COMP3211: Software Engineering
Design Document - *Jungle Game*

Student Name	Student ID	Contribution
Sun Hansong	23097775d	33.3%
Fu Guanhe	23097455d	33.3%
Wang Yuqi	23110134d	33.3%

1. Introduction

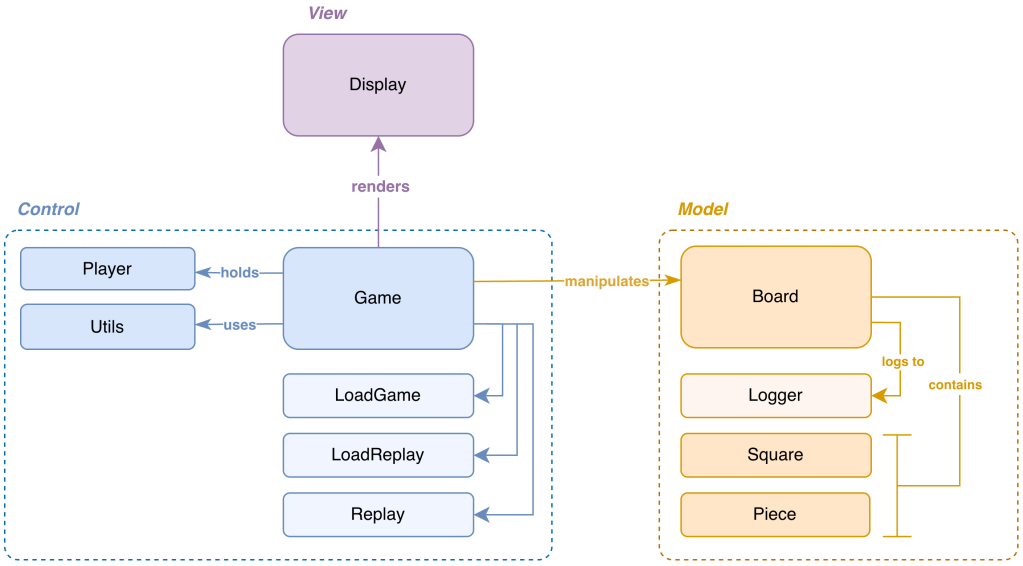

Figure 1: Our code structure demonstrating extensibility and modularity. The diagram illustrates the high-level separation of responsibility in our *Jungle Game* implementation. The **Control layer (Blue)** contains the Game controller that orchestrates gameplay, alongside utility components like LoadGame, LoadReplay, and Replay that handle different execution modes. **View layer (Purple)** consists solely of the stateless Display component, with a unidirectional rendering connection. **Model layer (Yellow)** contains domain logic within Board, Logger, Square, and Piece classes. Arrows denote control flow.

1.1. Main Architectual Pattern

Our implementation for the *Jungle Game* adopts **Model-View-Controller (MVC)** pattern. This pattern separates the application into three interconnected components. The MVC pattern was selected because the game requires multiple way to view game state (normal gameplay mode vs. replay mode). Its components are instantiated as follows in the *Jungle Game* context:

Model Within the `Model` package. This layer contains all domain logic and game state. The `Board` class acts as the central component, maintaining a 7×9 grid of `Square` objects, each potentially containing a `Piece` object. The `MovingValidator` enforces movement rules specific to the *Jungle Game*, such as terrain constraints (river, trap, den), piece capture mechanisms, and specific abilities (rat swimming, tiger/lion leaping); at each round, the `Board` query the validator with specific move attempts. The `Logger` component records all game events as reversible operations, allowing for withdraw and replay.

View Implemented through the `Display` class. It renders the board state to the console using Unicode box-drawing characters. The view is stateless and purely reactive; in other words, it accepts a `Board` reference and produces formatted output without maintaining any game states. The `Display` class have a few methods worth mentioning. The `displayBoard` method traverses the board grid and foramt each square according to tis type (normal, river, trap, den) and occupying piece. The players are distinguished by uppercase vs. lowercase letters.

Controller The `Game` class functions as the primary controller. It orchestrates the game loop and mediates user input and model operations. The `Game` class has a reference to the `Board` model and two `Player` objects. Upon receiving user commands, the `Game` object delegates the validation and execution to the model layer, then triggers view update after successful state modifications by the model.

1.2. Complementary Architectual Pattern

Beyond MVC, our implementation also incorporates a **Repository Pattern** via the `Logger` class. The logger maintains a centralized event storage that all game state modification flows through. The

serialization mechnaism for saving game is effectively a simple file-based repository, persisting the entire object graph to a `.jungle` file.

2. Model

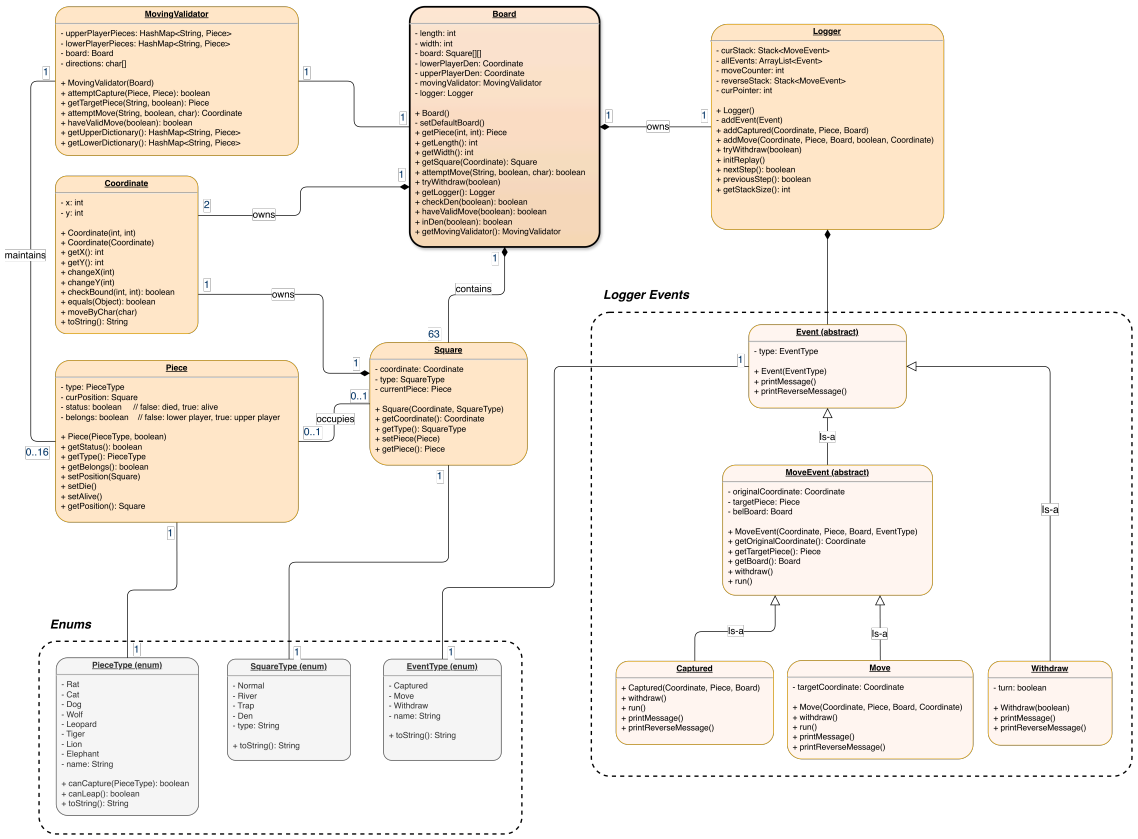

Figure 2: UML diagram depicts finegrained internal design of the Model layer. The principal classes are colored vivid orange, while secondary classes are colored light orange or grey. **Top Left:** The `Board` is the root of the structure; it composes a grid of Squares and delegates movement to the `MovingValidator` class. The moving validators use `HashMaps` to maintain player pieces, allowing $O(1)$ lookup. **Right:** The `Logger` stores the history via a unified `Event` abstract class hierarchy of `Move`, `Captured`, and `Withdraw` events. The `Piece` and `Square` classes has a bidirectional association: each piece references its containing square, while each square optionally holds one piece; this allows efficient movement queries and positional updates. **Bottom Left:** Emumeration classes (`PieceType`, `SquareType`, `EventType`) encodes domain-specific constants (e.g., `Rat`, `Cat`, `River`), as well as logic via methods like `canCapture()` and `canLeap()`.

As illustrated in Figure 2, our model component handles the entirety of the game's domain concepts and business rules. The `Board` class serves as the aggregate **root of the model**, exposing coarse-grained operations like `attemptMove()` and `tryWithdraw()` to the controller upstream. Internally, the board delegates fine-grained game logic to the `MovingValidator`, such as rats can swim in water but cannot capture when exiting water, etc. These rules are encoded through a combination of `PieceType` enumeration methods which the `MovingValidator` queries to determine context-specific legality.

The `Logger` component introduces temporal dimension to our model. It records a complete audit train of events. Each event contains self-sufficient information to execute forward (when replay) and reverse (when withdraw). As illustrated in the figure, the events are implemented as three: `Captured` event, `Move` event, and `Withdraw` event. The move event stores the previous and next coordinate, along with piece reference; captured events additionally stores the captured pieces. Then, withdraw mechanisms can be implemented as a simple popping and reversing events from a stack, the replay simply a sequential list of events. In the future, this highly versatile design can be further extended for features like game analysis and AI training dataset.

3. View

The view layer is strictly reactive-only, never initiating state changes or referencing beyond the scope of its role. The `Display` class exposes a **single primary operation**: `displayBoard(Board)`. This method accepts a board reference, traverses its matrix, and produce formatted output. By having the view layer designed like a pure function of current model state, no synchronization bugs can occur (e.g., due to stale view state), and the same `Display` methods work indentically regardless of whether it is an active game, replay mode, or loading from a unfinished saved game.

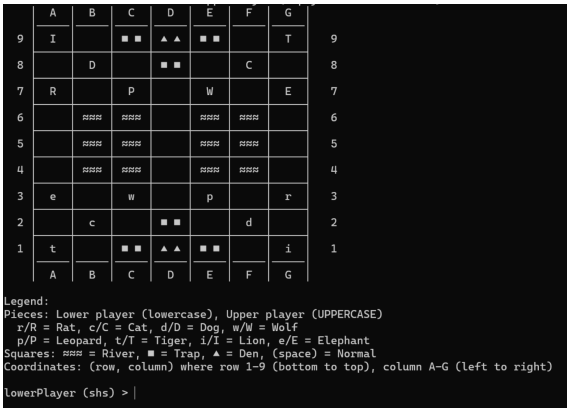

Figure 3: Screenshot of console output of the board render by the view module

The rendering algorithm produces a visually clear and aesthetic output. The board is displayed with both row and column coordinate legends (letters A-G for x-axis, numbers 1-9 for y-axis). Pieces are distinguished by case (uppercase for upper player, lowercase for lower) rather than color, ensuring compatibility with older terminal environments that lack color support.

The view also never prints error messages, prompts, or game status updates. These remains solely the controller's responsibility. This design demonstrates our clear understanding of high cohesion low coupling design. The only coupling between view and model is the read-only query methods on Board, Square, and Piece classes, which `Display` method traverses but **never** mutates.

4. Control

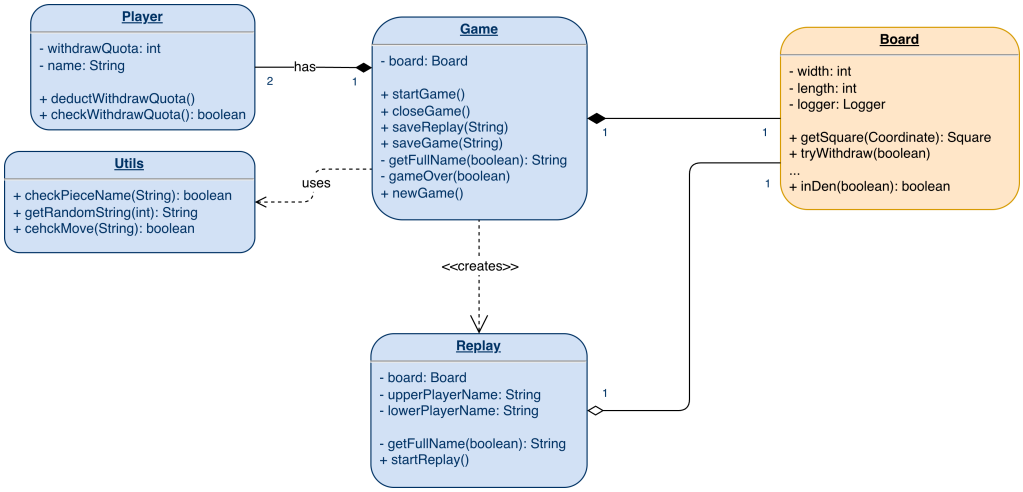

Figure 4: The `Game` class is the central controller, maintaining composition relationships with two `Player` instances and one `Board` instance. The `Game` class also depends on the `Utils` class for input validation and the `Display` for rendering. When provided the `saveReplay` command, a `Replay` object is instantiated by `Game`, and this `Replay` object holds a reference to a `Board` object.

The controller is the bridge between the player's command and the game's internal logics. The `Game` class implements the central game loop: display the board, check for end conditions, prompt the current

player, parse input, delegate to model operations, update turn state, and render the result (see Figure 5 for more detail).

Input parsing and validation occurs in stages. Early validation acts as sanity checks to prevent unnecessary model calls, such as whether the command mentioned a valid known piece still on the board, or directional characters (U/D/L/R) before invoking `Board.attemptMove()`. However, these are only synactic validation, true semantic validation (is piece alive? does player own it?) remains entirely within the `Model`. This clear separation of responsibility prevents business logic of the model leaking into the controller.

5. Collaboration of Components (*via sequence diagram*)

5.1. Pre-Turn Rule Checks

The sequence diagram captures the runtime collaboration among objects during a players' turn, showcasing message-passing logic that implements MVC pattern's separation of concerns. The logic begins with the game controller in a preparatory phase via methods `Board.checkDen()` and `Board.haveValidMove()` to determine whether the game should be continued or not in the first place. These prevent unnecessary user prompts when the game as already ended. Only after confirming the game is ongoing does the controller display the player prompt and block on input.

5.2. Read, Validate, and Execute Commands

Upon receving a move command (e.g., "rat U"), the Game performs lightweight *syntactic validation* (e.g., whether "rat" matches a piece name and "U" is a valid direction). Once that passes, the controller invokes `Board.attemptMove("rat", currentTurn, "U")`, which the board immediately delegates to `MovingValidator.attemptMove()`. A bulk of validation logic is then performed: piece lookup in the corresponding player's `HashMap`, confirming alive, computing destination coordinates, checking boundary, conditions, evaluating movement rules specific to the piece type and surrounding terrain. If all aforementioned *semantic validation* passes, it goes through a final *capture validation* via `MovingValidator.attemptCapture()`, where it checks whether the destination position is already occupied by an opponent piece, and whether the capture is allowed.

- **If Failed**: upon any rule violations, the validator throws `IllegalArgumentException` with a descriptive message, which then propagates upwards through the `Board` (model) → `Game` (controller), caught, printed, and turn retried.
- **If Succeeded**: when validation succeeds, control returns the board with the validated destination coordinates. The board then checks whether the destination square is occupied and record a `Captured` event (see Figure 2) via `Logger.addCaptured()`. Then, the board records a `Move` event via `Logger.addMove()` regardless of whether captured. These events are added to the logger's internal data structures: stack (for potential withdraw) and list (for replay)

After the model is successfully updated, control returns to the `Game` controller, which receives a boolean `True` from `attemptMove()`. The controller then switch player by toggling the `currentTurn` flag and invoking `Display.displayBoard()` to render the updated chess board. Within, the `Display` queries the `Board` for each square's type and occupancy, formats and prints them.

Alternatively, if the player issued a `withdraw` command, the Game first queries `Player.checkWithdrawQuota()` to confirm that the player has withdraws remaining. If so (i.e., `quota ok`), the controller calls `Board.tryWithdraw()`, which delegates to `Logger.tryWithdraw()`. The logger pops last two moves (collectively a turn), reversing the game state.

5.3. Repeat Game Loop

Up to this point, the game has completed a full **pre-turn check → command input & validation → command execution** loop. The controller then returns to the top of the game loop where pre-turn validation begins again for the next player. This repeats until the `close` command is issued, which the loop terminates.

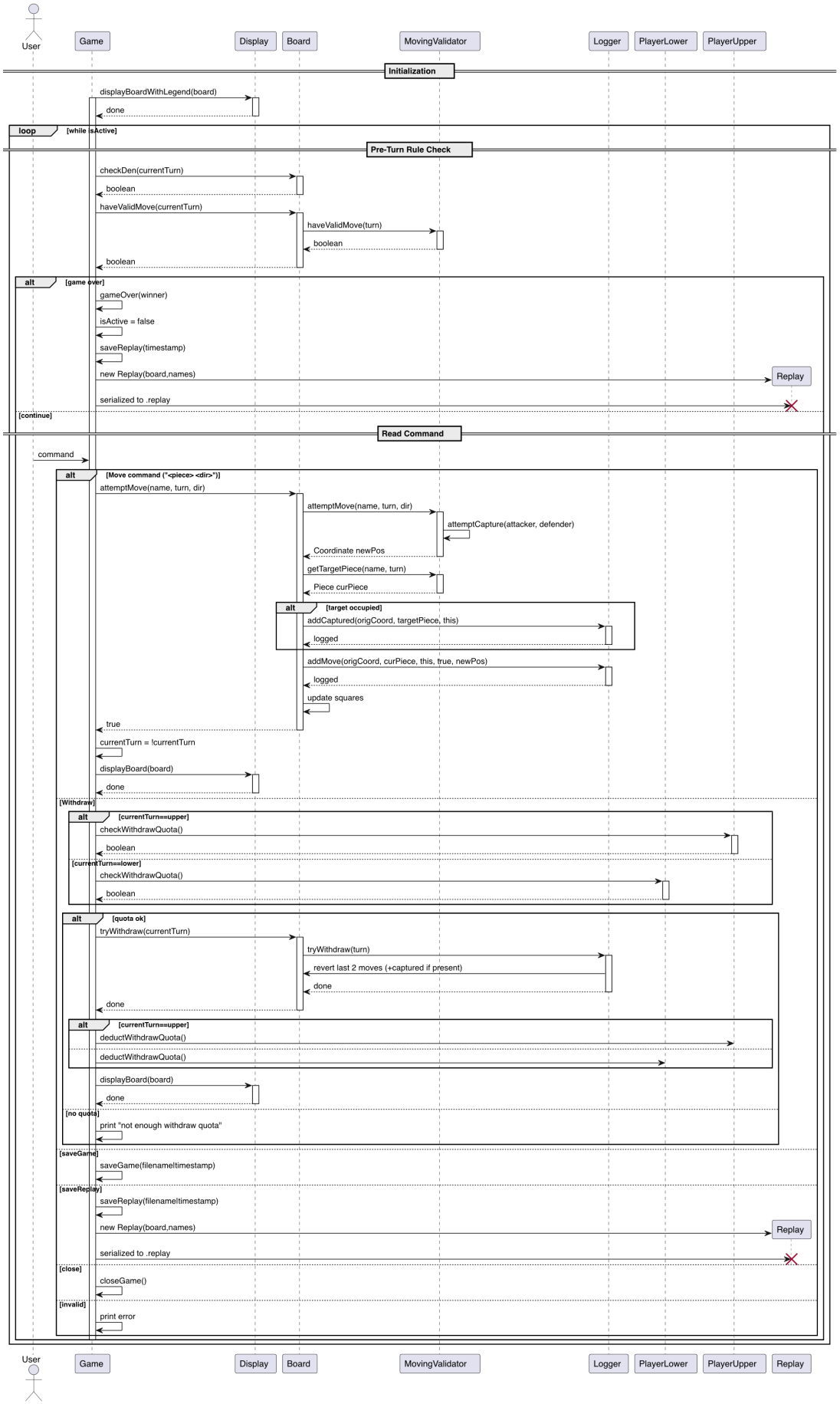

Figure 5: Sequence diagram tracing the complete lifecycle of the gameplay loop. It reveals three phases: **(1)** pre-turn validation, **(2)** move execution, and **(3)** post-move rendering phase.