# CS 246 FINAL ASSIGNMENT PLAN

**Alexander F. (a22fu 20938923)**
**Ivan H. (i6hu 20931267)**
**Colin R. (c9ruan 20954742)**

**Due Date 1: Friday, November 25, 5pm**
**Due Date 2: Tuesday, December 6, 11:59pm**

## Plan of Attack:

By the end of Due Date 1, we plan to have finished the general design of our project through the creation of the UML, although the specifics may be subject to change. By this date, we have also planned out the delegation of tasks, as well as how teamwork will be handled in the setting of a group project. The header files will be worked on as a group, in order to accurately map out all of the connections between different components and the role of each of the functions, but the implementations themselves will be done individually, split among our group members. The first step we wish to take is to solidify the design we settled on by our UML, by first finishing all of the header files and the relationships between all of them. In this step, these files should be thoroughly commented on, describing the role of each function and variable, as well as how they affect the other parts of our project. After this, the work is fairly straightforward, with the implementation files as well as a focus of testing all of the different functions as we implement them.

**Due Date 1: November 25**

**November 26:**

- Write black box test cases for all intended behaviors, as well as for different errors that may occur, grouping all of this into a test suite for after all the implementation has happened.

**November 28**

- Write out the header files for the pieces abstract class, as well as all of the individual pieces.
- Write out the connections and header files for the board and player classes, making the appropriate relationships to each other and to the pieces abstract class.
- Write out the Observer Pattern, with all necessary connections, implementing the text based display.
- In this step, certain parts of our UML that we discover may not fully work as intended will be changed, and finalized.

**November 30**

- Delegate the different necessary implementation files, and write them according to the function specified through the header files. By this deadline we wish to finish the implementation of all the pieces and the players.
- Implement setup mode into the main interface. With the board and pieces classes finalized, can be easily implemented into our interface.

**December 1**

- Write the main function that allows people to interact with the program through the terminal.
- By this deadline, there should be a rough prototype that is playable through the text based display.

**December 2**

- Create and link the graphical display as another observer, updating with every single move.
- Write computer player levels 1 - 3, and include this into our program as different player subclasses.

**December 3**

- Research and write computer player level 4, linking this also into our player subclasses
- Begin using the blackbox testing suite made during the beginning of this project, while additionally writing new whitebox tests in order to make sure all of the different components coded work as they should

**December 4-6**

- Continue different methods of testing, and make sure all the different parts of the code are well documented and are clear to readers.
- Manually test the viability and function of the computer players, making sure that the moveset, particularly with computer player level 4, is viable and reasonable.
- If there is extra time, attempt the bonus features, (undo button, chess variants, standard opening variations)

**DD2: December 6**

## Chess-Based Questions:

**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 would implement a tree of moves. Each node would contain a move, and would point to an array/list of responses moves by the opponent, which would be represented as their own nodes containing the moves themselves. A computer implementing this book of responses would choose a random move in the array/list on its turn and move down that node and perform that move. On the opponent's turn, the computer would wait for the opponent's move then move down the corresponding node. This would continue until the opponent performs a move that isn't in the tree or the end of the tree is reached, in which the computer will start its actual "adaptive" programming based on what level it was selected as.

**Question: How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?**

- We can implement a stack from the start of the game which saves every single move made as a pair of coordinates for where the piece originated from and ended up, whether a piece was captured and the piece, and if it was the first move for that piece if it was a pawn, king or rook. Undo moves the piece back and resets any changed states on the board (first moves, taken pieces, promotions etc.), and removes that move from the stack.

This way an unlimited number of undos is possible, back to the beginning of the game. This lets us only need to store the pieces captured in a game rather than entire board states since the piece that was moved can be found by looking at the past move and the current board state.

**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.**

- Once we have implemented the rules, pieces and board, it would be quite simple to create variants by changing parts of our interface or setup. For four handed chess specifically, we would have to increase the number of players connected to the board, set two more different piece colors, and change the way the board setup is which can all easily be done by changing the default board state and our interface loops. Since four-handed chess includes four directions instead of the usual two, we would have to implement the same rules that are based on directions (promotions, castling, pawns only moving forward etc.) in the horizontal direction. In four-handed chess, the board shape is not just a 8x8 square. To store the current board state, a two dimensional array would suffice, but the corners would have to be empty spots to represent a square that cannot be visited and that would have to accompany a slight change in our valid move check.

## Program Design

The main portions of our design center around the board, which implements the grid of pieces, as well as moves them around, deciding which moves are allowed or not. Each of the pieces are a subclass of the pieces class, defining mainly abstract functions for the checking of the validity of moves. The pieces themselves will check for validity of the moves before they are made, but certain checks must be done on the board in order to make sure things such as discovered checks and similar are not allowed. This allows for the easy addition of extra pieces if one was so inclined or simply changing the code related to the logic of how pieces work through the pieces class without ever touching the individual subclasses. The board also checks for checks, checkmates, and stalemates after every move. The board is a subclass of the Subject class, which uses the subject observer pattern in order to allow for both graphical and text based observers to spectate the same game.

Since we needed to accommodate multiple different types of players, humans and various levels of computer difficulty, the way we decided to set up the main function was by interacting with two player objects, with five subclasses of the player class (human, computer[1-4]) who all move differently . The main function allows the actual interaction with the program, creating the board and observing it, and calling the player move functions, which call all of the board's functions. It is also responsible for setup mode, which is easily implemented through various calls to board functions which modify the board by adding and removing pieces.

One important design trait we had very early on that allowed for much of the checking for discovered checks or disallowing moves that would leave your king in check was the undo function, whereas we could play a move, check if there was any issue with the position then undo to the previous state if there was illegal portions. This was implemented with a stack of all previous moves that were played, with a robust undo function that was able to reverse engineer each of the moves made. We also had a stack of all the pieces that were captured, so that undoing a capture would simply take the top piece off the stack and place it where it had been before the capture. All of these functionalities were made to be compatible with the features of castling, en poisson and promotions.

In addition to human players, we also implemented computer players, with levels 1, 2, 3, 4 with different specifications. All players fall under a Players Class which share the same virtual move function and identical undo and resign functions, allowing for easy addition of additional computers or players as the need arises. This allows for plenty of design space with how you want the computers to move or if you would like to change the input specifications of a human. Our design for the difficulties of bots

The most important aspect of your design document is a discussion of how your chosen design accommodates change, as described earlier in this document. You should also discuss the cohesion and coupling of your chosen program modules. Most of the design portion of your grade will be based on this summary.

Generally, our program is resilient to change in program specifications. As all of our code is based on classes that we created, changes in program specification will be small and quick changes. For example, if we were to program a chess variant where all the back rank pieces start in random spots, the only thing that would need to be changed is the location where the pieces are placed. Other changes such as more computer levels or a larger board could be done relatively quickly.

## Bonus

We ended up implementing unlimited undos using the method that we specified in the initial question that we answered. As described above, it was implemented using a stack of moves, that had all the information necessary inside to undo them.

We also implemented a level 4 computer. This computer level is a combination of the other levels of computer in which it looks through possible moves and finds a random one with the highest scoring. It has many levels of scoring, but only looks at the next move. The highest scoring is a mate in one, so if there is a possible checkmate for the level 4 computer, it will spot it. The computer also looks for trading pieces for both sides. It takes the point value of both

pieces that will be exchanged, and sees if the point value is beneficial towards the computer. For example, if the computer had the opportunity to trade a knight for a rook, it would take the rook. Next, it also prioritizes promotions as it is equivalent to turning one pawn into a major piece, most likely a queen. Other types of moves that are generally better are castling and checks, so they have a higher point scoring over other moves, but lower than trading a piece and gaining value. The computer also tries to protect its pieces, similar to computer level 3. Whenever a piece is hanging, it will try to protect it if it has lower value than the attacking piece, or try to move it if it has higher value. If none of these conditions are satisfied, and there isn't a clear move for the computer to choose, it will choose a random move that is available.

## Changes from Original Plan

Most of the decisions we made were based on our beginning UML and plan, so there was not a lot of change that we needed to do. We needed to add a player class, in order to implement the human and computer players, which was not reflected on the original UML, but other then that, we made several small changes such as changing the logic for castling and promotions to the board, as well as making it so that the board made the moves. There were a lot of different additional helper functions implemented such as get and set functions, which due to the sheer number aren't all included in the UML for the sake of readability.

## End of Project Questions

1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

This project was the first time developing software in teams for most of our group. The most important lesson that was learned was how to divide up the coding, while every group member still understood the purpose of every method or class. This was especially important for our chess program, as all our classes and files depended on each other. It is usually more difficult to understand code written by someone else, but good organization and an attack plan for this project was very helpful. In this case, our UML was our base design for our chess game that helped structure our code. Another lesson was the importance of discussing the best approach to code a certain part of the program. For instance, there could be a variety of ways to code the undo function in chess, with some that are better than others. By working as a team, a better solution would usually be implemented because more ideas would be available. Lastly, we used Git and GitHub as a method to work together on a single code base. This project familiarized our group with how Git works and how it maintains version control with all the collaborators. It also introduced us to many common issues such as forgetting to commit code, or some git commands failing. We also realized the importance of separate compilation and splitting our code into multiple files through Git as well. If everything were to be in one file, it would be much more difficult to work as a group to push changes as it would create issues if there are two different

commits on one branch. If we had worked alone, the importance of separate compilation is also very important. As our code is quite long, scrolling through a few files to find the method that needs to be changed is very difficult in only one file. In addition, writing large programs takes a long time, so if I had worked alone, this project would have taken much longer. Thus, I would have had to spend more time on it, so I would have needed to devote more time daily to create it.

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

When we started, the deadlines gave very little leeway for the amount of time that we had for each individual part, and certain changes to the UML and the way that our program was structured created unexpected delays that worked against our initial schedule. Instead of putting all of the deadlines so close together, having more flexibility would have been better. Starting earlier (before deadline 1) for the project would have also been nice to have a bigger window that we could work on it.

Many of the different components of the project were very codependent on each other, so sometimes if there were big issues with one portion of it, then other parts would have stalls in order to figure it out. More time put into the better implementation of the UML would have solved many of these issues.

Other than those, we didn't have many major issues that we weren’t able to tackle, and despite certain bumps in the road, we are all very proud of how the end product turned out.