Plan of Attack:

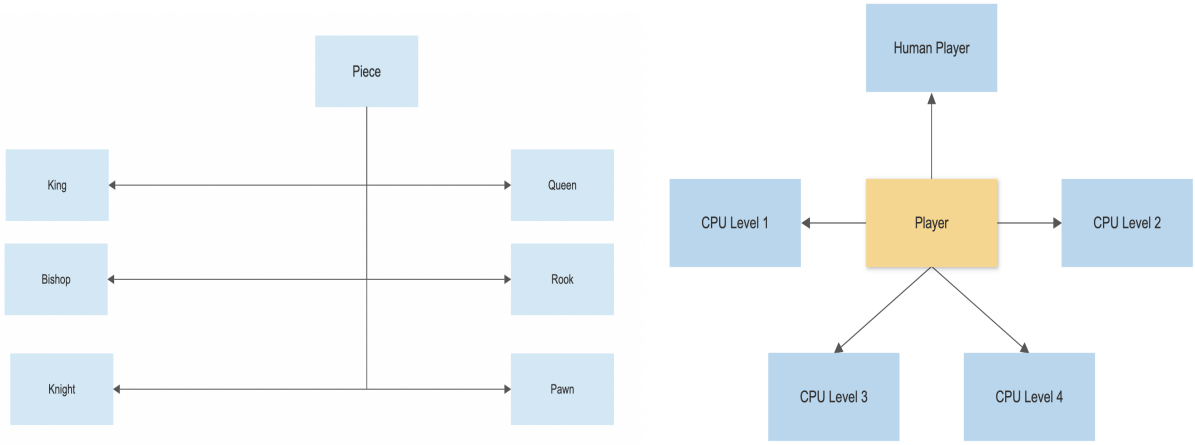
Members of the Group:

Manyuvraj Singh Sandhu, Jackson Howe, Blake Naumann

Plan:

The chosen project is chess. We first conducted a group meeting to discuss the UML and the classes and inheritance relations we would need to build this game of chess on November 21st. The UML and responsibilities were thoroughly discussed in this group meeting. Then, we discussed the answers to the questions posed in the document chess.pdf and the solutions to the problems, and finalized them by November 22nd.

The first programming step of our plan is to build the interfaces and code the class shells required to build this game. This will be done by November 23rd. This will consist of creating all the .h files required for the main classes in the program. For example, the `Board` class – which will handle the state of the board, the `Game` class – which contains the methods to update the board, the `Player` superclass – which subclasses into different levels of computer players and a human player, and a `Piece` superclass – which subclasses into the six types of pieces of the chess game will all have their fields and methods translated to `.h` files. This will be done by all three of the group members.



The second step is to build a rough command line interpreter and setup mode. The command interpreter will support commands that will be used to interact with the game program. All the commands like `setup`, `done`, `game`, `resign` and `move` will be supported by this interpreter. The majority of the initial work will be to get the setup mode working to facilitate testing. This will be done by Jackson Howe and Blake Naumann by November 24th.

The third step is to build the `Board` class which handles the state of the board, and to get the implementation of the `Piece` superclass and all six subclasses done. The subclasses of `Piece` include `King`, `Queen`, `Bishop`, `Rook`, `Knight`, and `Pawn`. Each piece will have a `getType()` and a `getColour()` method. Moreover, they will have an `isValidMove()` method which determines if the move made by a piece is valid or not. This will restrict the movement of every piece according to the rules of the game of chess. This will be done by Blake Naumann and Manyuvraj Singh Sandhu by November 25th.

The fourth step is to build the implementation of the `Game` class and to get the text view of the chessboard done. The `Game` class handles the state of the game and manages the rules. It is

what will be interacted with through the command line interpreter. This will be done by Jackson Howe and Manyuvraj Singh Sandhu by November 26th.

The fifth step is to get the implementation of the `Player` superclass done. Initially, we will work on the Human Player and level 1 of Computer Player. Once those are done, we will look into some more complexity for the higher levels of the `Computer` Player. This will be done by Blake Naumann by November 27th.

The sixth step is to look into and design the Graphics for the chess game. This will be done simultaneously with step five after the Level 1 computer and human `Player` classes are implemented correctly. This will be done by Jackson Howe by December 1st.

The final programming step is to implement higher levels of the `Computer` class, namely levels 2,3, and 4. The current plan for the level 4 `Computer` class is to utilize one layer decision trees whilst prioritizing avoiding capture, checks, and checkmates. This will be done by Manyuvraj Singh Sandhu and Blake Naumann by December 4th.

The Design Document and the code Documentation will be written side-by-side after every step and the command interpreter will also be updated side-by-side after every step.
As we build this game, we will be updating the UML and our plan of attack (if there are any changes) and update the Final Design Document, explaining more in depth how each class is implemented and we will answer the questions posed regarding the lessons learned while working on a team project. This will be done by all three group members by December 5th

Answers to Questions posed in chess.pdf A5:

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.

In order to have opening moves, and be able to respond to following moves, we would use a tree structure that branches at each move of the opponent. For each move the opponent makes, it leads us to a corresponding move to make, down the tree. This tree would contain references to board layouts corresponding to each given point in a tree. The board layouts will be stored as 2-dimensional arrays or vectors of pointers to objects of the `Piece` class. This way, to follow the move from the book, we can copy the board configuration from the current position in the tree. We recognize that this saves a fair amount of redundant information, and it would be potentially more optimal to store a tree with position pairs representing the move to be made, however this would require a more complicated decoding of the opponent's move, rather than a simple board comparison to determine what branch of the tree to follow. It also makes it more difficult to execute a move, as it requires either passing the optimal move to the user, or passing it directly in as input, rather than copying the board state. We expect to have a function that swaps the current board with a new one, which is involved with validating moves, so this swap function would already exist, and would thus be easier for us to implement.

This implementation assumes that the book only considers "good" moves for the opponent to make, and assumes that discussing the first "dozen or so" moves means between both players, not "a dozen or so" each. If either of these assumptions are not met, the size of the decision tree would become unreasonably large, even if we stored the moves more efficiently, such as in a tuple of positions.

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

Similar to the opening handbook, we would store previous board configurations. In the case of a single undo, we would store the previous board configuration with a simple pointer or reference to the 2-d array or vector in a variable. In the case of multiple undos, we would store previous board configurations by pushing pointers onto a stack data structure, and popping them off to facilitate an undo. We are confident that our stack will be large enough to store all previous moves since there are only finite moves a player can undo.

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.

First, we would have to modify the board size and shape, and would likely store it as a large square, with some values in the corner areas to specify those as non-playable spaces. This means we would have to modify our `validateMove()` method to check if a move crosses a non-playable space. We would also need to expand our `getState()` method that checks to see if a move puts an opponent in check or checkmate, as well as if it puts the player in checkmate, as there are now two other opponents to consider. Of course our setup mode and game command interpreter loop would be modified to support four players rather than two, and we would need to keep track of four players' results. Finally, we would need to add new cases into the `validateMove()` method for the `Pawn` class, to facilitate pawns moving left or right as part of the two new opponents.