1. **Introduction**
   The program is a regular two-player chess game which has 4 different levels of artificial intelligence. The levels of AI levels 1 to 3 are described as in the assignment guidelines, while the AI of level 4 will pick out the best move by looking 2 moves ahead, and assigning a value to each move. We have a cell class with has subclasses for each piece as well as the empty and NULL cell classes. Each of these subclasses determine where the piece in question is supposed to go. Our chessBoard class aggregates the cell classes, gets all the possible moves from the individual cells, where the possible moves are subject to further conditions to determine whether they are actually valid. We have a strategy overclass to implement our players, which include human and computer. We also have text and graphics display classes for the displays.

2. **Overview(describe the overall structure of your project)**
   Design Pattern: Decorator Pattern and Strategy Pattern
   Number of classes: 19
   Super Class: Cell, Strategy
   Owns A: gameBoard owns a GraphicsDisplay and a TextDisplay
   All subclasses of Strategy owns a gameBoard
   Inheritance: NULLCell, Empty, Pawn, Knight, Bishop, Rook, Queen and King are subclasses of Cell class.
   Level1, Level2, Level3, Level4, Player are inheritance of Strategy class.

3. **Updated UML**
   On due date 1 UML, we planned to use observer pattern to find out all possible moves for each pieces. Hence we used observer and subject classes to implement the observer pattern. However, after discussing we realized that it is not necessary to use observer pattern to find out possible moves for all pieces. This is because the one to many relationship used in the chessboard is a finite, unchanging amount. For each piece, the algorithm used to find possible moves does not require the abilities of the observer pattern. In fact, it would bring no modularity and instead only bog down the program. Additionally, on due date 1 UML, there is no strategy classes(strategy class and its subclasses). We find out strategy pattern online and it is the best way to show four different levels of artificial intelligence. As a result, we decided to use it and add it to our new UML.

4. **Design(describe the specific techniques you used to solve the various design challenges in the project)**

Decorator Pattern: In the game board, we consider that the board has 64(8*8) cells. Each cell can be either empty cell or piece(pawn, bishop, knight, rook, queen and king). In order to help the algorithm of finding out possible moves of a knight work, we planned to add 2 more columns or rows for each side. In order to make difference between empty cells and extra cells, we create a new subclass NullCell to represent all extra cells. In this way, we are able to distinguish the pieces and find out all possible moves. If a cell would potentially move to a null cell, we know that this must be wrong as the null cell represents a space off our board. We use a border of 2 to ensure the knight will not jump off the board. This allows for easy checks for whether a piece has moved off the board. Also, we make a copy of the cells information before any move made so that all we need to do to revert a specific move is to delete the cells at the original locations and load the copied cells in.

Strategy Pattern: In this game, we need to generate at least 3 levels of artificial intelligence. At first we tried to make 4 methods(level1, level2, level3, level4) and one method for human player's input in gameBoard class. However, it is too complicated to store all algorithms for various levels in the chessBoard class. And it is too hard to add new levels of artificial intelligence. As a result, we searched online and ask lecturers for advice. We finally found strategy pattern is the most appropriate way to solve our problems. It makes each level and human player to be different subclasses of Strategy class.

Vector: In our project, we used a lot of vectors to store move history and all possible move. We can use push_back method to add new moves to move history or add new possible moves to possibleMove.

5. **Resilience to Change (describe how you design supports the possibility of various changes to the program specification)**
Our program is very resilient to change. One of the major aspects of our program is how we defined each one of our pieces can move. Essentially, we provide our individual pieces a vector of directions that symbolize the directions in which the specific piece can move. From here, it is possible to not only have pawns that move vertically, but also pawns which can move horizontally which will be needed when 4-handed chess is implemented. If the rules for piece movements also change, we can easily accommodate that by changing our directional vectors which determine where the pieces can move. For move advanced changes including the entire change of rules, all we have to do is change the actual movement function in our chessboard.

Another way we have a resilient to change program is that our existence of NULL cells allows us to be able to make very unorthodox boards, such as a circular chess board. In this case, instead of having to create complicated if statements, all we have to do is create the board in the unorthodox manner, filling in the extra spaces with null cells as well as adding in a border of 2 null cells, so that our pieces can automatically detect moving off the strange board by checking if the location it wants to move to contains a null cell.

Our program also easily supports a change in input format. This is because our input already has a conversion function which converts standard notation moves into the coordinate system used internally. The only changes that are needed to accomodate a new coordinate system is to create a seperate function which deals with the new format of instructions. If we anticipate a significant number of such changes in input format, we can simply create a input strategy class which will change depending on which input method is selected.

Another aspect of our program that allows for change is the fact that our implementation for describing a chess move is using a coordinate system with pairs of integers. It is extremely easy to change this structure to a template instead, where coordinates are described with vectors of integers. Note that our operations in the program can simply be replaced by a function with iterates over the coordinate vectors. The ability to accommodate this change is what allows our program to support higher dimension chess, including 3-D chess played on a cube structure. To accomplish the above tasks, we require very little changes to the original programs.

One aspect of our program which allows for extremely easy implementation of higher levels of AI is the fact that we have a way to undo and redo moves, as well as the fact that we implemented a function which returns the valid moves. The existence of such functions allows the client to construct pruned trees of potential moves and the changes to the pieces by repeated performing valid moves, figuring out whether subsequent moves are good, and selecting the best move from the pruned tree. Additionally, we can also attach a standard book of openings to any subclass of our strategy function, which allows for a very quick early game, and reduces computation time for the early game, where the times would be the longest. In this case, we don’t require any changes to the original program to make this change. Our strategy class only has the responsibility of returning the next move, which is why we can easily create a subclass to work with different kinds of AI, including stockfish.

To minimize compilation time, we used the PIMPL idiom with pointers to the object implementations instead of simply containing the object. This way, changing the private members of an object won’t trigger a recompilation from a class which holds the object, for example our chessboard and strategy classes. Additionally, forward declarations are also used to cut down compile time.

Cohesion and Coupling
Our program is also very cohesive. The cell class only provides the locations to where the cells can move, the strategy subclasses only have provide the next valid move that can be made, and the chessBoard class only concerns on enforcing correct moves and providing information for other classes to make decisions. The chessBoard uses the textdisplay and graphicsdisplay classes to print out the board, instead of printing it out itself and lowering cohesion. It notifies these 2 classes to update their information.

6. Answers to Questions

1)Question: Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves and responses to opponents’ moves, for the first dozen or so moves of the game. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

We will implement a “book of openings” with a trie. The edge of each trie will be a recommended move supported by the book of openings. So the root will have a certain number of edges that are the recommended moves by the book of openings, and each will lead to a child that represents the board state after that initial move. The next layer of moves in the trie will belong to the opposite colour, and so on. This way, all a program has to do to display the recommended moves for a particular chess board is to search the trie for the state, and return the children of the node which was found, or nothing if the state could not be found. We won’t be able to return any subsequent moves if the game state has progressed too far or if it has deviated wildly, but that is not the responsibility of a book of openings. We will initialize the trie when the program is first executed, and store the trie in the computers hard drive so all future execution of this program can skip the initialization. This trie will be associated with the ChessBoard data structure, and the ChessBoard data structure will provide a list of good moves to make to the display, which will then display the moves. We can augment the trie by adding the names of what type of opening you are playing towards to each node, so the display can also tell the player what moves to make for a particular strategy, for example Scandinavian Defense. No change.

2)Question: How would you implement a feature that would allow a player to undo his/her last move? What about an unlimited number of undos?

Answer: We will implement a feature that allows a player to undo an arbitrary number of their moves by storing all moves performed in a stack associated with the ChessBoard data structure. When any player performs a move, the information for the coordinate of the move as well as the piece that was captured (if any) will be pushed on a stack. The player can choose to issue a undo i command, which will undo the last i commands, or all commands if there are less than i moves made. When the ChessBoard data structure receives this command, it will pop off moves from the stack one at a time, and for each move popped of it will perform the reverse of that move. After i iterations, the last i moves will have been reverted. This also doubles for a history of the moves made. We implemented this.

3)Question: Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game.

Answer: Change the ChessBoard class into the chess board of four-handed chess. That is, we will use an augmented coordinate system to be able to display four sides. We will also increase the number of colours to 4 and treat opposing colours identical we calculating capture, valid moves and checkmate. We will also have to treat allied chess pieces as those that we cannot capture, but also cannot move. We will also have to change the behavior of pawns depending on their colour, as they can only move away from their home base. Everything else will remain the same. Changing calculating capture, valid moves, and checkmate doesn’t need much work at all, we just have to augment the definitions of allies and enemies. We will also add modes for 4 player chess, including a free for all as well as modes for teams. In a free for all there is no need for implementing allies, but for teams we will have to use the ally functionality. We will also manually remove all pieces of a checkmated player from the game. We change the function of checkmate() that both players in the team is checkmated will lose the game. Otherwise, if this is a free-for-all, there is a winner until only one colour remains. We also change the functionality of drawing such that if any 2 players draw then the entire game has been drawn. Identical answer. Also covered above.

7. **Extra Credit Features(what you did, why they were challenging, how you solved them)**
   In our project, we can print out the move history because there is an attribute named moveHistory in chessBoard which is a vector of chessMove(a struct to should move type, start position, end position). So we are able to implement a method in chessBoard to print out the move history. It will read the vector in reverse order, and print out the moves line by line.

   Similarly, we have a method named undoMove() to undo your steps because in move validation checking process, when we try to figure out after this move whether it will cause a checkmate for opponent. If it is not valid, we will undo that step. We have extended this method for the player to use, and so a player only has to call undo if we let undo be a command in main.cc.

   We also have a method called possibleMoves which prints out all the possible moves a user can make. It simply gets our vector of possible moves and prints them out one by one.

8. **Final Question**
   1)What lesson did this project teach you about developing software in teams?
   For this project, I have learned that it is extremely important to have a appropriate design pattern for the software. With an inappropriate design pattern, it will take you a lot of time to implement your idea. Also, UML is necessary before you start your code. It is always easier for a team to start separately with a UML. UML will be a reference for team member who's part may need methods from other team member's part. They can work at the same time without waiting for others' code.

   Additionally, writing clean codes is also very important for a large project. Everyone's code may have bugs need to be fixed. If your code is not readable, it is impossible to have peer review and debug. What should be done next time is to have extremely detailed comments with all code that is written in order to promote understanding. Although our code was self-explanatory, we definitely would benefit from very detailed comments.

2)What would you have done differently if you had the chance to start over?

If we have a chance to start over, we will start to consider our design pattern earlier so that we do not need to redo our project in the second week. Also, we should follow the schedule so that we can have plenty of time to debug. What happened was that the observer class did not meet or requirements, and implementing it would not take advantage of what it offers. As such, should have carefully considered what we actually needed, instead of choosing the first design pattern that sort of makes sense.