# Module Interface Specification for Software Engineering

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

November 5, 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 (-$\infty$, $\infty$) |
| natural number | $\mathbb{N}$ | a number without a fractional component in [1, $\infty$) |
| real | $\mathbb{R}$ | any number in (-$\infty$, $\infty$) |

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 | |
| Behaviour-Hiding | Input Parameters<br>Output Format<br>Output Verification<br>Temperature ODEs<br>Energy Equations<br>Control Module<br>Specification Parameters Module |
| Software Decision | Sequence Data Structure<br>ODE Solver<br>Plotting |

Table 1: Module Hierarchy

# 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 CV Model Module

## 7.1 Module

CV Model

## 7.2 Uses

Image Queue, Board Representation

## 7.3 Syntax

### 7.3.1 Exported Constants

None

### 7.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| processImage | imageData : Image | boardState : BoardState | InvalidImageError |

## 7.4 Semantics

### 7.4.1 State Variables

currentFrame : Image
boardState : BoardState

#### 7.4.2 Environment Variables

None

#### 7.4.3 Assumptions

Lighting and camera position are consistent; images are valid.

#### 7.4.4 Access Routine Semantics

processImage(imageData):

- transition: updates *currentFrame* with input imageData
- output: returns *boardState* computed from *currentFrame* using:
  - detectPieces(currentFrame)
  - convertToBoardState(pieces)
- exception: InvalidImageError if imageData is unreadable or corrupted

#### 7.4.5 Local Functions

detectPieces(image : Image) : PieceList
convertToBoardState(pieces : PieceList) : BoardState

# 8 MIS of Board Representation Module

## 8.1 Module

Board Representation

## 8.2 Uses

CV Model, Game-State Manager, User Interface

## 8.3 Syntax

### 8.3.1 Exported Constants

None

### 8.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| updateBoard | boardState : BoardState | - | InvalidStateError |
| getBoardState | - | boardState : BoardState | - |

## 8.4 Semantics

### 8.4.1 State Variables

board : BoardState
Initial state = empty board

### 8.4.2 Assumptions

Board state updates atomically.

### 8.4.3 Access Routine Semantics

updateBoard(boardState):

- transition: replaces *board* with *boardState*
- exception: InvalidStateError if boardState inconsistent

getBoardState():

- output: returns *board*

### 8.4.4 Local Functions

validateBoardState(boardState : BoardState) : Boolean

# 9 MIS of User Interface Module

## 9.1 Module

User Interface

## 9.2 Uses

Board Representation, Game-State Manager

## 9.3 Syntax

### 9.3.1 Exported Constants

None

### 9.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| displayBoard | boardState : BoardState | - | - |
| getPlayerMove | - | playerAction : Action | InputError |

## 9.4 Semantics

### 9.4.1 Environment Variables

Screen, Keyboard/Mouse

### 9.4.2 Assumptions

User provides input; boardState is valid.

### 9.4.3 Access Routine Semantics

displayBoard(boardState):

- output: renders boardState on screen

getPlayerMove():

- output: returns player's chosen move
- exception: InputError if invalid move

### 9.4.4 Local Functions

parsePlayerInput(input : Keyboard/Mouse) : Action

# 10 MIS of Game Database Module

## 10.1 Module

Game Database

## 10.2 Uses

Game-State Manager

## 10.3 Syntax

### 10.3.1 Exported Constants

None

### 10.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| recordGame | gameData : GameData | - | DatabaseError |
| fetchRecord | query : Query | gameData : GameData | NotFoundError |

## 10.4 Semantics

### 10.4.1 State Variables

database : List[GameData]
Initial state = empty list

### 10.4.2 Assumptions

Database integrity maintained; queries are valid.

### 10.4.3 Access Routine Semantics

recordGame(gameData):

- transition: appends *gameData* to *database*
- exception: DatabaseError if write fails

fetchRecord(query):

- output: returns matching *gameData*
- exception: NotFoundError if no record matches query

### 10.4.4 Local Functions

validateQuery(query : Query) : Boolean

# 11 MIS of Game-State Manager Module

## 11.1 Module

Game-State Manager

## 11.2 Uses

Game Database, AI Model, Board Representation

## 11.3 Syntax

### 11.3.1 Exported Constants

None

### 11.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| updateState | playerAction : Action | - | InvalidMoveError |
| getNextAction | - | aiAction : Action | - |

## 11.4 Semantics

### 11.4.1 State Variables

currentState : BoardState
Initial state = starting board layout

### 11.4.2 Access Routine Semantics

updateState(playerAction):

- transition: applies playerAction to *currentState*
- exception: InvalidMoveError if action illegal

getNextAction():

- output: returns AI's next action based on *currentState*

### 11.4.3 Local Functions

validateAction(action : Action, state : BoardState) : Boolean

# 12 MIS of AI Model Module

## 12.1 Module

AI Model

## 12.2 Uses

Game-State Manager, Reinforcement Learning Environment

## 12.3 Syntax

### 12.3.1 Exported Constants

None

### 12.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| inferMove | boardState : BoardState | aiAction : Action | ModelError |
| trainModel | reward : float, loss : float | - | TrainingError |

## 12.4 Semantics

### 12.4.1 State Variables

weights : NumericMatrix
Initial state = random initialization

### 12.4.2 Access Routine Semantics

inferMove(boardState):

- output: returns *aiAction* predicted by model
- exception: ModelError if inference fails

trainModel(reward, loss):

- transition: updates *weights* using reward and loss
- exception: TrainingError if model update fails

### 12.4.3 Local Functions

predictNextState(boardState : BoardState) : BoardState
updateWeights(loss : float)

# 13 MIS of Reinforcement Learning Environment Module

## 13.1 Module

Reinforcement Learning Environment

## 13.2 Uses

AI Model

## 13.3 Syntax

### 13.3.1 Exported Constants

None

### 13.3.2 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| generateReward | aiAction : Action | reward : float | - |
| computeLoss | reward : float, aiAction : Action | loss : float | - |

## 13.4 Semantics

### 13.4.1 Access Routine Semantics

generateReward(aiAction):

- output: returns numeric reward based on AI action

computeLoss(reward, aiAction):

- output: computes *loss* for model update

### 13.4.2 Local Functions

calculateReward(aiAction : Action) : float
calculateLoss(reward : float, aiAction : Action) : float

# 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

[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)