Design Document

Jungle game

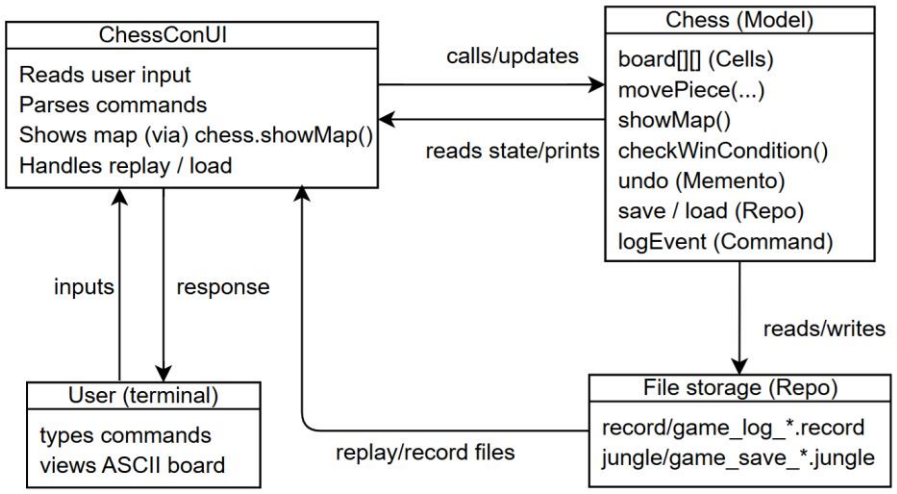
Group 104, Fall 2025

1 Architecture Design

The Jungle Game (JG) is implemented using the Model-View-Controller (MVC) design pattern. This architecture is chosen to keep the game rules and state management separate from I/O handling, for easier testing, streamlined support for replay and undo features, and reduce overall complexity.

Chess (Model) is responsible for full game state (board[][], Cell, Piece, Player), rules and move validation (movePiece), win detection (checkWinCondition), undo storage (boardHistory), logging (logEvent), setting up game (GameSetUp).

ChessConUI (View + Controller) is responsible for rendering board and messages to the terminal; collect and parses user input commands (*start, move, save, undo, load, replay, exit*), invoke corresponding Model operations (*movePiece, undo, GameSetUp*), drive main game loop by coordinating user interaction with the game state.



2 Structure of and Relationship among Main Code Components

The methods of each class are described below:

Class	Method	Argument Name and Type	Return Type	Exception Declarations (if any)	Explanation
Chess	checkWinCondition	N/A	void	An *IOException* occurs when an input or output operation fails or encounters an unexpected problem during execution.	Check if either player has won the game. The game ends if all opponent's pieces captured, or a piece enters opponent's den. The boolean *gameOver* turns true when a win occurs and updates *gameOver* and *winnerName*.
	isGameOver	N/A	boolean		A get method for game state.
	getWinnerName	N/A	String		A get method for winner's name.
	placePieces	N/A	void		Places all pieces in their starting positions.
	countRemainingPieces	Side side	int		Counts the number of pieces remain for a given side.
	Capture	Piece attacker, Piece defender, Cell fromCell, Cell toCell	boolean		It implements the capture logic for pieces. Higher rank pieces can capture lower rank ones. Special rules are as follows: ● Rat can capture Elephant ● Elephant cannot capture Rat ● Rat can only capture if both are in river or are on land ● Pieces in traps can be captured regardless of rank
	movePiece	int fromRow, int fromCol, int toRow,	boolean		Before moving a piece, the method first save a board copy of the current board. It validates moves based on the game rules (e.g.

		int toCol			horizontal/vertical jump movement, Tiger/Lion river jumping, rat on river). It also checks if the player moves the piece in bound and moving a piece at the source position. Players are allowed to move their own pieces only and cannot move to any self-occupied cell. After moving, the movement is recorded into *logEvent*, and switch to the next player and check the win condition by *checkWinCondition()*.
	undo	N/A	boolean	An *IOException* occurs when an input or output operation fails or encounters an unexpected problem during execution.	Restores up to 3 previous board state per player in each game. It first checks if there are movements to undo, and undo allowance for the player who is invoking undo. Then the method creates a deep copy of the previous board state into the current board, update *boardHistoryIndex* and *boardHistoryCount*, *currentPlayer*, remaining undo limit, and log into the record. The method works as follows (see Section 3): ● If red player requests to undo his/her move, the "undo" input is entered under blue's turn, i.e. when blue is the current active player. ● The method returns the most recent state before red moves. ● The remaining undo count for red player would reduce by one. ● Red player becomes

					the current active player.
	inBounds	int row, int col	boolean		Check if a position is within the board.
	setPlayerName	Side side, String name	void		Assign manual input names for red and blue players respectively.
	getPlayerName	Side side	String		A get method for player's name.
	getCurrentPlayer	N/A	Player		Returns the current active player.
	setRandomPlayerNames	N/A	void		Assigns random names to players from a predefined list. It ensures no duplicate names are assigned.
	GameSetUp	N/A	void	An *IOException* occurs when an input or output operation fails or encounters an unexpected problem during execution.	Initializes a new game by initializing the board and placing pieces with *initializeBoardHistory()* and *initializeBoard()*. If there is no record folder, it creates a new directory for the new game.
	getNextLogNumber	File recordDir	int		Get next available log number in record or jungle folder.
	logEvent	String event	void	If no current game log is set or writing fails. Tries the fallback by adding a timestamp.	Records user's events to the current game log file.
	initializeBoardHistory	N/A	void		Initializes the board history. Creates an array to store the most recent 3 board states, and resets history counters.
	initializeBoard	N/A	void		Set up the initial board for new game, and place pieces by calling *placePieces()*.
	showMap	N/A	void		Display the current game state with pieces, rivers, dens, traps, current active player, and both players' states.

	saveGameToJungle	String id	boolean	An *IOException* occurs when an input or output operation fails or encounters an unexpected problem during execution.	Save game state to ".jungle" file and handles file naming and directory creation. Record player names, current turn, undo counts, and pieces positions into the file. If there is no jungle folder, it creates a new directory for the new game.
	loadGameToJungle	String id	boolean	An *IOException* occurs when an input or output operation fails or encounters an unexpected problem during execution. Also if any malform commands that cannot be parsed.	Load game state from file. Parsed the commands by *initializineBoard()*, resetting game state by parsing the commands in the file.
ChessConUI	main	String[] args	void	An *IOException* occurs when an input or output operation fails or encounters an unexpected problem during execution.	Main JG game loop that handles the menu: start, load, replay, or exit. Loops until user inputs exit.
	startGame	N/A	void	If the move's Column is not between A-G, or the move is invalid, then it will throw exception.	Initializes a new game session by *chess.GameSetUp* . It first asks for player's names. Leave the input empty if player wishes to use random name with *chess.setRandomPlayerName, chess.setPlayerName* . It loops the game by supporting commands including: ● move [srcPos] [desPos]

					• undo • save • stop After user inputs, the method would validate its inputs and coordinates. User can continue the game after saving. The loop continues until the game is ended, or stopped.
	loadFromFile	N/A	void	If no current game log is set or read fails. If move coordinates cannot be parsed, out of bounds, a cell reference is null, or any other unexpected runtime exceptions occurs.	Handles replay of recorded games from ".record" file. The method takes a record number as input, then reads and parses game record files. The method offers two replay modes: auto or wait for user's input between moves. User can cancel replaying by typing "cancel" in the "step" mode. The record loads by preparing a fresh game for replay, calling *chess.GameSetUp()*, and handles commands found in record. It shows the game progression and final game state.
	loadSavedGame	N/A	void		The method loads previous game states from ".jungle" file. It takes a record number as input, then reads and parses game record files, calling *Chess.loadGameFromJungle* and *startGame()*. After restoring, players can continue their loaded game. The game continues until game ends or manual stop.

The following UML class diagram of JG system shows the relationships and structure among the major code components.

ChessConUI depends and uses Chess. It also acts as a console View + Controller.

Commented [KC1]: I NEED HELP

Commented [C[2R1]: explained, plz check

Commented [KC3R1]: I give up, Elaine help me

Reads user input, calls Chess APIs (GameSetUp, movePiece, undo, save/load, showMap), and print output. ChessConUI depends on Chess by holding its reference and drives game replay logic.

Users interact with the ChessConUI by inputting commands, in which the UI prints the results.

ChessConUI calls Chess when any of the following is invoked: start / move / undo / save / load.

ChessConUI reads record files for replay, replays commands by calling Chess (with review mode enabled so Chess doesn't re-log).

Chess contains Cell[][] (board). It acts as domain Model, as well as maintaining board state, rules, move validation, win detection, undo history, logging, save/load.

Chess acts as a 2D array of Cell objects (board [][])or aggregation to the Chess board.

Chess updates model (board, boardHistory, currentPlayer), calls logEvent to write records.

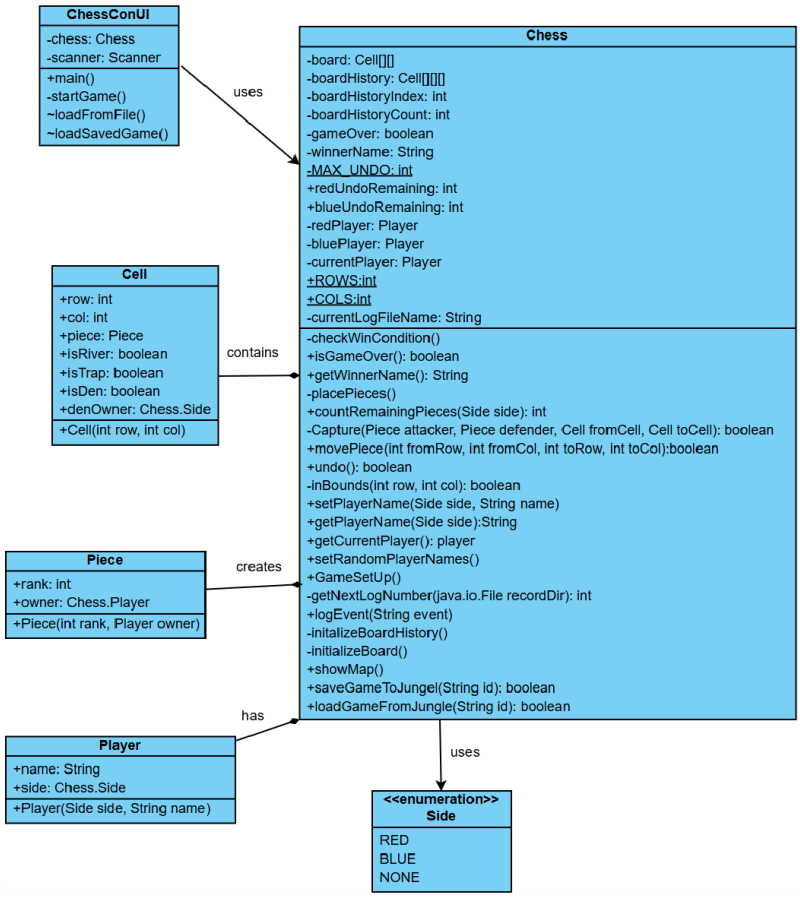
The relationship of Chess and Filesystem is that logEvent and save/load persist state or records.

Chess creates or instantiates Piece, in which Piece is a small value object (rank + owner). Chess constructs Piece instances when placing/restoring pieces.

Cell contains Piece (0..1).It performs cell state (row/col, flags: isRiver/isTrap/isDen, actionLabel). It may hold a Piece.

Chess associates Player (redPlayer, bluePlayer). Player holds name and Side; Pieces reference a Player as owner; currentPlayer tracks whose turn it is.

Chess uses filesystem (record, jungle files) as repository. logEvent writes move/undo/start lines; saveGameToJungle / loadGameFromJungle read/write .jungle saves. ChessConUI reads record file id for replay or load a saved game.



3 Example Use

3.1 Main flow of JG

The following UML activity diagram of JG system mainly demonstrates how users implement JG based on the above major code components.

Top-level command loop

- Entry: ChessConUI.main(...) prompts: start | load | replay | exit.
- Dispatch: switch in main() calls one of:

- start -> ChessConUI.startGame()
- load -> ChessConUI.loadSavedGame()
- replay -> ChessConUI.loadFromFile()
- exit -> close scanner and return

Start a new game (left branch on diagram)

1. ChessConUI.startGame()
2. Read names; call chess.setRandomPlayerNames() and/or chess.setPlayerName(...) as needed.
3. Start model: chess.GameStart(false)

- inside GameStart(): initializeBoardHistory(), initializeBoard(), reset undo counters, create a log file and call logEvent("start ...") (unless review mode).

4. Enter game loop: while (!chess.isGameOver())

- chess.showMap() prints board and status
- read command: move / undo / save / stop

  - move: parse positions then chess.movePiece(fromRow, fromCol, toRow, toCol)
    - movePiece() saves a deep copy into boardHistory, validates move, possibly calls Capture(...), performs the move, logEvent("move ..."), switches currentPlayer, then checkWinCondition()
  - undo: chess.undo()
    - undo() restores previous board from boardHistory, updates undo counters, switches player back, logEvent("undo")
  - save: chess.saveGameToJungle(id)
    - saveGameToJungle(...) writes current game state to jungle/*.jungle
  - stop: return from startGame() (breaks to main menu)

5. When chess.isGameOver() becomes true, ChessConUI prints "Game over! Winner: " + chess.getWinnerName()

Load saved game (middle branch)

1. ChessConUI.loadSavedGame()
2. Validate id and call new Chess().loadGameFromJungle(id) to test; then instantiate chess = new Chess() and chess.loadGameFromJungle(id)
   - loadGameFromJungle(id) reads jungle/*.jungle, calls initializeBoard(), clears pieces then repopulates board and restores current player and undo counts
3. After successful load, enter the same play loop as in startGame() (use chess.showMap(), chess.movePiece(...), chess.undo(), chess.saveGameToJungle(...), etc.)
4. Finish when chess.isGameOver() → print chess.getWinnerName()

Replay recorded game (right branch)

1. ChessConUI.loadFromFile()
2. Read record file (record/game_log_N.record) into commands list
3. Ask user for replay mode: "auto" or "step" (or cancel)
4. Prepare fresh chess = new Chess(); review flag is used:
   - When replay triggers starting the model, code calls chess.GameStart(true) so review = true and logEvent() is suppressed during replay
5. Iterate commands from file:
   - "start ..." -> set names via chess.setPlayerName(...) or chess.setRandomPlayerNames(); call chess.GameStart(true)
   - "move A7 A6" -> parse and call chess.movePiece(fromRow, fromCol, toRow, toCol)
   - "undo" -> chess.undo()
   - "stop" -> set stopReplay = true and break
6. After each command: chess.showMap(); check chess.isGameOver() and chess.getWinnerName()
7. Replay pacing:
   - mode == "step" -> prompt user to press Enter (or "cancel")

- mode == "auto" -> Thread.sleep(1500) between commands

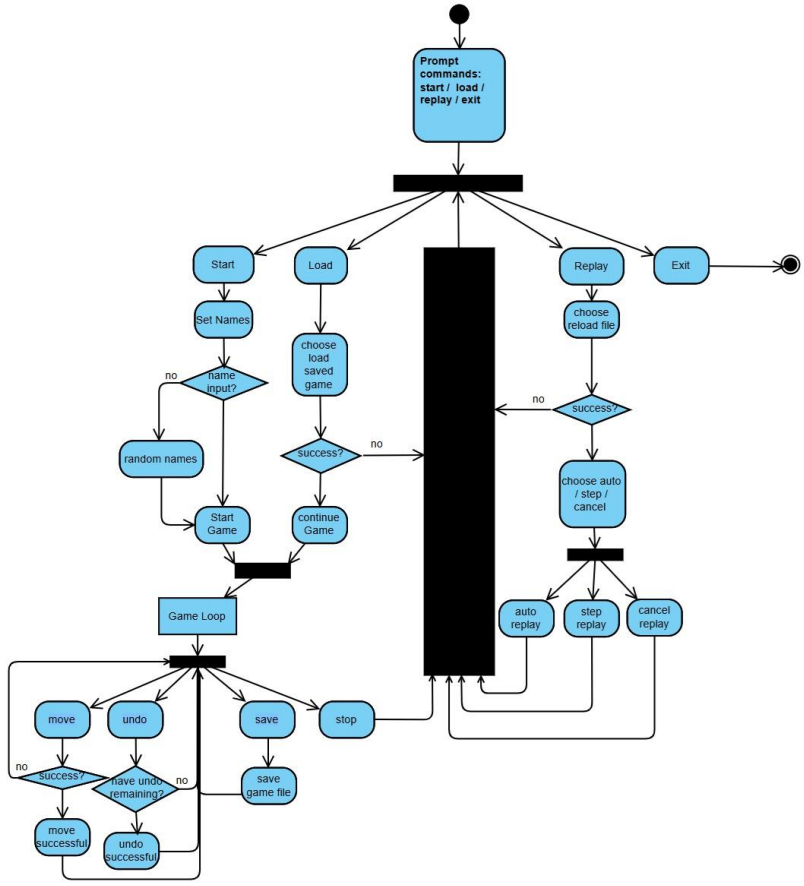
8. End: report final status or winner

Cross-cutting / persistence / logging

- logEvent(String) is called by GameStart (when not review), movePiece, undo to append lines to the current record file; review mode (set by GameStart(true)) prevents logging during replay.
- initializeBoardHistory() and boardHistory[] are used by movePiece() and undo() to implement the move/undo behavior.

Termination

- From main(): choosing "exit" closes scanner and returns. From start/load play loops: "stop" returns to the main menu. When model sets gameOver (via checkWinCondition()), the UI prints winner and returns to main menu.



3.2 Logistics of “undo” in the Game

The following diagram shows the logistics of “undo” in the game. Let red player be the player requests to undo his/her move. If the most recent player wishes to take back his/her move, the request should be entered under blue’s turn, i.e. when blue is the current active player. When the system received “undo” input from the current active player (blue), it interprets as the previous player undo his/her move (the red player). The method returns the most recent state before red moves. The remaining undo count for red player would reduce by one, and Red player becomes the current active player.

Similarly, if there are “undo” inputs received consecutively, the system would repeat the undo function until the player run out of undo counts or there are no movements to undo. Here is an example of how “undo” works: