Module Interface Specification for Software Engineering

Team 8, RLCatan
Rebecca Di Filippo
Jake Read
Matthew Cheung
Sunny Yao

November 13, 2025

1 Revision History

Date	Version	Notes
11/03/2025	1.0	Draft Rev 1

2 Symbols, Abbreviations and Acronyms

See SRS Documentation at https://github.com/SY3141/RLCatan.

Contents

3 Introduction

The following document details the Module Interface Specifications for our project RLCatan. This project aims to create a competent reinforcement learning AI agent designed to master the board game Settlers of Catan through autonomous self-play training. The AI will use deep reinforcement learning algorithms to learn optimal decision-making strategies across several game states including resource management, territory expansion and adaptive responses to opponent actions.

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at https://github.com/SY3141/RLCatan.

4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol := is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1|c_2 \Rightarrow r_2|...|c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by Software Engineering.

Type/Symbol	Notation	Description
boolean	$\mathbb{B}$	a logical value, either true ($\top$) or false ($\bot$)
character	char	a single symbol or digit
integer	$\mathbb{Z}$	a number without a fractional component in (-∞, ∞)
natural number	$\mathbb{N}$	a number without a fractional component in [1, ∞)
real	$\mathbb{R}$	any number in (-∞, ∞)
string	Σ^*	A sequence of zero or more characters.
sequence	seq of T	A list of zero or more elements of the same type T.
tuple	(T1, T2, ...)	A finite, ordered list of elements, potentially of different types.
dictionary	dict	A collection of key-value mappings (e.g., KeyType $\rightarrow$ ValueType).
function	T1 $\rightarrow$ T2	A function mapping an input of type T1 to an output of type T2.
true	$\top$	The boolean value for true.
false / null	$\bot$	The boolean value for false, or a value that is undefined, null, or void (no output).

In addition, Software Engineering uses abstract data types (e.g., `FrameData`, `GameStateData`, `AIMove`) which are defined by their use in the modules that hide their implementation details. Local functions are described by giving their type signature followed by their specification.

5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

Level 1	**Level 2**
Hardware-Hiding Module	Hardware-Hiding Module (OS) Computer Vision Model
Behaviour-Hiding Module	User Interface Game State Manager Reinforcement Learning Environment
Software Decision Module	AI Model Game State Database Image Queue

6 MIS of Hardware-Hiding Module

6.1 Module

Hardware-Hiding Module (M1)

6.2 Uses

- **FrameData (Data Type):** Represents the raw image data captured from the camera or a simulated source.
- **HardwareInitError, CaptureError, HardwareShutdownError (Exception Types):** Indicate errors that may occur during hardware initialization, frame capture, or shutdown processes.

6.3 Syntax

6.3.1 Exported Constants

- `CAMERA_RESOLUTION`: tuple of (int, int) - width and height of captured frames
- `FRAME_RATE`: $\mathbb{R}$ - frames per second captured by the hardware layer

6.3.2 Exported Access Programs

Name	In	Out	Exceptions
initializeHardware	-	-	HardwareInitError
captureFrame	-	FrameData	CaptureError
shutdownHardware	-	-	HardwareShutdownError
isInitialized	-	$\mathbb{B}$	-

6.4 Semantics

6.4.1 State Variables

- `hardwareStatus`: $\mathbb{B}$
  Indicates whether the hardware subsystem has been successfully initialized.
- `frameBuffer`: FrameData
  Stores the most recent frame captured from the camera.

6.4.2 Environment Variables

- `CameraDevice`: external hardware input for image capture
- `DisplayInterface`: external interface for visualization

6.4.3 Assumptions

- The CameraDevice is available and functional within the operating environment.
- Calling modules will invoke `isInitialized()` before executing `captureFrame()` or `shutdownHardware()`.

6.4.4 Access Routine Semantics

initializeHardware():

- transition: Attempts connection to `CameraDevice`. If successful: `hardwareStatus` := true.
- output: -
- exception: raises `HardwareInitError` if initialization fails.

captureFrame():

- transition: Reads a new frame from `CameraDevice` into `frameBuffer`.
- output: Returns the current `frameBuffer` of type FrameData.
- exception: raises `CaptureError` if `hardwareStatus` = false or frame capture fails.

shutdownHardware():

- transition: Releases device resources and sets `hardwareStatus` := false.
- output: -
- exception: raises `HardwareShutdownError` if hardware cleanup fails.

isInitialized():

- transition: -
- output: returns `hardwareStatus`: $\mathbb{B}$
- exception: -

6.4.5 Local Functions

- `checkDeviceConnection()`: verifies device presence
- `allocateBufferMemory()`: allocates buffer for captured frames
- `releaseResources()`: frees hardware-related resources

7 MIS of Computer Vision Module

7.1 Module

Computer Vision Module (M2)

7.2 Uses

- **M1.captureFrame():** Provides a single frame of image data from the hardware or simulated camera.
- **FrameData (Data Type):** Represents the raw image frame used for analysis.
- **GameStateData (Data Type):** Abstract representation of detected game state.
- **ProcessingError, DetectionError (Exception Types):** Raised during processing or detection failures.
- **seq of Elements, dict** (Derived Data Types): Sequences and mappings used to store detected features and confidence metrics.

7.3 Syntax

7.3.1 Exported Constants

- `DETECTION_THRESHOLD`: $\mathbb{R}$ - minimum required confidence for object recognition
- `MODEL_PATH`: Σ^* - file system path to trained CV model

7.3.2 Exported Access Programs

Name	In	Out	Exceptions
processFrame	frame:FrameData	GameStateData	ProcessingError
detectBoardElements	frame:FrameData	seq of Elements	DetectionError
getConfidenceMetrics	-	dict	-

7.4 Semantics

7.4.1 State Variables

- `lastFrame`: FrameData
  Stores the most recent processed frame.
- `lastConfidence`: dict
  A mapping from element identifiers (Σ^*) to confidence scores ($\mathbb{R}$).

7.4.2 Environment Variables

None.

7.4.3 Assumptions

- All `FrameData` inputs are valid and camera-calibrated.
- The model at `MODEL_PATH` is present and loaded into memory before use.

7.4.4 Access Routine Semantics

processFrame(frame):

- transition: Executes the full CV pipeline on $frame$; updates `lastFrame` := frame.
- output: Returns detected game state as `GameStateData`.
- exception: raises `ProcessingError` if any stage of the pipeline fails.

detectBoardElements(frame):

- transition: Runs object detection model on $frame$; updates `lastFrame` := frame and updates `lastConfidence`.
- output: Returns seq of detected Elements.
- exception: raises `DetectionError` if detection fails.

getConfidenceMetrics():

- transition: -
- output: Returns the `lastConfidence` mapping (a dict from Σ^* to $\mathbb{R}$).
- exception: -

7.4.5 Local Functions

- `calibrateCamera()`: Corrects lens distortion in input frames.
- `filterNoise()`: Removes low-confidence or spurious detections.
- `parseElementsToState()`: Converts detected Elements into `GameStateData`.

8 MIS of User Interface Module

8.1 Module

User Interface (M3)

8.2 Uses

- **GameStateData**: Abstract representation of the current game state.
- **AIMove**: Encoded AI-generated move.
- **seq of Corrections**: A sequence of user-provided corrections to the game state.
- **RenderError**: Exception raised when visualization or rendering fails.

8.3 Syntax

8.3.1 Exported Constants

- `REFRESH_RATE`: $\mathbb{R}$ - frequency of UI updates (Hz)

8.3.2 Exported Access Programs

Name	In	Out	Exceptions
renderBoard	state:GameStateData	-	RenderError
displayAIMove	move:AIMove	-	-
getUserCorrections	-	seq of Corrections	-

8.4 Semantics

8.4.1 State Variables

- `uiState`: Abstract UI representation (type unspecified, internal)
- `correctionQueue`: seq of Corrections A sequence of user-provided corrections where each correction may include:
  - element identifier: Σ^*
  - correction value: arbitrary derived type

8.4.2 Environment Variables

- `window`: graphical display device managed by the environment

8.4.3 Assumptions

- The `window` device is functional and compatible with the UI rendering framework.
- User interactions are delivered to the system via environment-level event handling.

8.4.4 Access Routine Semantics

renderBoard(state):

- transition: Updates `uiState` := graphical representation of *state*; renders `uiState` onto `window`.
- output: -
- exception: raises `RenderError` if rendering fails.

displayAIMove(move):

- transition: Updates `uiState` to include the visualization of *move*; refreshes `window`.
- output: -
- exception: -

getUserCorrections():

- transition: `tmp` := `correctionQueue`; `correctionQueue` := empty sequence.
- output: Returns `tmp`: seq of Corrections.
- exception: -

8.4.5 Local Functions

- `updateDOM()`: Updates internal UI representation elements.
- `highlightElements()`: Highlights tiles, pieces, or game move indicators.
- `onUserInput()`: Adds user-generated corrections to `correctionQueue`.

9 MIS of Game State Manager Module

9.1 Module

Game State Manager (M4)

9.2 Uses

- **MoveData**: Represents the details of a player's move, including action type and parameters.
- **GameStateData**: Abstract representation of the complete game state.
- **InvalidMoveError**: Raised when an illegal move is applied.
- **M7.writeState()**: Persists updated game state to storage.

9.3 Syntax

9.3.1 Exported Constants

- `MAX_PLAYERS`: $\mathbb{Z} = 4$

9.3.2 Exported Access Programs

Name	In	Out	Exceptions
updateState	move:MoveData	-	InvalidMoveError
getState	-	GameStateData	-
validateMove	move:MoveData	$\mathbb{B}$	-

9.4 Semantics

9.4.1 State Variables

- `currentGameState`: GameStateData
  The full internal representation of the game board.

- `playerAssets`: dict
  Mapping from player identifiers (Σ^*) to their resource, structure, and score information.
  Each value may include:

  - resources: dict from resource type (Σ^*) to quantity ($\mathbb{Z}$)
  - structures: seq of structure types
  - score: $\mathbb{Z}$

9.4.2 Environment Variables

None.

9.4.3 Assumptions

- All instances of `MoveData` conform to the expected internal format.
- Calling modules will evaluate `validateMove(move)` before invoking `updateState(move)`.

9.4.4 Access Routine Semantics

updateState(move):

- transition:

  - If `validateMove(move)` = true, apply changes encoded in *move* to `currentGameState` and `playerAssets`.

– Otherwise, no state change.

- output: -
- exception: raises `InvalidMoveError` if `validateMove(move)` = false.

getState():

- transition: -
- output: Returns a copy of `currentGameState`: GameStateData.
- exception: -

validateMove(move):

- transition: -
- output: $\mathbb{B}$ - true iff *move* is valid according to Catan rules and the current values of `currentGameState` and `playerAssets`.
- exception: -

9.4.5 Local Functions

- `calculateResources()`: Computes resource changes resulting from a move.
- `updateScores()`: Updates player score values based on new state.
- `checkVictory()`: Determines whether any player meets winning conditions.

10 MIS of Reinforcement Learning Environment Module

10.1 Module

Reinforcement Learning Environment (M5)

10.2 Uses

- **M4**: Uses the access programs `updateState`, `getState`, and `validateMove`.
- **AIMove**: Encoded AI-selected action.
- **GameStateData**: Abstract representation of a full game state.
- **Reward**: A real-valued score, $\mathbb{R}$.
- **StepError, RenderError**: Exceptions raised on invalid steps or rendering failures.

10.3 Syntax

10.3.1 Exported Constants

- `MAX_TURNS`: $\mathbb{Z}$ - maximum number of turns in an episode
- `REWARD_SCALE`: $\mathbb{R}$ - scaling factor applied to reward calculation

10.3.2 Exported Access Programs

Name	In	Out	Exceptions
step	action:AIMove	(GameStateData, $\mathbb{R}$)	StepError
reset	-	GameStateData	-
render	-	-	RenderError

10.4 Semantics

10.4.1 State Variables

- `simulatedState`: internal instance of M4
  Maintains the environment's evolving game state.
- `turnCount`: $\mathbb{Z}$
  Number of turns elapsed in the current episode.

10.4.2 Environment Variables

- `display`: Rendering surface or graphical context used to visualize simulatedState.

10.4.3 Assumptions

- All inputs of type `AIMove` are syntactically valid.
- The environment is responsible for handling opponent moves internally.

10.4.4 Access Routine Semantics

step(action):

- transition:
  - If `validateMove(action)` from M4 = true, apply *action* to `simulatedState` via `updateState(action)`.
  - Apply `applyOpponentLogic()` to simulate adversarial moves.
  - turnCount := turnCount + 1.

- output: Returns the tuple (`simulatedState.getState()`, `calculateReward()`) of type ($GameStateData, \mathbb{R}$).
- exception: raises `StepError` if `validateMove(action)` = false.

reset():

- transition: `simulatedState` := new initial game state (via M4). turnCount := 0.
- output: Returns initial game state: GameStateData.
- exception: -

render():

- transition: Updates the `display` device to visualize the current `simulatedState`.
- output: -
- exception: raises `RenderError` if display rendering fails.

10.4.5 Local Functions

- `applyOpponentLogic()`: Generates and applies opponent actions based on heuristics or baseline policies.
- `calculateReward()`: Returns a reward value in $\mathbb{R}$ based on changes in `simulatedState`, scaled by `REWARD_SCALE`.

11 MIS of AI Model Module

11.1 Module

AI Model (M6)

11.2 Uses

- **GameStateData**: Abstract representation of the game state.
- **AIMove**: Encoded move predicted by the model.
- **PredictionError**: Exception raised when the model cannot generate a valid move.

11.3 Syntax

11.3.1 Exported Constants

- `MODEL_PATH`: Σ^* - file path to stored model weights.

11.3.2 Exported Access Programs

Name	In	Out	Exceptions
decide	state:GameStateData	AIMove	PredictionError
getMoveConfidence	move:AIMove	$\mathbb{R}$	-
explainMove	move:AIMove	Σ^*	-

11.4 Semantics

11.4.1 State Variables

- `modelWeights`: internal representation of learned parameters (vector or tensor values in $\mathbb{R}$ or $\mathbb{R}^n$).
- `policyNetwork`: function mapping processed state input to a probability distribution over candidate moves:

$$\texttt{policyNetwork} : \texttt{ProcessedState} \rightarrow \text{dist(AIMove)}$$

- `lastPredictionCache`: dict mapping AIMove $\rightarrow \mathbb{R}$ storing confidence values for the most recent decision.

11.4.2 Environment Variables

None.

11.4.3 Assumptions

- The input `GameStateData` conforms to the expected internal structure.
- `MODEL_PATH` correctly references initialized modelWeights.

11.4.4 Access Routine Semantics

decide(state):

- transition:
  - Converts *state* using `preprocessState()`.
  - Evaluates result with `policyNetwork`.
  - Updates `lastPredictionCache` with move-confidence pairs.
- output: Returns the optimal AIMove predicted by the model.
- exception: raises `PredictionError` if the model evaluation fails or produces an empty distribution.

getMoveConfidence(move):

- transition: -
- output: Returns $\mathbb{R}$ - the confidence score associated with *move* retrieved from `lastPredictionCache`.
- exception: -

explainMove(move):

- transition: -
- output: Returns Σ^* - a human-readable textual explanation of the rationale behind selecting *move* (e.g., feature contributions or decision breakdown).
- exception: -

11.4.5 Local Functions

- `preprocessState()`: Converts GameStateData into an abstract processed representation suitable for model inference.
- `postprocessOutput()`: Converts model policy output into a structured AIMove.

12 MIS of Game State Database Module

12.1 Module

Game State Database (M7)

12.2 Uses

- **GameStateData**: Abstract representation of a game state.
- **DBWriteError**: Exception raised on failed write operations.
- **DBReadError**: Exception raised on failed read or query operations.
- **seq(GameStateData)**: Ordered collection of game states.

12.3 Syntax

12.3.1 Exported Constants

- `DB_PATH`: Σ^* - file system path or database URI.
- `MAX_ENTRIES`: $\mathbb{Z}$ - maximum number of stored states.

12.3.2 Exported Access Programs

Name	In	Out	Exceptions
writeState	state:GameStateData	-	DBWriteError
readState	gameID:Σ^*	GameStateData	DBReadError
queryHistory	playerID:Σ^*	seq(GameStateData)	DBReadError

12.4 Semantics

12.4.1 State Variables

- `dbConnection`: Active connection handle to the database (abstract type).
- `gameRecords`: dict mapping $\Sigma^* \rightarrow$ GameStateData (cache of recently accessed or frequently used game states).

12.4.2 Environment Variables

- `database`: External file system or DB server located at `DB_PATH`.

12.4.3 Assumptions

- The database at `DB_PATH` is reachable and adheres to the expected schema.
- Identifiers such as `gameID` and `playerID` are valid Σ^* strings.

12.4.4 Access Routine Semantics

writeState(state):

- transition:
  - Converts *state* to a storable format via `serializeState()`.
  - Writes serialized data to `database`.
  - Updates `gameRecords` cache if applicable.
- output: -
- exception: raises `DBWriteError` if the write operation fails.

readState(gameID):

- transition:
  - If $gameID$ is cached, return entry from `gameRecords`.
  - Else, load corresponding data from `database`.

  - Deserialize via `deserializeState()`.
  - Update `gameRecords`.
- output: GameStateData corresponding to the given $gameID$.
- exception: raises `DBReadError` if no entry exists for $gameID$ or if read fails.

queryHistory(playerID):

- transition:
  - Queries `database` for all entries linked to $playerID$.
  - Converts results to seq(GameStateData) via `deserializeState()`.
- output: seq(GameStateData) containing all states associated with $playerID$.
- exception: raises `DBReadError` if the query fails.

12.4.5 Local Functions

- `serializeState()`: Converts GameStateData into a storable representation.
- `deserializeState()`: Converts stored representation back to GameStateData.
- `openConnection()`: Initializes `dbConnection`.
- `closeConnection()`: Terminates `dbConnection`.

13 MIS of Image Queue Module

13.1 Module

Image Queue (M8)

13.2 Uses

- **FrameData**: Abstract representation of an image frame from the camera or simulation.
- **QueueFullError**: Exception raised when attempting to enqueue a frame into a full queue.
- **QueueEmptyError**: Exception raised when attempting to dequeue or peek from an empty queue.

13.3 Syntax

13.3.1 Exported Constants

- `QUEUE_SIZE` $\in \mathbb{Z}^+$ - maximum number of frames in the buffer.
- `TIMEOUT` $\in \mathbb{R}^+$ - maximum wait time (seconds) for enqueue/dequeue operations.

13.3.2 Exported Access Programs

Name	In	Out	Exceptions
enqueue	frame:FrameData	$\perp$	QueueFullError
dequeue	-	FrameData	QueueEmptyError
peek	-	FrameData	QueueEmptyError
isFull	-	$\mathbb{B}$	-
isEmpty	-	$\mathbb{B}$	-

13.4 Semantics

13.4.1 State Variables

- `queueBuffer` : $[0..\text{QUEUE\_SIZE} - 1] \to \text{FrameData} \cup \{\perp\}$ - circular buffer storing frames.
- `head` : $\mathbb{Z}$ - index of the front of the queue.
- `tail` : $\mathbb{Z}$ - index of the end of the queue.
- `size` : $\mathbb{Z}$ - current number of frames in the queue.

13.4.2 Environment Variables

None.

13.4.3 Assumptions

- Calling modules will check `isFull()` before `enqueue()`.
- Calling modules will check `isEmpty()` before `dequeue()` or `peek()`.

13.4.4 Access Routine Semantics

enqueue(frame: FrameData):

- transition:

$$\text{queueBuffer}[\text{tail}] := \text{frame}, \quad \text{tail} := (\text{tail} + 1) \bmod \text{QUEUE\_SIZE}, \quad \text{size} := \text{size} + 1$$

- output: $\top$ if successful
- exception: raises `QueueFullError` if isFull() = $\top$.

dequeue():

- transition: frame := queueBuffer[head]; queueBuffer[head] := $\bot$; head := (head+1) mod QUEUE_SIZE; size := size-1
- output: frame
- exception: QueueEmptyError if isEmpty() = True

peek():

- transition: -
- output: queueBuffer[head]
- exception: raises `QueueEmptyError` if isEmpty() = $\top$.

isFull():

- transition: -
- output: $\top$ if size = QUEUE_SIZE, $\bot$ otherwise
- exception: -

isEmpty():

- transition: -
- output: $\top$ if size = 0, $\bot$ otherwise
- exception: -

13.4.5 Local Functions

- `resetQueue()`:

  $$\text{head} := 0, \quad \text{tail} := 0, \quad \text{size} := 0$$

  Resets the queue to an empty state.

References

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL http://citeseer.ist.psu.edu/428727.html.

14 Appendix