Module Interface Specification for Software Engineering

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

November 12, 2025

1 Revision History

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

2 Symbols, Abbreviations and Acronyms

See SRS Documentation at [give url —SS]

[Also add any additional symbols, abbreviations or acronyms —SS]

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

[You should describe your notation. You can use what is below as a starting point. —SS]

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.

Data Type	Notation	Description
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 (-∞, ∞)

The specification of Software Engineering uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, Software Engineering uses functions, which are defined by the data types of their inputs and outputs. 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 [Module Name —SS]

[Use labels for cross-referencing —SS]

[You can reference SRS labels, such as R??. —SS]

[It is also possible to use LaTeXfor hypperlinks to external documents. —SS]

6.1 Module

[Short name for the module —SS]

6.2 Uses

6.3 Syntax

6.3.1 Exported Constants

6.3.2 Exported Access Programs

Name	In	Out	Exceptions
[accessProg —SS]	-	-	-

6.4 Semantics

6.4.1 State Variables

[Not all modules will have state variables. State variables give the module a memory. —SS]

6.4.2 Environment Variables

[This section is not necessary for all modules. Its purpose is to capture when the module has external interaction with the environment, such as for a device driver, screen interface, keyboard, file, etc. —SS]

6.4.3 Assumptions

[Try to minimize assumptions and anticipate programmer errors via exceptions, but for practical purposes assumptions are sometimes appropriate. —SS]

6.4.4 Access Routine Semantics

[accessProg —SS]():

- transition: [if appropriate —SS]
- output: [if appropriate —SS]

- exception: [if appropriate —SS]

[A module without environment variables or state variables is unlikely to have a state transition. In this case a state transition can only occur if the module is changing the state of another module. —SS]

[Modules rarely have both a transition and an output. In most cases you will have one or the other. —SS]

6.4.5 Local Functions

[As appropriate —SS] [These functions are for the purpose of specification. They are not necessarily something that is going to be implemented explicitly. Even if they are implemented, they are not exported; they only have local scope. —SS]

7 MIS of Hardware-Hiding Module

[Use labels for cross-referencing —SS]

[You can reference SRS labels, such as R**??**. —SS]

7.1 Module

Hardware-Hiding Module (M1)

7.2 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.

7.3 Syntax

7.3.1 Exported Constants

- `CAMERA_RESOLUTION` - resolution of captured frames
- `FRAME_RATE` - frames per second captured by the hardware layer

7.3.2 Exported Access Programs

Name	In	Out	Exceptions
initializeHardware	-	-	HardwareInitError
captureFrame	-	FrameData	CaptureError
shutdownHardware	-	-	HardwareShutdownError
isInitialized	-	-	-

7.4 Semantics

7.4.1 State Variables

- `hardwareStatus`: Boolean - indicates if hardware is initialized
- `frameBuffer`: FrameData - stores the latest captured frame

7.4.2 Environment Variables

- `CameraDevice` - physical or virtual camera input
- `DisplayInterface` - Screen interface for visualization

7.4.3 Assumptions

- The CameraDevice is available and OS drivers are functional.
- Calling modules will check `isInitialized()` before invoking `captureFrame()` or `shutdownHardware`

7.4.4 Access Routine Semantics

initializeHardware():

- transition: Attempts to connect to . If successful, sets hardwareStatus to True
- output: -
- exception: raises `HardwareInitError` if initialization fails

captureFrame():

- transition: reads a new frame from CameraDevice into frameBuffer
- output: returns a copy of frameBuffer
- exception: raises `CaptureError` if hardwareStatus if False or frame capture fails

shutdownHardware():

- transition: sets `hardwareStatus` to False and releases CameraDevice and resources
- output: -
- exception: raises `HardwareShutdownError` if cleanup fails

isInitalized():

- transition:-
- output: returns the current value of hardwareStatus
- exception: -

7.4.5 Local Functions

- `checkDeviceConnection()`: ensures camera is available
- `allocateBufferMemory()`: manages memory for frame storage
- `releaseResources()`: frees hardware resources on shutdown

8 MIS of Computer Vision Module

8.1 Module

Computer Vision Module (M2)

8.2 Uses

- **M1.captureFrame():** Provides a single frame of image data from the hardware or simulated camera.
- **FrameData (Data Type):** Represents the raw image frame input used for further analysis.
- **GameStateData (Data Type):** Stores and tracks the current game state derived from processed images.
- **ProcessingError, DetectionError (Exception Types):** Raised during image analysis or element detection failures.
- **List[Elements], Dict (Data Types):** Standard data structures used to store and map detected objects and attributes.

8.3 Syntax

8.3.1 Exported Constants

- DETECTION_THRESHOLD - minimum confidence for object recognition
- MODEL_PATH - path to trained CV model (YOLOv9/OpenCV)

8.3.2 Exported Access Programs

Name	In	Out	Exceptions
processFrame	frame:FrameData	GameStateData	ProcessingError
detectBoardElements	frame:FrameData	List[Elements]	DetectionError
getConfidenceMetrics	-	Dict	-

8.4 Semantics

8.4.1 State Variables

- `lastFrame`: FrameData - last processed frame
- `lastCofidence`: Dict - confidence scores for last detections

8.4.2 Environment Variables

N/A

8.4.3 Assumptions

- Assumes `FrameData` provided is valid and preprocessed for analysis.
- Assumes model files at `MODEL_PATH` are loaded correctly before use.

8.4.4 Access Routine Semantics

processFrame(frame):

- transition: Runs full CV pipeline on frame to extract game state. Updates LastFrame to frame.
- output: returns `GameStateData` object representing detected board state
- exception: raises `ProcessingError` if parsing fails

detectBoardElements(frame):

- transition: Runs detection model on frame. Updates lastFrame to frame and updates lastConfidence.

- output: returns list of detected board elements (tiles, pieces, numbers)
- exception: raises `DetectionError` if detection fails

getConfidenceMetrics():

- transition: -
- output: returns confidence scores for last processed frame
- exception: -

8.4.5 Local Functions

- `calibrateCamera()`: adjusts image for lens distortion
- `filterNoise()`: removes spurious detections
- `parseElementsToState()`: converts detected elements to GameStateData

9 MIS of User Interface Module

9.1 Module

User Interface (M3)

9.2 Uses

- **GameStateData** (Data Type): Represents the current state of the game, including all player and board information.
- **AIMove** (Data Type): Defines an AI-generated move based on the current game state.
- **List[Corrections]** (Data Type): A list of correction objects used to adjust or validate game state data.
- **RenderError** (Exception Type): Raised when rendering or display operations fail.

9.3 Syntax

9.3.1 Exported Constants

- `REFRESH_RATE` - frequency of UI updates

9.3.2 Exported Access Programs

Name	In	Out	Exceptions
renderBoard	state:GameStateData	-	RenderError
displayAIMove	move:AIMove	-	-
getUserCorrections	-	List[Corrections]	-

9.4 Semantics

9.4.1 State Variables

- `uiState`: current UI rendering data
- `correctionQueue`: list of user inputted corrections

9.4.2 Environment Variables

- window: graphical display device

9.4.3 Assumptions

The window environment is compatible with the frontend framework.

9.4.4 Access Routine Semantics

renderBoard(state):

- transition: Modifies uiState to match state. Updates the window to visualize the new uiState.
- output: -
- exception: raises `RenderError` if rendering to the window fails

displayAIMove(move):

- transition: Modifies uiState to include a visualization of the move. Updates the window.
- output: -
- exception: -

getUserCorrections():

- transition: clears correctionQueue
- output: returns the current list of user corrections for misdetected board state
- exception: -

9.4.5 Local Functions

- `updateDOM()`: handles DOM updates in the window
- `highlightElements()`: highlights tiles, pieces, or moves
- `onUserInput()`: Internal event handler that adds user actions to correction Queue

10 MIS of Game State Manager Module

10.1 Module

Game State Manager (M4)

10.2 Uses

- **MoveData** (Data Type): Represents the details of a player's move, including action type and parameters.
- **GameStateData** (Data Type): Contains the full current state of the game used for validation and updates.
- **InvalidMoveError** (Exception Type): Raised when a move violates game rules or state constraints.
- **M7.writeState()** (from Game State Database Module): Writes the updated game state data to persistent storage.

10.3 Syntax

10.3.1 Exported Constants

- `MAX_PLAYERS` = 4

10.3.2 Exported Access Programs

Name	In	Out	Exceptions
updateState	move:MoveData	-	InvalidMoveError
getState	-	GameStateData	-
validateMove	move:MoveData	Boolean	-

10.4 Semantics

10.4.1 State Variables

- `currentGameState`: the full "Digital Twin" of the Catan board.
- `playerAssets`: A collection tracking each players resources,settlements,roads,etc.

10.4.2 Environment Variables

N/A

10.4.3 Assumptions

- Assumes `MoveData` is provided in the standard, non-corrupt format.
- Assumes the calling module will use `validateMove` before attempting `updateState`.

10.4.4 Access Routine Semantics

updateState(move):

- transition: iff validateMove(move) is True, the move is applied to currentGameState and playerAssets are updated.
- output: -
- exception: raises `InvalidMoveError` if move is False

getState():

- transition: -
- output: returns a copy of the current `currentGameState` and playerAssests as GameStateData
- exception: -

validateMove(move):

- transition: -
- output: Returns True if the move is a legal action according to the rules of Catan given the currentGameState, False otherwise.
- exception: -

10.4.5 Local Functions

- `calculateResources()` : computes resource changes from moves
- `updateScores()` : updates players score
- `checkVictory()` : checks if any player has won

11 MIS of Reinforcement Learning Environment Module

11.1 Module

Reinforcement Learning Environment (M5)

11.2 Uses

- **M4** – Uses the following access programs:
  - updateState
  - getState
  - validateMove
- **Data Types:** AIMove, GameStateData, Reward (Float)
- **Exception Types:** StepError, RenderError

11.3 Syntax

11.3.1 Exported Constants

- `MAX_TURNS`
- `REWARD_SCALE`

11.3.2 Exported Access Programs

Name	In	Out	Exceptions
step	action:AIMove	GameStateData, Reward	StepError
reset	-	GameStateData	-
render	-	-	RenderError

11.4 Semantics

11.4.1 State Variables

- `simulatedState` : An internal instance of M4 to manage the simulation
- `turnCount` : the number of turns elapsed in the current episode

11.4.2 Environment Variables

`display`: the graphical output window or context used to visualize the simulation

11.4.3 Assumptions

Assumes AIMove actions are provided in the valid, non-corrupt format.

11.4.4 Access Routine Semantics

step(action):

- transition: Applies action to simulatedState using M4. Runs opponent logic. Increments turnCount.
- output: Returns the new state (from simulatedState.getState()) and the calculated Reward.
- exception: raises `StepError` if action is invalid

reset():

- transition: Resets simulatedState to a new initial game. Resets turnCount to 0.
- output: Returns the initial simulatedState.getState().
- exception: -

render():

- transition: Modifies the display environment variable to show a visualization of simulatedState.
- output: -
- exception: raises `RenderError` if the display fails

11.4.5 Local Functions

- `applyOpponentLogic()` : Simulates opponent moves
- `calculateReward()`: computes reward based on change in simulatedState

12 MIS of AI Model Module

12.1 Module

AI Model (M6)

12.2 Uses

- **GameStateData** (Data Type): Represents the current digital state of the game, used as input for AI predictions.
- **AIMove** (Data Type): Encodes a move generated or evaluated by the AI model.
- **PredictionError** (Exception Type): Raised when the AI model fails to generate a valid move prediction.

12.3 Syntax

12.3.1 Exported Constants

- `MODEL_PATH` - path to trained neural network weights

12.3.2 Exported Access Programs

Name	In	Out	Exceptions
decide	state:GameStateData	AIMove	PredictionError
getMoveConfidence	move:AIMove	Float	-
explainMove	move:AIMove	String	-

12.4 Semantics

12.4.1 State Variables

- `modelWeights` - the loaded neural network parameters
- `policyNetwork` - DRL policy network
- `lastPredictionCache` - cache the last predictions outputs for getMoveConfidence

12.4.2 Environment Variables

N/A

12.4.3 Assumptions

- Assumes input `GameStateData` is valid.
- Assumes model weights at `MODEL_PATH` are loaded correctly before use.

12.4.4 Access Routine Semantics

decide(state):

- transition: Evaluates state with policyNetwork. Caches results in lastPredictionCache.
- output: Returns the optimal AIMove.
- exception: raises `PredictionError` if model fails to evaluate the state

getMoveConfidence(move):

- transition: -
- output: Returns a Float (e.g., 0.0-1.0) representing the confidence score for the specified move, retrieved from lastPredictionCache.
- exception: -

explainMove(move):

- transition: -
- output: returns a human-readable String explanation of why the model selected the move.
- exception: -

12.4.5 Local Functions

- `preprocessState()`: Converts GameStateData into model input format
- `postprocessOutput()`: Converts model output into a strctured AImove

13 MIS of Game State Database Module

13.1 Module

Game State Database (M7)

13.2 Uses

- **GameStateData** (Data Type): Represents the current or historical state of the game.
- **DBWriteError** (Exception Type): Raised when writing to the database fails.
- **DBReadError** (Exception Type): Raised when reading from the database fails.
- **List[GameStateData]** (Data Type): A collection of game states, e.g., for querying player history.

13.3 Syntax

13.3.1 Exported Constants

- `DB_PATH` - database file or server location
- `MAX_ENTRIES` - maximum stored states

13.3.2 Exported Access Programs

Name	In	Out	Exceptions
writeState	state:GameStateData	-	DBWriteError
readState	gameID:String	GameStateData	DBReadError
queryHistory	playerID:String	List[GameStateData]	DBReadError

13.4 Semantics

13.4.1 State Variables

- `dbConnection` - active connection to the database
- `gameRecords` - cached recently accessed game records

13.4.2 Environment Variables

`database` - external file system at DB_PATH

13.4.3 Assumptions

Assumes the database is accessible and the schema matches the expected structure.

13.4.4 Access Routine Semantics

writeState(state):

- transition: Serializes and writes the state to the database.
- output: -
- exception: DBWriteError if the write operation fails.

readState(gameID):

- transition: Retrieves and deserializes the game state for the specified gameID.
- output: returns `GameStateData` for specified game
- exception: raises `DBReadError` if gameID is not found or read fails

queryHistory(playerID):

- transition: Queries the database for all game states associated with playerID.
- output: returns list of `GameStateData` for given player
- exception: raises `DBReadError` if query fails

13.4.5 Local Functions

- `serializeState()` : converts GameStateData to storable format
- `deserializeState()` : converts stored format back to GameStateData
- `openConnection()` : establishes dbConnection
- `closeConnection()` : terminates dbConnection

14 MIS of Image Queue Module

14.1 Module

Image Queue (M8)

14.2 Uses

- **FrameData** (Data Type): Represents an image frame from the camera or simulation.
- **QueueFullError** (Exception Type): Raised when attempting to enqueue a frame into a full queue.
- **QueueEmptyError** (Exception Type): Raised when attempting to dequeue or peek from an empty queue.

14.3 Syntax

14.3.1 Exported Constants

- `QUEUE_SIZE` - maximum buffer size
- `TIMEOUT` - maximum wait time for enqueue/dequeue

14.3.2 Exported Access Programs

Name	In	Out	Exceptions
enqueue	frame:FrameData	-	QueueFullError
dequeue	-	FrameData	QueueEmptyError
peek	-	FrameData	QueueEmptyError
isFull	-	Boolean	-
isEmpty	-	Boolean	-

14.4 Semantics

14.4.1 State Variables

- `queueBuffer` - stores frames in order
- `head` - pointer to front of the queue
- `tail` - pointer to end of the queue
- `size` - current number of frames in the queue

14.4.2 Environment Variables

- None

14.4.3 Assumptions

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

14.4.4 Access Routine Semantics

enqueue(frame):

- transition: Adds frame to queueBuffer at tail. Increments tail and size.
- output: returns True if successful

- exception: raises QueueFullError if isFull() is True.

dequeue():

- transition: Removes the frame from queueBuffer at head. Increments head and decrements size.
- output: Returns the FrameData from the head.
- exception: raises `QueueEmptyError` if isEmpty() is True.

peek():

- transition: -
- output: returns next FrameData without removing it from queueBuffer.
- exception: raises `QueueEmptyError` if isEmpty() is True

isFull():

- transition: -
- output: returns True if size == QUEUE_SIZE, False otherwise
- exception: -

isEmpty():

- transition: -
- output: returns True if size == 0, False otherwise
- exception: -

14.4.5 Local Functions

- `resetQueue()` : Resets head, tail, and size.

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.

15 Appendix

[Extra information if required —SS]

Appendix — Reflection

[Not required for CAS 741 projects —SS]

The information in this section will be used to evaluate the team members on the graduate attribute of Problem Analysis and Design.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?
2. What pain points did you experience during this deliverable, and how did you resolve them?
3. Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. your peers, stakeholders, potential users)? For those that were not, why, and where did they come from?
4. While creating the design doc, what parts of your other documents (e.g. requirements, hazard analysis, etc), it any, needed to be changed, and why?
5. What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)
6. Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)