Project Breakdown

Task	Member	Description	Expected Due Date
Main	Ken	Create Match, Call game_init, and interpreter	Nov. 29th
Match	Ken	Initialize board, players. Command control flow	Nov. 29th
Player	Ken	Read Player's move from std input	Nov. 29th
Computer	Ken	Computer generates move based on different levels	Dec. 1st
Board	Tim	Facilitate the board piece placement and connect piece positions to graphic display	Nov. 29th
check_checkmate	Tim	Scan the board and check if there are any legal moves for the opposite color King	Nov. 29th
check_stalemate	Tim	When king is not in check and there's no more legal move	Nov. 29th
check_draw	Tim	Check if there's	Nov. 29th

		sufficient material on the board	
Pieces	Tim	Responsible for piece’s legal move check	Nov. 28th
Pawn	Ken	Pieces’ individual attributes and check their own legal moves	Nov. 28th
King	Sam		Nov. 28th
Pawn	Ken		Nov. 28th
Knight	Sam		Nov. 28th
Bishop	Ken		Nov. 28th
Rook	Sam		Nov. 28th
Queen	Ken		Nov. 28th
Empty	Sam		Nov. 28th
Display	Tim	Contains TextDisplay and GraphicDisplay	Nov. 28th
TextDisplay	Tim	Display the board’s current status with Text in the Terminal	Nov. 29th
GrahpicDisplay	Tim	Visualize the board using XWindow	Nov. 29th
XWindow	Tim	The interface helps GraphicDisplay to visualize	Nov. 29th

Match Class: This class manages the overall chess game. It keeps track of scores for white and black players, references the game board, and holds instances of the two players. Key functions include game_init() to start a new game and interpreter() to process and apply player inputs.

Player Class: Represents a player in the chess game. It holds a reference to the board, player color, and turn status. The `generate_move()` function is responsible for generating moves for the player, which could be extended for AI functionality in the case of computer players.

Computer Class: Inherits from the Player class, with an additional attribute `level` indicating the difficulty level of the AI.

XWindow Class: Handles the graphical display aspects of the game, including drawing rectangles and strings on the window. Functions like `fillRectangle()` and `drawString()` are used to render the game board and text on the screen.

TextDisplay and GraphicDisplay Classes: Both classes are responsible for different modes of displaying the game. They update the game's visual representation when notified of changes in the game state. They inherit from a common `Display` interface, which requires a `notify()` method.

Board Class: Represents the chessboard. It stores the positions of all the pieces and maintains the turn status. Functions include `board_init()` for initializing the board, methods like place_piece() for placing pieces on the board, and various checks for game conditions like check, checkmate, stalemate, and draw.

Pieces Class (and subclasses): This is an abstract class representing a generic chess piece, with subclasses for specific types of pieces (Pawn, Knight, Bishop, Rook, Queen, King, and Empty). Common attributes include position, color, and a reference to the board. Key methods are `move()` for moving pieces, `check_legal_move()` for validating moves, and `get_state()` for getting the state of a piece. Specific piece classes may have additional attributes (like `moved` for Pawns, Rooks, and Kings) and override some methods (like `piece_check()`).

King Class: A subclass of Pieces, with a boolean attribute `moved` indicating if the king has moved (important for castling rules). Includes a custom implementation of `piece_check()` specific to king movements.

Project Specification 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.

To manage different chess openings efficiently, we will employ a structured approach using multiple text files, each named after a specific opening. For example, we'll have files like 'Italian_opening.txt', 'Sicilian_Defense.txt', and so on, corresponding to various popular openings. Within each file, the sequences of moves that define the opening will be detailed. Our chess program will read these files to access the required opening moves. This method enables easy organization and updating of individual openings. When a match starts, the program will select the appropriate file based on the opening chosen or recognized from the opponent's moves. This system provides a straightforward and modular way to incorporate a wide range of chess openings into our program, enhancing its play style and adaptability.

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

1. Single Undo Functionality: For allowing a player to undo their last move, we'll create a copy of the chessboard before each move is made. This copy serves as a snapshot of the game state prior to the move. When a player opts to undo, we simply revert to this copied board. Additionally, this copied board is used to verify the legality of a move. If a move is found to be illegal, the program disregards the temporary board and prompts the player to make a different move.

2. Unlimited Undos: To facilitate an unlimited number of undos, we will maintain a vector (or a list) that stores the state of the board after each move. This vector acts as a history of the game's progress. Each element in the vector represents the state of the board at a particular point in the game. When a player wishes to undo multiple moves, the program will traverse this vector backwards, moving to the previous state with each undo action. This allows players to revert the game to any previous state, effectively providing an unlimited undo capability.

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.

Expanding the Match Class: The Match class will be updated to include two additional players, each with their own scores. This modification will manage the flow of the game among four players instead of two. The turn-taking mechanism needs to be restructured to ensure that each player gets their turn in a defined order.

Board Redesign: The Board class requires significant changes. The traditional 8x8 chessboard will be expanded to accommodate more pieces and provide playing space for four players. This might involve creating a larger board or a non-standard shaped board, common in four-handed chess variants.

Piece Class Adjustments: In the Piece class, the logic for each chess piece needs to be revised to account for the presence of pieces from two additional players. The movement and attack patterns of each piece should be redefined to consider interactions with multiple opponents. This involves not only checking for standard chess threats but also strategizing against moves from two additional players.

Handling Checks and Checkmates: The program must be equipped to handle checks and checkmates in a more complex environment where a player could be threatened by any of the other three players. The logic for determining check and checkmate situations must be adapted to this multi-player scenario.

Game Rules and Logic: Finally, the game rules need to be updated or customized to fit the variant being played. Four-handed chess can come with its own set of rules and winning conditions, which the program must faithfully implement.