Module Interface Specification for SFWRENG 4G06: Dice Duels: Duel of the Eights

Team 9, dice_devs
John Popovici
Nigel Moses
Naishan Guo
Hemraj Bhatt
Isaac Giles

January 12, 2025

1 Revision History

Table 1: Revision History

Date	**Developer(s)**	**Change**
20-01-10	Hemraj Bhatt	Added content to section 3
20-01-11	John Popovici	Added content to section 4; Updated section 2
...	...	...

Contents

2 Symbols, Abbreviations and Acronyms

See SRS Documentation and MG Documentation.

3 Introduction

The following document details the Module Interface Specifications for the Duel of the Eights game. The game takes inspiration from Yahtzee and introduces a platform that enables players to create custom Yahtzee-like game variants. This platform includes preset options such as classic Yahtzee and an octahedron version, while also offering flexibility for users to define their own game variables. Players can customize aspects such as the number and type of dice, ranging from cubed (6-sided) to octahedral (8-sided) and other multi-sided dice, as well as scoring mechanisms. Scoring can either follow the traditional end-of-game calculation seen in classic Yahtzee or adopt a per-round format, allowing for head-to-head matchups. The game can be played both locally and online, allowing players to be able to play with one another regardless of their distance from one another. [Fill in your project name and description —SS]

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at https://github.com/John-Popovici/duel-of-the-eights/tree/main. [provide the url for your repo —SS]

4 Notation

The structure of the MIS for modules comes from **?**, with the addition that template modules have been adapted from **?**. Template used from the SFWRENG 4G06GitHub.

The following table summarizes the primary scene-building classes and elements used by *Dice Duels: Duel of the Eights* through Godot. Many different types of nodes are used, each with specific functions and properties.

Object	**Description**
Object	Dynamic base class holding properties, methods, signals, and scripts.
Node	Inherits object, can be assigned as a child of another node.
Scene	A tree of nodes.

The following table summarizes the primary coding data types used by *Dice Duels: Duel of the Eights* through Godot.

Data Type	**Notation**	**Description**
null	*null*	Denotes the absence of a value.
boolean	*bool*	Holds one of 2 values, true or false.
integer	*int*	A number without a fractional component within the bounds of signed 64-bit integer type $[-2^{63}, 2^{63} - 1]$.
float or decimal	*float*	A decimal number, as representable by 64-bit double-precision floating-point.
string	*String* or " . . . "	A sequence of unicode characters.
signal	*signal*	Allows connected objects to react to events.

The following table summarizes the primary data structures used by *Dice Duels: Duel of the Eights* through Godot.

Structure	**Example**	**Description**
array	$[5, "Hello", 2.3]$	Sequence of data elements.
dictionary	$\{"White" : 35, "Red" : 10\}$	Set of key and value pairs.

The specification of *Dice Duels: Duel of the Eights* uses some derived data types such as Vector3. Vector3 contains three floating point numbers, representing coordinates or direction vectors. In addition, *Dice Duels: Duel of the Eights* 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 Output Format Output Verification Temperature ODEs Energy Equations Control Module Specification Parameters Module
Software Decision	Sequence Data Structure ODE Solver Plotting

Table 2: 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]

This is the draft MIS section

3.1 MIS of MultiGameManager Module

3.1.1 Module

MultiGameManager

3.1.2 Uses

GameManager, PlayerManager, CustomizationMenu

3.1.3 Syntax

3.1.3.1 Exported Constants

None.

3.1.3.2 Exported Access Programs

- `start_game_sequence()`
- `finish_game(winner: String)`
- `apply_customizations()`

3.1.4 Semantics

3.1.4.1 State Variables

- `games_played`: Tracks completed games.
- `total_games`: Total number of games in the sequence.

3.1.4.2 Environment Variables

Player inputs for selecting upgrades.

3.1.4.3 Assumptions

All dependencies, such as PlayerManager and GameManager, are initialized.

3.1.4.4 Access Routine Semantics

- `start_game_sequence()`
  Transition: Initializes the first game in the sequence.
  Exception: Throws error if no players are connected.
- `finish_game(winner: String)`
  Transition: Updates win count and triggers customization phase.
- `apply_customizations()`
  Transition: Applies player-selected upgrades to the next game.

3.1.4.5 Local Functions

- `update_scores()`
- `check_win_conditions()`

3.2 MIS of GameManager Module

3.2.1 Module

GameManager

3.2.2 Uses

PlayerManager, ScoreCalculator

3.2.3 Syntax

3.2.3.1 Exported Constants

None.

3.2.3.2 Exported Access Programs

- `initialize_game()`
- `end_round()`
- `finalize_scores()`

3.2.4 Semantics

3.2.4.1 State Variables

- `current_round`: Tracks the current round in the ongoing game.

3.2.4.2 Environment Variables

Network manager for multiplayer actions.

3.2.4.3 Assumptions

Game rules are preloaded, and dice configurations are valid.

3.2.4.4 Access Routine Semantics

- `initialize_game()`
  Transition: Sets up dice, players, and round configurations.
- `end_round()`
  Transition: Ends the current round and updates scores.
- `finalize_scores()`
  Transition: Calculates and displays final scores for the game.

3.2.4.5 Local Functions

- `process_dice_rolls()`
- `apply_modifiers()`

3.3 MIS of PlayerManager Module

3.3.1 Module

PlayerManager

3.3.2 Uses

GameManager

3.3.3 Syntax

3.3.3.1 Exported Constants

None.

3.3.3.2 Exported Access Programs

- `update_player_stats(player_id: String, stats: Dictionary)`
- `apply_consumable(player_id: String, consumable: String)`

3.3.4 Semantics

3.3.4.1 State Variables

- `player_stats`: A dictionary containing player-specific data (health, score, etc.).

3.3.4.2 Environment Variables

Input from game events or player actions.

3.3.4.3 Assumptions

Each player has a unique `player_id`.

3.3.4.4 Access Routine Semantics

- `update_player_stats(player_id, stats)`
  Transition: Updates player state variables based on the input dictionary.
- `apply_consumable(player_id, consumable)`
  Transition: Applies the effect of a consumable item to the specified player.

3.3.4.5 Local Functions

- `track_modifiers()`
- `reset_player_state()`

3.4 MIS of NetworkManager2P Module

3.4.1 Module

NetworkManager2P

3.4.2 Uses

GameManager, PlayerManager, CustomizationMenu

3.4.3 Syntax

3.4.3.1 Exported Constants

None.

3.4.3.2 Exported Access Programs

- `start_connection()`
- `disconnect()`
- `send_data(data: Dictionary)`
- `receive_data()`
- `handle_disconnection(player_id: String)`

3.4.4 Semantics

3.4.4.1 State Variables

- `connected_peers`: A dictionary tracking the status of connected players.

3.4.4.2 Environment Variables

Network events, such as player join or disconnect signals.

3.4.4.3 Assumptions

Assumes a reliable network transport layer. Players do not exceed the 2-player limit for this module.

3.4.4.4 Access Routine Semantics

- `start_connection()`
  Transition: Establishes a peer-to-peer connection and initializes the network state.
- `disconnect()`
  Transition: Cleans up network state and disconnects from peers.
- `send_data(data)`
  Transition: Sends serialized data to the connected peer.
- `receive_data()`
  Output: Processes incoming data and synchronizes game state.
- `handle_disconnection(player_id)`
  Transition: Notifies the system of the disconnection and updates state.

3.4.4.5 Local Functions

- `serialize_data(data: Dictionary) -> String`
- `deserialize_data(data: String) -> Dictionary`

7 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)