Final Report – CS 246

# Design:

See the attached UML for a class overview of the functions and details each class includes.

The final design plan has not changed significantly from what it was in due date one. To briefly outline the design of the game, essentially there are four fundamental components, which are: *Board*, *Piece*, *Player* and *View*. The *Piece*, *View*, and *Player* classes are abstract classes and specialized classes are derived from each one of them. Whereas, *Board* is a concrete class on its own and has an “owns a” relationship with *Piece*. To summarize the flow of the game, *main()* is where *Board, Player* and *View* are created, *Board* then proceeds creating appropriate *Piece*s, user inputs are then taken by *main()* and lastly appropriate methods are called to create a fully functional game.

# Differences in Final Design

As previously mentioned, the classes that were planned in due date one did not go through a lot of changes. The only major change is making the relationship between *Board* and *Piece* from a “has a” to an “owns a” relationship, this change was made to portray a more realistic relationship that occurred in the code between the two classes. Furthermore, throughout the development of this game additional methods were required to improve the functionality of the program and hence all the necessary methods were later added to the final design.

# Overview of the Project

**Board Class**

Relationship:

Composition: *Board* owns *Piece*

Class Overview:

The constructor of this class creates the 2-D array of *Piece* pointers and assigns null to all of them and false all to its private *bool* variables, which are essentially their default values. It contains several methods that are required to fulfill all its responsibilities, the most important ones are:

* *bool isCheck(), bool isCheckMate() and bool isStaleMate()*
* Methods help to determine if the state of the board is in a check, stalemate or checkmate
* *int move(int, int, int, int, char)*:
* Method helps to check if each move that the player inputs, which is called by *Player::move()*, are valid. The validity of the move is determined by calling the *Piece::move(…)* method and *isCheck()* method
* *void editBoard(string, int, int)*:
* Method helps in adding pieces for the setup process, a helper method *bool isDone(…)* is called to validate the board layout when the setup is done

The destructor frees all the allocated heap space that was occupied by 2-D array.

**Piece Class**

Relationship:

(none)

Class Overview:

The *Piece* class is an abstract class. It is used to derive six other classes which are *King*, *Queen*, *Bishop*, *Rook*, *Knight* and *Pawn*. Furthermore, the constructor takes in a coordinate, two *int*s, a *char* for color and a *bool* for keeping track if the piece has been move. There are three important methods in this class and they are:

* *int pushValidMove(int, int, Piece\*, vector<string>, bool)*
* The method takes in two *int*s (destination coordinates x and y) and layout of the board (2-D array) in order to determine if that particular piece can successfully reach those coordinates. It returns a 0 if it is not possible and 1 if it is. The method also pushes the move to *vector* <string> *validMoves* if the move is possible.
* Note: this method does not inspect if the move is valid (i.e. if it puts itself in check)
* *void getAllValidMoves(Piece\*, bool)*
* This pure virtual method generates every single move that the piece can make and then calls *pushValidMove(…)* and passes its generated moves as parameters to it.
* *bool move(int, int, Piece\* )*
* This pure virtual method that is usually called by *Board::move(…)* to see if the piece that is occupied by the cell can moved to the destination x and y. The two *int*s correspond to this destination coordinates and the method also takes the layout of the board (2-D array) as a second parameter.
* The method then calls on *void getAllValidMoves(…)*, it then iterates over the populated *vector<string> validMoves* to see if destination coordinates match any move that are in the vector. If there is a match it returns true, else it returns false

**Player Class**

Relationship:

Aggregation: *Player* has a *Board*

Class Overview:

The *Player* class is an abstract class. *Human, Comp1, Comp2* and *Comp3* are all constructed using it. The constructor takes a pointer to a *Board* and a *char* to assign the player a color. There is only one important method in this class which is:

* *bool move()*
* This pure virtual method varies depending on who the player is. If the player is a *Human*, then the current and destination coordinates are taken using *cin*. However, if the player is either *Comp1, Comp2* or *Comp3* the coordinates are generated using their corresponding methods. Once the coordinates are acquired either by *cin* or generated *Board::move(…)* is then called

**View Class**

Relationship:

Aggregation: *Player* has a *Board*

Class Overview:

The *View* class is an abstract class. It is used to create two other classes they are *Text* and *Graphic*, and its constructor just takes in a pointer to a *Board*. There is only one important method in this class and that is:

* *Void draw()*
* This pure virtual method behaves differently in the *Text* and *Graphic* classes. In the *Text* class the method prints out the layout of the board using *cout*. Whereas in *Graphics* the layout is drawn using *XWindow*.

Overview Summary

While implementing and designing this game the team was thriving for a high cohesion and low coupling software design. Each class was intended to focus on only certain actions and this was mostly achieved in our final design, giving us a high cohesion design. For instance, the *Board* class only focuses on setting up, moving individual pieces and validating those move, whereas the *View* class only job is to print out the layout of the board. The same can be said for *Piece* where its only job is to initialize pieces on the board and maintain a list of all valid moves that particular piece can make.

On the other hand, maintaining a low coupling was a challenge because we did not want to duplicate code and as a result some methods rely on other methods that are only available in other classes. However, the team tried to keep this at a minimum. For instance, all of the Comp classes depend on *Board::move()* to generate an appropriate move. But if something major needed to be changed in *Board::move()*, depending on what the change is, the Comp implementations will require little to no change. This is because Comp does not care how *Board::move()* is implement it only requires appropriate return statements and as long as they match, Comp does not require any change. The same thing would apply for relationship between *Board::move()* and *Piece::move()*.

In conclusion, design of this game at some degree has achieve high cohesion and low coupling software design. As a result, accommodating to changes will be of relative ease.

# Questions:

1. Undo Feature: We implemented an undo feature on the board by using a stack. The stack is located in the board class, and contains the moves that have been used already, and the piece type of a piece that has been taken. One of the hard parts of implementing an undo feature is the need for remembrance of taken pieces – they must be reintroduced to the board after they’ve been deleted. The undo feature would not be able to use the move function to undo a move as if the piece was in check before move will not work. Thus we need to make the undo function directly affect the board and create a new piece as well. By creating a stack we allow for an unlimited number of undos, otherwise we’d just keep track of the listed things above for the last turn.
2. Four-handed chess: Fourhanded only requires small changes in implementation. First in our layout, we’d need to add 3 rows at the top, bottom, and sides. A way to do this would be to use a 14 x 14 array, and fill the 3x3 corners with a pointer that cannot be taken. Our pieces are identified by the char ‘b’ ‘w’, and we’d just add ‘B’ (blue), and ‘r’ for red. In the move implementation, we check to see if any move is in the valid moves. In the validMoves() function, we’d need to make almost all of our changes. Firstly, pawns will need to be altered to add side-side movement if the piece is ‘B’ or ‘r’ (this will also need to be done for castling). In addition, validMoves iterates through all possible moves. It currently checks to see if a cell is occupied and stops, and adds to the list of valid moves if the cell is enemy owned, and stops prematurely if the cell is self-owned. We’d need to add a little check there to see if (posX < 4 || posX > 11) && (posY < 4 || posY > 11).
3. Biggest lesson: The biggest lesson we learned is that it is very important to rigorously plan and organize tasks when developing software as a team. We fortunately recognized the importance to do this from the start and regularly met to discuss our future plans for this project. Ultimately, the transition from planning state to the implementation stage was crucial to get right. Because from that point on it was easier for us to build the program and implement all required features. This taught us building software in team environment can get sometimes complicated, however if you spend enough time planning the process becomes easier.
4. Start Over: If we had an opportunity to start over, we would definitely have better communication between one and other. This would have allowed us to work more efficiently as it would help the team identify their comfort zone and ultimately, motivate each team member to deliver quality code and an overall better product.