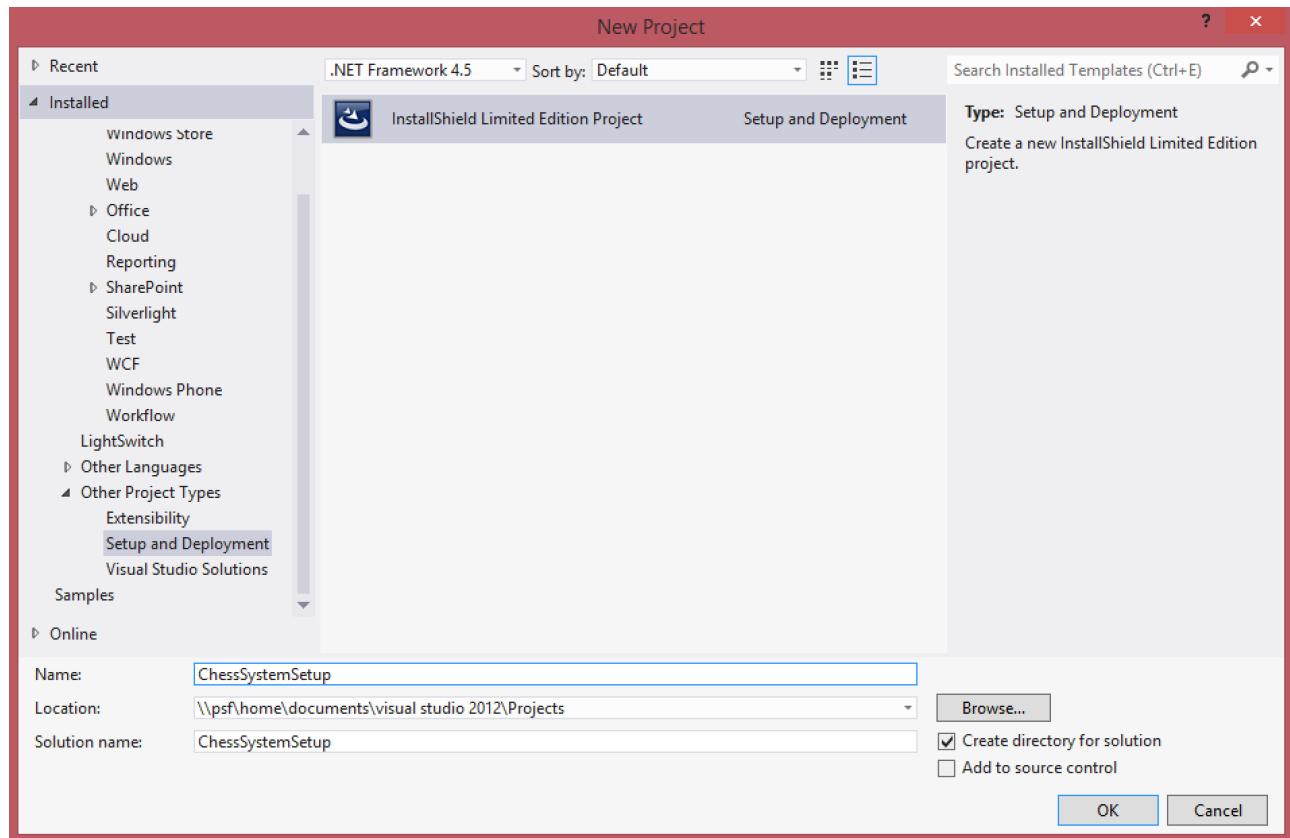
This form does not need any variables as it functions on the properties of the controls on the form. For instance, the text boxes change contents when a different question is selected and the questions are stored in a list in the **boardAdmin** class.

## ● Installation Logs

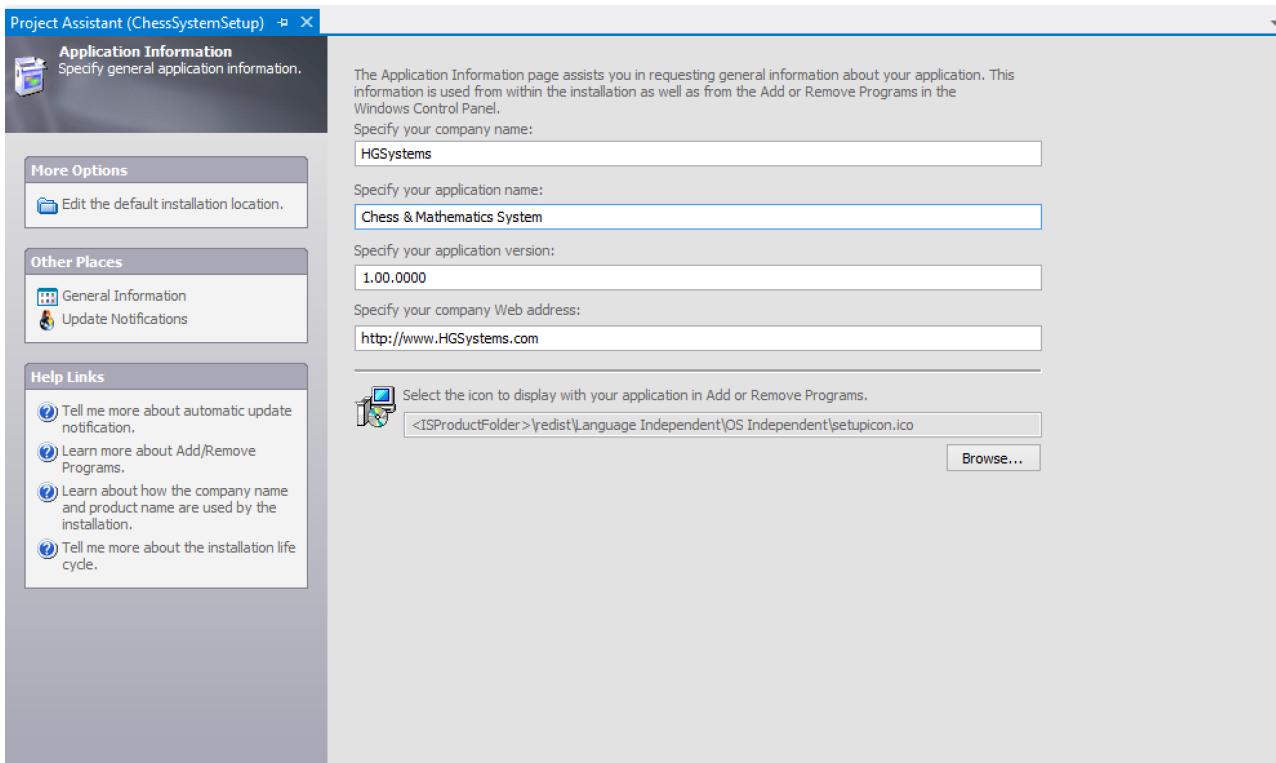
The following shows design views of the installation deployment process. This was how the setup Installation files were made in visual studio and how the executable files are then installed on the target computers.

This installation package would be opened and deployed on every target machine via some server side scripting on the server of the school. This would push out the software to the client machines.

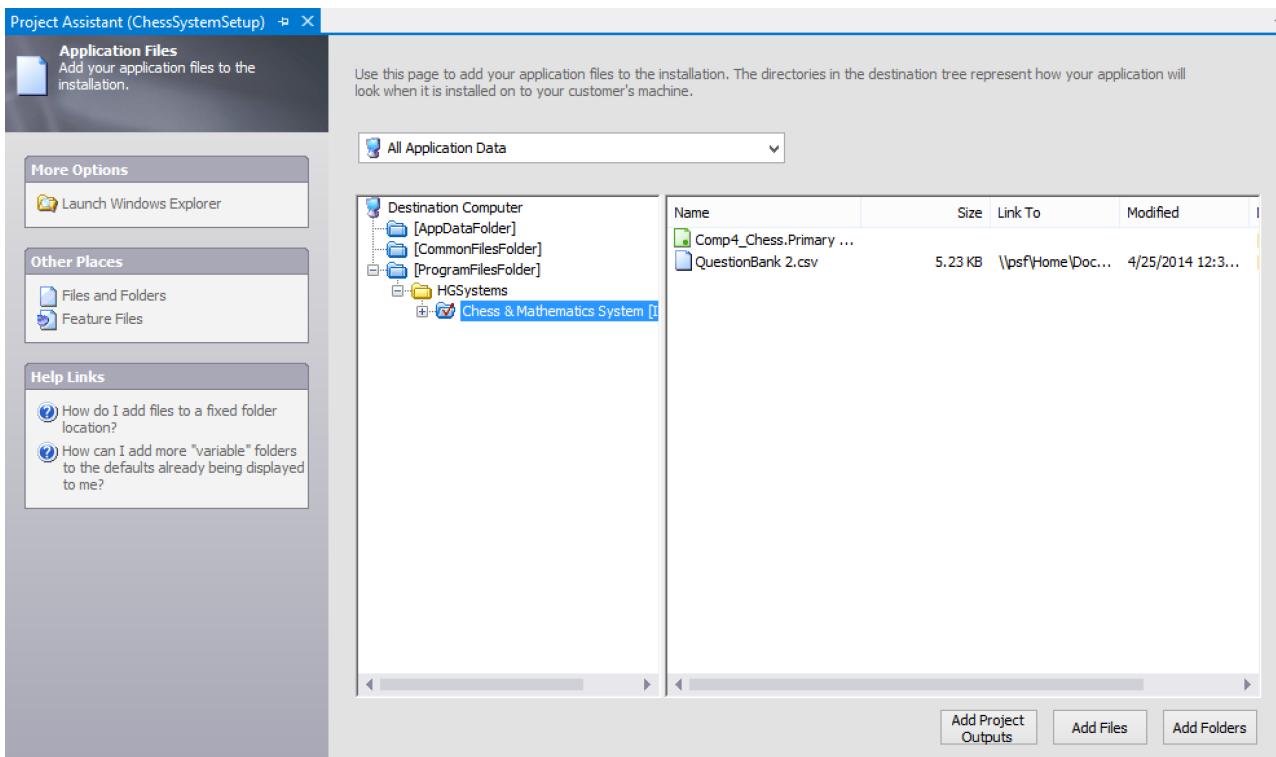
- An installation project was created



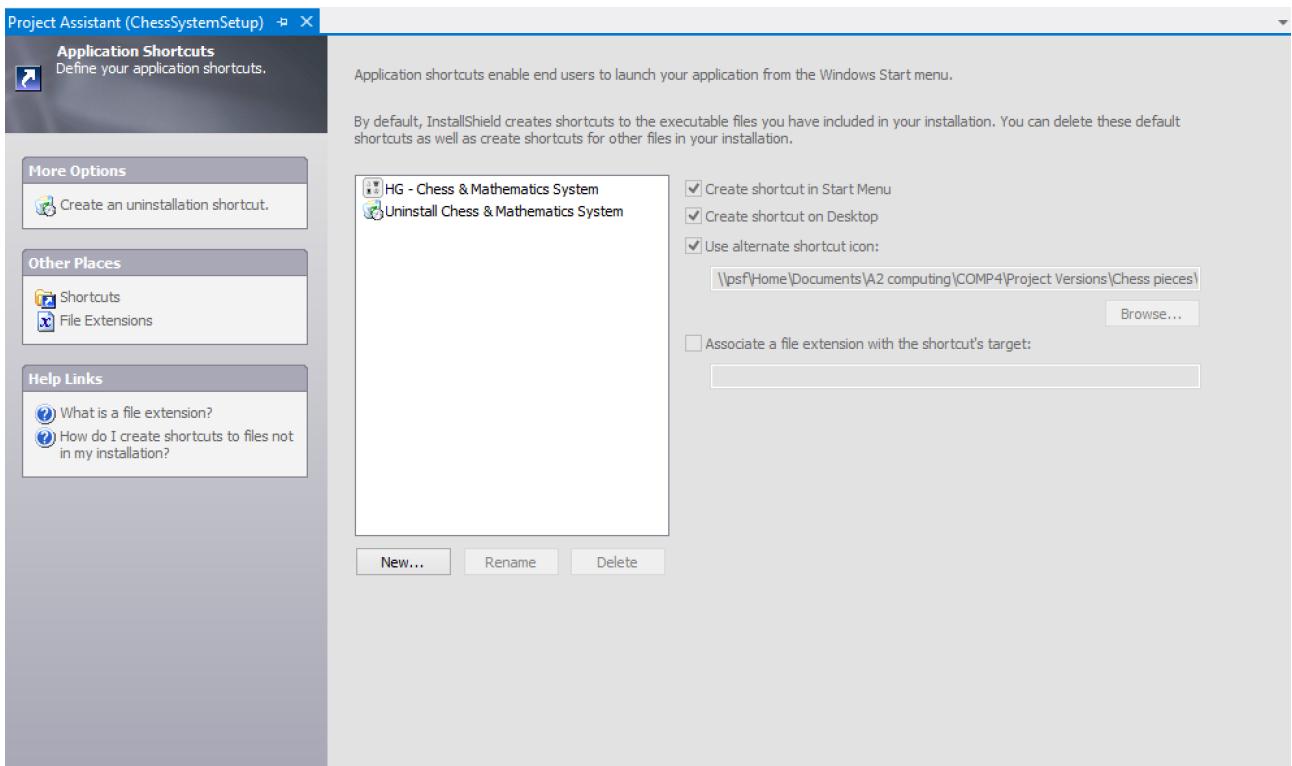
- The name of the application and company was set



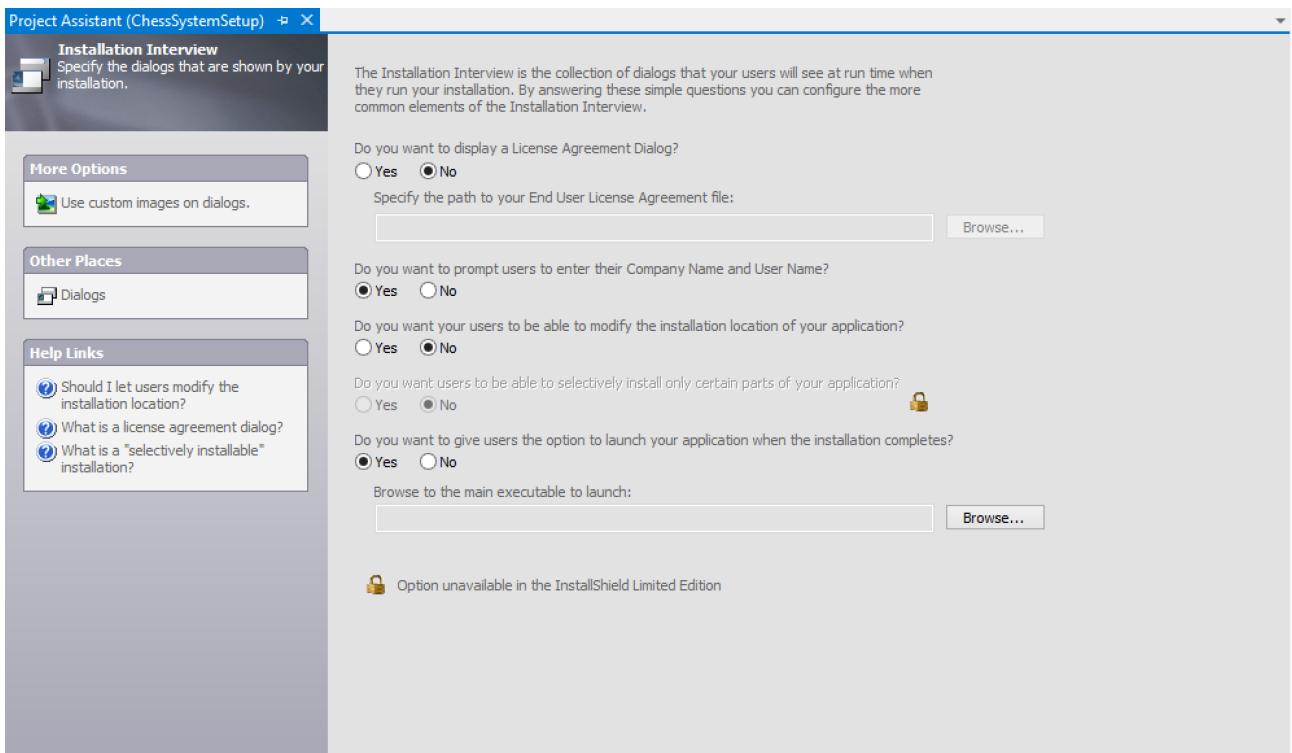
- The folder directory for the system was set and the primary output file was uploaded from the chess project



- The application's executable file and also an uninstall file was given shortcuts here and their icons were also changed to a customised icon.



- Options were set for dialogs and the background of the dialog was set to the same as the main menu in the chess game.



- The setup file is located in the project folder along with the installation logs. this is what will be pushed out to client machines by the server.



- The log files show the solution was successfully built

```
AdminExecuteSequence table successfully built
AdminUISequence table successfully built
AdvtExecuteSequence table successfully built
AdvtUISequence table successfully built
InstallExecuteSequence table successfully built
InstallUISequence table successfully built
Directory table successfully built
Feature table successfully built
FeatureComponents table successfully built
Component table successfully built
Loading File table
Building File table
ISEXP : warning -6245: One or more of the project's components contain .NET properties
that require the .NET Framework. It is recommended that the release include the .NET
Framework.
File table successfully built
Building MsiFileHash table
MsiFileHash table successfully built
Class table successfully built
Extension table successfully built
ActionText table successfully built
AppSearch table successfully built
CCPSearch table successfully built
Condition table successfully built
AppId table successfully built
Started signing 2BFE.tmp ...
Property table successfully built
Binary table successfully built
```

# Appraisal

## ● SMART Objectives Evaluation

The project overall seemed pretty successful as everything works as it should and the system is quite robust and also pleasant to use. But how did the system compare with the original objectives for the project from the analysis stage?

A recap of the objectives set in the analysis stage:

## ● Objectives for the system

### Chess

1. The program should enable a fully functional game of chess to take place between two human players on the computer.
2. In addition to the standard moves of all the chess pieces; support needs to be included for castling.
3. The functionality of the system should cater for the promotion of pawn pieces (a pawn may be promoted when it reaches the other side of the board). This option should clearly be presented to the user in another window.
4. The game must provide two game modes – timed and non-timed. Such modes can be toggled from the game setup section, before the game commences.
5. The game will have an option to change the time variations when using the timed-mode (Increments of 5 minutes up until 60 minutes) that the user can select from.
6. When playing the timed mode, the timers for each player must pause when that player has finished their move. The timer must pause within 2 seconds of that player completing their move.
7. The chess board will highlight squares to show possible move when left clicking on a piece, to aid in the learning of the game. This feature can be enabled or disabled during the in-game menu.
8. There should be appropriate feedback to let the user know who's turn it is, this should be visual and clear so that the turn of the game can be identified without having to click a piece.
9. There should be appropriate feedback of a game result, when a player has been checkmated. A resolution to the game must be made.
10. The Board setup should be dynamic, with Player 1's pieces, whatever colour, at the bottom of the screen (moving upwards).

11. The user interface and chess board must be clear and uncluttered. When the user makes a move, appropriate visual and auditory feedback is required. For example, the user needs to be clear what piece they have selected, where the piece will move to, and the impact of this move.
12. There should be a dynamic list of previous moves shown in unicode notation on the board form, this should be clear what piece moved where in a particular move.
13. There should be a fully functioning “undo” button which allows the user to retract a move that is made for whatever the reason. This button should also therefore delete the move from the list of previous moves shown to the user (objective above).

### Maths

14. The software should have a mathematics mode in which randomised mathematical related questions are presented when a given player makes a move.
15. Each question should be retrieved from a bank of questions and displayed within 2 seconds. These questions should be stored in an easily accessible area which is quick to communicate with (Such as text files).
16. These questions should be retrieved randomly (from the appropriate difficulty of questions) to avoid any sequence to the game and so the games are different every time.
17. Each mathematical question will have an associated test level. Such levels can be selected from the game setup section, before the game commences.
18. The questions asked in the maths mode should be sorted into difficulty which can be selected as a difficulty level before the game starts - the time deducted for a wrong answer should vary with difficulty.
19. A scoring system will keep track of correct and incorrect answers within a game.
20. If a player answers a question wrong, the solution must be shown and time deducted from the current players score. The time deducted should vary with the level of difficulty of the questions. (e.g on level 1 questions, 30 seconds is deducted; on level 2 questions, 20 seconds deducted and on level 3 questions, 10 seconds deducted).
21. The Mathematics question bank should be stored on the school’s server, which should then be able to communicate with the client machines.
22. An interface should be available for an admin user to go in and view, add or remove questions from the question bank stored on the server.
23. Access to the question bank alterations should be password protected to prevent students entering the question bank file.
24. Questions should include full use of unicode characters (such as superscripts [ $x^2$ ] and square root signs  $\sqrt{8}$ ) to make the proposed questions and answers clear.
25. When adding or editing a question in the admin section, appropriate buttons should be available to the user for input of special unicode characters that are not present on the standard keyboard. These buttons should append the symbols correctly to the current text box the user cursor is in.

**Here is a table showing each objective, whether it was met and how it was met:**

Objectives evaluation		
Objective	Objective met?	How was it met? / Description
1	Yes	The system allows a fully functioning game of chess. This was achieved by breaking the game down into logical pieces and tackling each piece one at a time. After building up a structure of object orientation, the chess game could follow using the foundations layer out in code.
2	Yes	The system carries out castling moves correctly. This was achieved by using the methods and properties already used in the other classes to determine whether castling could occur. A canCastle method was created to determine whether castling was possible and another procedure; performCastle was made to carry out the move (for the rook). These were used in combination with the standard system methods to carry out the move correctly.
3	Yes	This objective was met. The promotion of pawn pieces is carried out successfully. The objective was met by using another form to allow the user to make the choice on which piece to promote to and then this choice was stored globally so both modules could use the information. Then using the object properties, an algorithm was made to carry out the necessary changes to the piece object.
4	Yes	At the main menu the user is given choice of which game mode they would like to play. The first selection for game mode is standard game or timed game which is what the objective asked for. From here the user may choose a maths game if they select a timed game.

Objective	Objective met?	How was it met? / Description
5	Yes	Also on the main menu is a numeric option box which allows the user to select the time for the game. This option box increments in 5 and ranges between 10 and 60. This correctly translates to the game.
6	Yes	The timers do pause correctly when a player finishes their move. The time in which this happens is usually around 1 second but sometimes this fluctuates to 2 seconds but this is still as required. This particular objective was dependant on how i designed the algorithms which the system used. In the whole system i tried to ensure that i was not increasing algorithm time complexity where i did not need to (for example not scanning the board more than needed as this was an 8 x 8 process).
7	Yes	This objective was met and the option for turning highlighting on or off was given in the form of two radio buttons in the top right hand corner of the game board form. This object was achieved by using the design of the system and the process for gathering the possible moves for a piece, in conjunction with the images of the squares on the form - highlighting those corresponding to possible moves. The radio button toggle also has a dynamic effect on this highlighting which was achieved through two extra procedures to deal with changing this toggle during gameplay.
8	Yes	This objective was achieved by using a little chess icon that fluctuates between the player labels and this piece corresponds to the colour piece that the player is using as well. This little icon sits next to the label of the player to indicate that it is their turn. This proved to be a simple but effective method of providing this feedback.

Objective	Objective met?	How was it met? / Description
<b>9</b>	Yes	The game can end in two ways: check mate or timer running out. This objective was achieved and the game result is given to the user as a popup message box and is tailored to state why the winning player has won. After this, the game board is disabled so that further manipulation cannot occur.
<b>10</b>	Yes	This was achieved by having a procedure to setup the board and within this algorithm, the positioning of the pieces was determined by what colour player 1 was. This was mainly done by using integer indexes that represent coordinates on the board.
<b>11</b>	Yes/No	The board is most certainly uncluttered and the system in general has quite a low learning curve and it is quite intuitive what to do. The game board form is spacious and it is clear what is what. However there is no auditory feedback to the system due to time constraints in the project so this object was partially met. There is some auditory feedback when the user tries to click off a form that needs to be dealt with - in which case a windows sound occurs and the form flashes.
<b>12</b>	Yes	This was achieved by using a rich text box in the corner of the form and then the design of the system proved helpful here too. With the use of move objects, the previous move can be split up into its properties and these properties were used to add the appropriate information to the move history box. Little chess icons were used also to make it clear which piece was moved / captured.

Objective	Objective met?	How was it met? / Description
13	Yes	This objectives was achieved and there is a fully functioning undo button on the game form. It was fairly straight forward to undo standard moves by using the properties of the move objects to reverse the effects but the special moves such as castling and pawn promotion proved more difficult to undo - so boolean variables were used in the move objects and elsewhere to determine whether the previous move was one of these special moves and the undo algorithm catered for these special moves separately.
14	Yes	With each move of the game a question is presented to the user in a separate form window. This question is randomised. This was achieved by using a random number generator and using the difficulty selection of the player to retrieve a question to display on the form. In the maths mode, in the make move procedure - the system recognises a maths game and invokes the ask question form.
15	Yes	The objective was met but slightly differently than expected. The question is retrieved in under 2 seconds and this was achieved by minimising communication with the text file (stored on a central server) and instead populating question objects in memory at runtime so the system has its own copy of questions ready to go. This made processing much quicker and more efficient.
16	Yes	With each move of the game a question is presented to the user in a separate form window. This question is randomised. This was achieved by using a random number generator and using the difficulty selection of the player to retrieve a question to display on the form. In the maths mode, in the make move procedure - the system recognises a maths game and invokes the ask question form.

Objective	Objective met?	How was it met? / Description
17	Yes	On the main menu there is an option that becomes available when the user selects a maths game. This option is a difficulty level for each player and they select which difficulty level they want to lay on. This rating is 1-3 with 3 being the most difficult.
18	Yes	Each question stored in the text file has a part which denotes the difficulty. When these questions are translated into the system, the question objects have a property which stores the difficulty rating which can then be used as reference when handling the question objects.
19	Yes	A simple scoring system was implemented in the system and is shown in a little section to the side of the board on the form. The system keeps track of how many questions have been asked to each player and how many correct answers have been given. This information is displayed and updated with every move.
20	Yes	The procedure that handles the subtraction of time on the ask question module uses the properties of the question object to determine the difficulty level and if an incorrect answer was given, a time amount proportional was subtracted. This was implemented successfully.
21	Yes	The server is simply a location on the network and so in the system, the path URL of the text file was used to communicate with the file. The contents are put in memory at run time to avoid multiple clients trying to access the same file during gameplay. The admin user can therefore log into any client machine and make changes accordingly.

Objective	Objective met?	How was it met? / Description
22	Yes	This process works correctly and the objective was met successfully. The admin user can log into ANY client machine and access the admin section. This communicates with the file stored on the server and the different areas of the admin section let the user perform different tasks on the file.
23	Yes	This objective was achieved by having an admin login page which separates the admin section and the main menu of the system. This admin login page requires a username and password which is given in the user manual.
24	Yes	Using unicode encoding in the text file an the project, special characters were implemented successfully. Using rich text boxes in the forms aided this.
25	Yes	The forms have the correct pane of buttons available to the user which allows them to enter unicode characters. The objective was achieved by having a global text box variable which stored the textbox that currently has focus and then when the buttons are clicked, append the symbol to the current text box.

## ● Questionnaires and User Feedback

NAME: Aaron Bailean

Signed: 

### User Appraisal Questionnaire (Students Version)

1. Do you think the program solves all of the problems that the chess club had previously?

Yes, it saves so much time!

2. What feature(s) of the program do you particularly like?

The ability to change board design and the animation effects when the board loads.

3. Are there any parts of the program that you do not like?

No.

4. How easy do you find the system to use? (circle your answer)

Easy

Takes a little time to get used to

Difficult to use

5. Do you find the mathematics mode a helpful way to revise?

Yes, it added a fun twist to the revision.

6. Have you had any problems using the system?

No.

7. What features would you like to add to the program if you could?

Extend system to cater for A2 maths as well.

8. On a scale of 1(lowest) – 10 (highest) what would you rate the program?

8

NAME: Rayit Bayi

Signed: 

**User Appraisal Questionnaire (Students Version)**

1. Do you think the program solves all of the problems that the chess club had previously?

We get to play straight away now and all of us get to play

2. What feature(s) of the program do you particularly like?

Undo button

3. Are there any parts of the program that you do not like?

That you can't play against computer

4. How easy do you find the system to use? (circle your answer)

Easy

Takes a little time to get used to

Difficult to use

5. Do you find the mathematics mode a helpful way to revise?

Yes it's fun

6. Have you had any problems using the system?

No

7. What features would you like to add to the program if you could?

To play against other people or other laptops

8. On a scale of 1(lowest) - 10 (highest) what would you rate the program?

8

NAME: JAI Beila

Signed: J. Beila

**User Appraisal Questionnaire (Students Version)**

1. Do you think the program solves all of the problems that the chess club had previously?

- Too long setting up boards  
- With the game we could play straight away

2. What feature(s) of the program do you particularly like?

- The Move history box so you could track your moves

3. Are there any parts of the program that you do not like?

- No

4. How easy do you find the system to use? (circle your answer)

**Easy**

Takes a little time to get used to

Difficult to use

5. Do you find the mathematics mode a helpful way to revise?

- Yes because the short questions are good for revision

6. Have you had any problems using the system?

- No

7. What features would you like to add to the program if you could?

- Extend System to ~~AS~~ A2 Maths

8. On a scale of 1(lowest) – 10 (highest) what would you rate the program?

9

NAME: HARMEET RAI

Signed:           **User Appraisal Questionnaire (Students Version)**

1. Do you think the program solves all of the problems that the chess club had previously?  
- Yes, it saves time because it takes time to set up the boards also we have problems finding some of the pieces.

2. What feature(s) of the program do you particularly like?  
- Having highlighted area for where each piece can go.

3. Are there any parts of the program that you do not like?  
- On Maths mode when time is deducted there should be an indication that time has been taken off.

4. How easy do you find the system to use? (circle your answer)

 Easy**Takes a little time to get used to****Difficult to use**

5. Do you find the mathematics mode a helpful way to revise?

Yes.

6. Have you had any problems using the system?

- Main menu was a little confusing.

7. What features would you like to add to the program if you could?

1 - None.

8. On a scale of 1(lowest) - 10 (highest) what would you rate the program?

- 8

**User Questionnaire typed up by Mr C Coetzee, which was emailed to him for feedback:****User Appraisal Questionnaire (Mr. C Coetzee – Chess club organiser)****1. Do you think the program solves all of the problems that the chess club had previously?**

Absolutely. Running the club with missing chess pieces and loads of equipment issues is nightmarish. Your proposed and demonstrated solution makes the entire 'chess experience' digital and allows for much more efficient use of our time. Club members can start playing almost immediately after starting – no more finding missing chess pieces, setting up, clearing up, moving tables. It saves us at least 15 minutes on every session.

The inclusion of maths problems into chess is a welcome addition. When I demonstrated it to our head of maths, she was delighted. As you know, there is a strong link between maths and chess and studies have shown that regular chess playing increases students' maths scores.

**2. What feature(s) of the program do you particularly like?**

The interface is very intuitive and gives the user (mostly students) clear visual cues and feedback on what to do next. The ability to allow for maths questions with more than one answer (for example questions with parabolas) is a real bonus. The ability to 'undo' moves is great – especially being able have a written record to share the play with other club members is great.

The over-all look and feel of the program is very professional and is exactly what I would have expected from a commercial system.

**3. Are there any parts of the program that you do not like?**

Nothing springs to mind. If I have to be very nit-picky, I would comment on the position of the radio button on the selection screen being a bit awkward. But it is a minor annoyance and is more of an aesthetic issue than a functional problem.

**4. Are there any areas of the system that you think could be improved?**

I think the program would benefit from a splash screen when opening it. You need to advertise your expertise – it will surely bring in the good stead for future projects. If you could also allow for the maths questions to be taken from an online source – to minimize typing mistakes, it would be great.

**5. Is there anything that you might like to see added to the system, given more time?**

It would be nice at some point in the future to be able to play the program on tablets/mobile devices. I would think the interface could easily be adapted to allow for touch-access. Lastly, it would be great if players could play matches across a LAN/WAN, i.e. giving each player his/her own dedicated computer to play from.

**6. On a scale of 1(lowest) – 10 (highest) what would you rate your new program?**

Definitely a 10 – without a shadow of a doubt. It does exactly what we had hoped it would do and then some more. Thanks so much!

## Here is a letter I received from Mr Coetzee detailing feedback on the system:



Meadow Road, Gravesend, Kent DA11 7LS  
Telephone: 01474 533082 • Fax: 01474 533844  
Confidential Fax: 01474 561536

Website: [www.saintgeorgescofe.kent.sch.uk](http://www.saintgeorgescofe.kent.sch.uk)  
Email: [offadmin@saintgeorgescofe.kent.sch.uk](mailto:offadmin@saintgeorgescofe.kent.sch.uk)

Headteacher: Mrs A Southgate ED AKC

25 April 2014

Dear Harry

**Re: Chess/Math program for school Chess Club**

Herewith our most sincere thanks for creating such a lovely bespoke computer program for our Chess Club.

We were blown away by all the features you demonstrated at your demonstration to staff on Friday, 18 April. The members of the maths department were particularly pleased in the addition of a 'maths-component' to the program. As you mentioned, there is a clear link between playing chess and maths achievement and we are keen for any of our groups to achieve even better than they do already.

Of particular interest is the program's ability to restrict possible moves to actual legal moves, based on the state of play of the game. It will allow new students to pick up the game even sooner than our traditional method of playing induction games.

The program is professional the user manual extensive and well written. We've already had two requests from other schools in our chess league to see if they could install it there. We've taken the liberty to give them your contact details.

Thanks again for the creating this program for us.  
We wish you every success in your future Computing endeavors.

Kind regards

Mr. Chris Coetzee  
Chess Club organiser  
e-mail: [coetzeec@saintgeorgescofe.kent.sch.uk](mailto:coetzeec@saintgeorgescofe.kent.sch.uk)



Living the Olympic  
and Paralympic Values



INVESTORS IN PEOPLE

BRITISH COUNCIL  
International School Award



ARTS COUNCIL  
ENGLAND



THE QUEEN'S  
TEACHING  
INSTITUTE  
2013  
ENGLISH  
HISTORY



The questionnaires were made and given to students that attend the chess club after they used the system a few times. This was a concise and clear way to receive feedback from the users of the system. The questions included good and bad parts of the system, whether they feel the system is a good replacement from the current system and general ratings on the system.

The next feedback was from the main end user of the system, Mr C Coetzee - the chess club organiser. I emailed him a similar questionnaire but tailored questions for him and he replied with the answers typed up.

To Introduce the system, i held a short demonstration to some staff members from both the mathematics department and also the ICT department, showing them how the system works and its intentions - putting emphasis on how the maths side of the system is key for revision and effective consolidation of content.

I shortly after received the letter from Mr Coetzee with general feedback for the system.

## ● Analysis and Evaluation of User Feedback

The questionnaires were a great way to receive concise feedback from some students and their replies were extremely useful. A common thing that arose was the possibility of extending the mathematics system to A2 content which was great to hear because this means the students get along with the maths side of the system, so much so that they want to broaden the system with other content. This was certainly an understandable improvement pointer from the students.

The problems That the chess club had certainly seem to have been overcome which is great to hear because after all, that is what the system was made for. However, it is equally rewarding to hear of "extras" that some members are proposing such as the possibility for networked play.

The undo button and the highlighting of moves were key features that the students liked and also the ability to change the look of the board was also a favoured aspect.

The feedback from Mr Coetzee himself was extremely rewarding and overall the club are very happy with the system which is great to hear. The letter i received also was good feedback as this was after demonstrating the system to members of staff and confirmed that the clients like the new system.

I was extremely shocked to hear other schools wanted the system on their computers, this must mean that the new system is favoured by not just our school staff but others too. If they get into contact i would quite happily hand over the system to them also, but may have to give them guidance on making the system work with their server.

I would definitely make any small changes, however, that were pointed out from feedback and then reinstall the system here and then at other schools potentially too.

I think after feedback i realise the main menu could be a little more clear - perhaps by hiding the unavailable options all together, as it is not obvious straight away. Also more confirmation of check and checkmate would definitely be an improvement and i can understand, now, why this has arisen in user feedback.

Having mainly positive feedback is reassuring because when designing the system, a main intention was to make it easy to use and clarity was a key goal. This, judging by the feedback received seems to have been met successfully - with the odd few exceptions in the system.

Also having feedback about extending the maths system is a good indicator that students like the quick fire nature of the questions - which i anticipated. I stressed this feature in my demonstration and having feedback from the students themselves only reassured this.

Overall i am extremely happy with the feedback received and i understand the things that have been pointed out from the system as things that could be improved. Some extra features suggested could have been included given that time were not a constraint.

## ● System Extensions

If time was not a constraint in the project, there could be a number of things i could add to the system and extend the functionality based on both the user feedback given and also the design of the system itself.

### Artificial Intelligence engine - Playing against the computer.

This is something that was a limitation, as discovered in the analysis stage of the project. This was mainly because this has been developed for years in computer chess and the algorithms used to carry out this computer playing chess system are extremely advanced. However given more time, it is something i could potentially implement.

#### How would this be implemented into the system?

Based on the way i have designed the current chess system, there are numerous ways i could go about an AI chess engine.

The first thoughts would be to create a **computer** class - much like a player class; a computer will act as a player in the game. The only difference here will be, which move should the computer make?

This is where a rather significant algorithm will come into play. The approach would be for the computer to assume a “perfect” game and make moves based on a scale rating. This happens a certain number of times into a future state of the game and the computer makes the move that is most highly rated.

So the computer would make each possible move temporarily and then rate that move based on its impact. Once this particular move is made, the enemy pieces should be scanned to see whether the piece can now be captured and also see the status of check in the game. These properties will be used to determine a rating for that move. This means a potentially recursive algorithm as a process of determining the effects of a move could be infinite, with potential moves subsequently having a range of effects.

This would be modelling using a tree data structure. For every move of a white piece, there will be 20 possible black moves etc and so on and the computer will scan through this tree to determine the best move. When a move is finally made, the following turn this whole process will start again.

To implement this in my system i could make use of the way i have designed the chess engine and as stated, make a computer class. This computer class would then make a move and run a recursive algorithm to make the best move.

This could then be implemented as a difficulty rating of the computer player. The level of intelligence, so to speak. The more intelligent the computer player then the further they would scan into the game to determine the best move - so a cap would be used for discrete difficulties. For instance, a computer player, playing on level 1 difficulty would only scan 5 moves into the future of the game and determine a move from that. A more intelligent computer player would scan 15 moves, say, to determine a move to make. This would mean a more developed idea of which move would be best to make.

This algorithm to determine the best move could make use of my **makeTempMove** procedure and also my **isInCheck** and **isCheckMated** proceeders most certainly as these are key factors of move-making.

However, Algorithm complexity will be incredibly important here as the processing could be very slow if designed poorly.

## Networking game mode - playing across two computers.

This was an interesting prospect and given more time, would certainly be a possibility. This extension would mean adapting the system to run the application simultaneously across two networked clients. Which would be quite straightforward given the environment the software is currently in, is a local school network in which all machines are networked. This would extend the use of the school's server to not just store the file but use as a "middleman" between the two computers playing a game of chess. This approach would require **socket programming** and involve the use of IP addresses and TCP protocols.

### How would this be implemented into the system?

The first thing would be a new option in the main menu for a networked game. The system would then carry out a procedure to connect the two networked machines and have them handshake to establish a connection. The game would then have to run on both machines and the most likely idea would be to assign the machines as the players of the game and implement it so one machine cannot do anything while the other machine is making a move etc otherwise this would cause communication problems.

The server could be used with the possibility of some server side scripts to handle some of the requests from the machines.

Once the game runs across both computers. Appropriate validation would be needed in case there is a network error. Should this happen the machines should re-establish a connection and possibly undergo a recovery procedure to return the game to its previous state.

## Improvements to original system

There could be some good improvements to the original system swell as further extensions with the bonus of extra time. These improvements mainly included, an extended scoring system and the saving of games.

The extended scoring system could be in the form of a global high-score table which records games played and the information regarding that game. This could be useful for when the chess club is hosting tournaments and the scores of the tournament games want to be recorded. It therefore leads to printing, using the data recorded to print a sheet with the information about the games played. This would be a useful feature for reflection on a bundle of games played within the club.

As well as the scoring system, the ability to save a game and return to it later would also be useful. This wasn't included in the original system as it was not a somewhat "urgent" feature as the chess club runs in a certain period of time and the whole point of timed games is to play under time constraints. However having the ability to store the state of the game would be useful.

This would be implemented by storing the positions of each piece in the game and other important properties such as time values and maths scores, as well as the key information regarding the

board and the players (colour and style etc). This information could be recorded in another text file but then this would get large due to the number of clients. Instead, chess club members could have a login in which they can login and select from their own individual previous games to restore from.

Improvements on user interaction with the system, i discovered from user feedback that more confirmation of being in check and checkmate is needed, The message boxes are a little too subtle. This makes sense from a players point of view and would definitely be easy to implement. An idea would be a label to the side of the board which clearly shows in large text whether the game state is check or check mate or possibly a flash of text to make the action more dramatic.

One final thing would be the ability to save a game and load it back when the users come back. This would be implemented by using another text file, besides the question bank, but to store the state of the game and the information determining the state of the game.

## ● Reflection

I certainly had fun doing the project, particularly the designing stage and looking at potential routes for the technical solution and trying to build upon a structure for the system. I particularly enjoyed the heavily object orientated nature of the system as this was logical and the breaking down of the real world problem in this manor was extremely rewarding. I definitely feel my skills as a programmer have improved as a result of doing this project. However, looking back through the technical solution i think further refinements could have been made. I initially refined some code as i went along the technical solution which improved my logical reasoning when programming - thinking about how algorithms can be made more efficient, but some parts of the solution could have been refined further still. In the coded solution there is a lot of comparisons for whether it is black or white's turn and also whether it is player one or player two's turn. Because these things are separate entities, it meant i had to repeat code in these situations which was wasteful (This did not decrease efficiency however, because only one section would be run at any one execution). This could be refined to cater for both selections and reduce the amount of code needed.

Areas of difficulty included coming up with an object structure for the system, handling the text file contents and implementing special moves in the game. It took a while to get a design down on paper for the object structure of the system but as shown in the design, this ended up being refined further once the system had been thought about more. These refinements made the components of the system link together better and slowly merge into one big system. Not knowing where to start was daunting but when i sat down and started to break the problem down logically (splitting a game of chess into smaller pieces - which in turn, became objects in the system) it started to become much clearer in my head - theoretically anyway. Text file handling was quite tricky at first due to not knowing the best way to import and export data from the system but after designing the layout of the maths system and then thinking about the involvement with the server for security and efficiency purposes, this became quite a pleasant task. The special moves in the game sat outside the boundaries of the system i had built up and so these were tricky to implement, but using the way i had designed the object orientation and making use of other methods i created, these moves eventually found their way to the board - and quite effectively too. A particular struggle was undoing the castling procedure but the solution eventually came with a recursive routine. Luckily i had refined my model to using Move classes which are listed as the games history, the undo procedure simply removes the last move object and uses it to undo the effects of the move. So for castling, two move objects were added and so the undo procedure should be called twice - or recursively. This was satisfying when the solution was successful as it again, made use of the way i had designed the structure of the system.

If i had more time i would love to implement the suggestion from user feedback.

# Appendix

## ● User Manual(s)

The user manual is attached at the back of this Project. One of which is a standard manual for the system and another is a brief reference to rules of chess for beginner users.