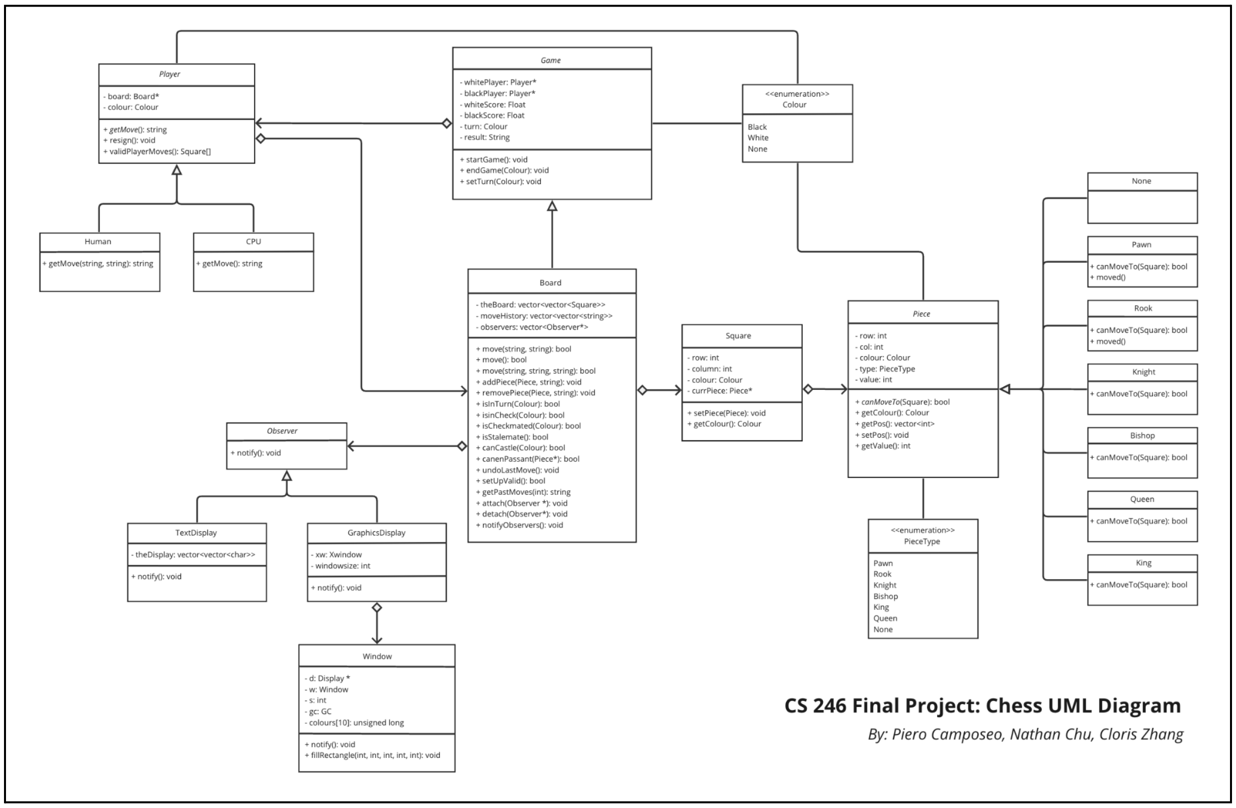
<u>CS 246 Fall 2023 Chess Project: Plan of Attack</u>

1) Breakdown of Project

For our CS 246 Final Project, we have chosen to design and implement a fully functional object-oriented chess game. Chess is played on an 8x8 chessboard, configured with a white square at the bottom right. The game progresses as players alternate, each taking a single move at a time, until the match's conclusion. This project involves a text-based display for simple representation of the system and a graphical display for a visually appealing display of the chessboard, and follows the intricacies of the standard chess rules and strategies. Furthermore, support for various player types will be implemented such as accommodating both human and computer players of varying difficulty levels. A robust testing and debugging process will be critical to ensure the system's reliability with the possibility of implementing additional features later on.

2) UML

We designed a Unified Modeling Language (UML) diagram to help illustrate the overall layout and structure of our chess game, depicting the classes, relationships and methods included. Our goal is to minimize coupling while maximizing cohesion, and have planned all our project features to accommodate change, with minimal modification to our original program.



3) Timeline Overview

	Task	Start Date	Due Date	Assigned To
	UML Diagram & Plan	11/20/2023	**11/24/2023**	All
	Write Test Cases	11/24/2023	11/29/2023	All
State of Game				
Get Board to Appear	Implement Game	11/23	11/24 by EOD	Cloris
	Implement Player, Human	11/23	11/24 by EOD	
	Implement Square	11/23	11/24 by EOD	
	Implement Piece, N,P,R,K,B,Q,K	11/23	11/24 by EOD	
	Implement Board	11/23	11/24 by EOD	Nathan
	Implement Observer	11/23	11/24 by EOD	
	Implement TextDisplay	11/23	11/24 by EOD	
	Implement Command Interpreter	11/23	11/24 by EOD	
Get Pieces on Board	Implement Square	11/24	11/24 by EOD	Piero
	Implement Piece	11/24	11/24 by EOD	
	Implement None, Pawn, Rook, Knight, Bishop, Queen, King	11/24	11/24 by EOD	
Get Setup to Work	Implement Board	11/24	11/24 by EOD	Cloris
	Implement Command Interpreter	11/24	**11/24 by EOD**	
*Get Pieces to Move Pseudo-Legally	Implement Piece	11/25	11/26 by EOD	Nathan
	Implement None, Pawn, Rook, Knight, Bishop, Queen, King	11/25	11/26 by EOD	
	Implement Board	11/25	11/26 by EOD	Piero
	Implement Player, Human	11/25	11/26 by EOD	Cloris
	Implement Command Interpreter	11/25	11/26 by EOD	

*Get Pieces to Move Legally	Implement Piece, None, Pawn, Rook, Knight, Bishop, Queen, King	11/26	11/28 by EOD	Nathan
	Implement Board	11/26	11/28 by EOD	Piero
	Implement Player, Human	11/26	11/28 by EOD	Cloris
	Implement Command Interpreter	11/26	11/28 by EOD	
Get a Game to Work (Human vs Human)	Implement Player, Human	11/27	11/28 by EOD	Nathan
	Implement Game	11/27	11/28 by EOD	
	Implement Command Interpreter	11/27	**11/28 by EOD**	
Implement CPU	Implement CPU (1, 2, 3)	11/28	12/01 by EOD	Cloris (1) Nathan (2) Piero (3)
	Implement Command Interpreter	11/28	12/01 by EOD	Piero
Implement Graphical Display	Implement Graphical Display, Window	11/28	12/01 by EOD	Cloris
	Implement Board, Observer	11/28	12/01 by EOD	
Implement Advanced CPU	Implement CPU (4)	TBD	12/01 by EOD	Nathan, Piero
	Implement Command Interpreter	TBD	**12/01 by EOD**	Nathan, Piero
	Testing & Debugging	12/01/2023	12/04/2023	All
	Implement Additional Features	TBD	TBD	All
	Documentation (Design, UML)	12/03/2023	12/04/2023	All
	Final Review	12/03/2023	**12/05/2023**	All

4) Plan

With the help of our UML diagram, we have devised a comprehensive timeline and breakdown of tasks to guide our chess game's implementation. We decided to break our project down and work in stages, each representing a state of the program. For instance, getting the board to appear, then getting the pieces to appear on the board, then ensuring the pieces on the board move legally, etc.

Before beginning work on each stage of the project we will create test cases to help further guide us on how we to implement each of our classes and methods to the desired result. By doing so, the process of programming the classes and methods should logically follow how the game is played from the command line ensuring both correctness and efficiency in our code. Once we've completed programming each set of classes and methods required at the stage we are on, we will run through the tests that we created and fix any unwanted behaviour, by going back and debugging. This way, we are able to continuously test after each stage is implemented and this iterative process allows us to refine our implementation throughout.

Here is a brief overview of each class required for our game and their purpose:

<u>Game</u>
Abstract class representing a game with a black and white player, who each have scores. Players take turns in a Game. The board inherits the game.
<u>Player</u>
Abstract class representing a chess player. The player has a colour and a board. They can make a move based on the board, resign, or return all of their valid moves.
<u>Human</u>
Concrete Player. The Human moves based on a predetermined move.
<u>CPU</u>
Concrete Player. The CPU has a level, and moves based on a series of calculations. The calculations are more complex if the CPU has a higher level
<u>Board</u>
Concrete Game. The board is represented by 64 squares, has a history of moves, and is a subject for some observers. The board can move its pieces, edit its pieces, give information about its state (like if a certain piece can move to a certain square), and give information related to the validity of a move.
<u>Square</u>
1/64 of a chess board. The square has a row, a column, a piece, and a colour. It can give information about itself, and set its own piece.

Piece

Abstract class representing a piece on a chess board. A piece has a row, a column, a colour, a type, and a value. It can set its own position, give information about itself, and give information regarding its capabilities to move to a certain square.

None, Pawn, Rook, Knight, Bishop, Queen, King

Concrete Piece. May contain additional information for special moves.

Observer

Abstract class representing something that needs to be notified when another object in the program changes. The observer can be notified.

TextDisplay

Concrete observer. Displays a board as a vector of characters. Can be notified when the board changes.

GraphicsDisplay

Subclass of the abstract base class observer, which will be an observer of the board object. Can be notified when the board changes.

Window

Subclass of the GraphicsDisplay class which handles the mechanics of getting the graphics to appear on-screen in a windowed format.

5) **Answers to 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. See for example https://www.chess. com/explorer which lists starting moves, possible responses, and historic win/draw/loss percentages. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

In order to implement a book of standard openings, we could take two different approaches both using learnt data structures which we will discuss. The first method we could use to implement the standard book of openings is to essentially keep a database of opening lines with a **map**. In this map, we would store the sequence of moves in the format of a single string as the keys of the map and have the values of the map be the name of the standard opening. Throughout the first few moves of the game, we would store the sequence of moves that have occurred so far and then perform a lookup operation on the map to give us the name of the standard opening or line at each move. In another approach, we could construct a **tree** with the leaf nodes of the tree containing the standard opening name, the root node signifying the starting position and child nodes representing the subsequent moves. We would implement this using a Node class for the root of the tree which would then have an array of pointers to Nodes representing the subsequent children of each Node. Within each Node, we could store the move as a string and then when we perform the lookup for the standard opening we would then traverse the tree based on the sequence of moves that have been played so far until we reach a leaf node of the tree, signifying the end of a specific opening sequence.

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

To incorporate an additional undo feature which will allow a player to take back a move we will need to have some method of keeping track of the previous board states. Let us say for example that a piece was captured during a player's move, but the player wishes to take back that move. We would need to remember the previous state of the board where the opponent's piece was not captured and the moved piece was still on its starting square. With this board state in memory, we would require some way to recall that board state from memory and display that again when a player wishes to undo their move. To accomplish this we could consider using a **stack** that would store all the previous states of the board after each player's move has been made so that when we want to undo a move, all we would need to do is pop that board state off the stack and recall the previous board state (last in, first out). In terms of creating the stack, we would create a Node class representing the bottom of the stack which would contain a pointer to another Node representing the next element on the stack and a board state value which would be a string written in Forsyth–Edwards Notation (FEN) representing the board's state at a particular moment or use built-in linked lists. This would allow for an unlimited number of undos until the start of the game, however, if we wanted to recover the undo move we would have to store all those popped values elsewhere and then push them back onto the stack to keep track of the states of the board again.

A more intuitive method to facilitate the undo feature which would also allow for easier recovery of the current state or even any other state of the game would be to make use of a double-ended queue (**deque**). Similar to how we would make use of a stack, we would store all the previous states of the board as FEN strings within the nodes of a doubly-linked list representing our deque. With the use of a deque, if we wanted to undo a move, all we would need to do is traverse the deque backward from whichever node we are currently on, which is only possible because of the doubly-linked nature of all the nodes. In addition to adding the undo functionality to the game, the deque would also allow for redos of moves in a fairly intuitive manner since to redo a move, we would just need to move forward within the deque. Unlike a stack, with a deque, we would be able to locate any played state on the board (bidirectional traversal) and recall the state to the game. This could be helpful for additional features such as game analysis, where players can usually go to any state of the game freely to analyze positions.

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.

Four-handed chess is played on an 8x8 chessboard with an additional 3 rows of 8 squares that protrude from the sides forming a "plus" shape. To allow for this new board size we would have to make changes to our board class within our program to be initialized to this new structure. This would involve increasing the number of vectors stored within our "theBoard" field to accommodate for the increase in the size of the board. Although the number of vectors has increased, the "plus" shape requires a few extra changes to ensure the proper behaviour of pieces. We would need some way of telling the pieces that the 3x3 squares at the corners of the board are now dead zones and that no piece can actually move those squares. This could be achieved by making some vectors smaller than others to represent unreachable squares or we could keep all the vectors the same size and modify our Square class to be an abstract class containing valid and invalid squares.

Since the board now takes a different shape, the pawns promote in the middle of the board instead of at the end which means that in our program we would have to have a new method of checking whether a pawn is capable of promoting or not. On chess.com, free-for-all four-handed chess follows a point system which we currently have in our Game class. Since the game would have 4 players we would need to add in more player pointers to get each player's individual moves and more player score fields within our Game class to keep track of who is currently winning the game. Instead of having four separate data fields as pointers to player objects, we could change it so that we have a vector of player pointers. This change would facilitate a more scalable approach, allowing for easy adaptation to future variants. With four players, there is of course a new win condition which is when three players are defeated or the game simplifies down to a 2 person game and one person has 20 or more points over the other player in terms of piece captures. To account for this, we would need to modify our main file to ensure we end the game at the proper time. If we want lower coupling within our program we could add in a rules class that would be responsible for determining whether moves are legal, checking if checks, checkmates and stalemates are on the board, and making moves. This way, all the changes that would be necessary would mostly happen inside the rules class. Having a separate rules class would additionally allow for the implementations of other variants to be much easier as it would be the only class we would need to modify most of the time.