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 15, 2025

1 Revision History

Table 1: Revision History

Date	**Developer(s)**	**Change**
25-01-10	Hemraj Bhatt	Added content to section 3
25-01-11	John Popovici	Added content to section 4; Updated section 2
25-01-14	Naishan Guo	Added content to section 6
25-01-14	Nigel Moses	Added content to section 5
...	...	...

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 section is taken directly from the Module Guide document for this project.

The modules are categorized into three types: Hardware-Hiding, Behavior-Hiding, and Software Decision modules. Below is the hierarchy:

5.1 Hardware-Hiding Modules

- **NetworkManager2P Module:** Manages connection and synchronization for two-player games.
- **GameUI Module:** Provides the interface for interacting with the game and displaying relevant data.

5.2 Behavior-Hiding Modules

- **MultiGameManager Module:** Manages the sequence of multiple Yahtzee games and customization phases.
- **PlayerManager Module:** Tracks player states, scores, and upgrades.
- **GameManager Module:** Handles a single game of Yahtzee.
- **GameSettings Module:** Loads and stores settings for this Yahtzee variant.
- **CustomizationMenu Module:** Implements dice and game customization between games.
- **DynamicScoreboard Module:** Tracks and displays scores dynamically.
- **CustomBaseDie Module:** Handles the 3D dice models, textures, and physics.
- **DynamicDiceContainer Module:** Manages dice interactions and rendering.

5.3 Software Decision Modules

- **ScoreCalculator Module:** Calculates scores for dice rolls based on Yahtzee rules and custom modifiers.

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

GameSettings, GameManager, CustomizationMenu, networkmanager2p

3.1.3 Syntax
3.1.3.1 Exported Constants
None.
3.1.3.2 Exported Access Programs

- `finish_game(winner: String, myPlayerFinalStats: Dictionary, OpponentFinalStats: Dictionary)`
- `loadGameSetup()`

3.1.4 Semantics
3.1.4.1 State Variables

- `current_game_count`: Tracks the number of completed games
- `total_games`: Total number of games in the sequence.
- `self_wins`: Tracks the number of wins the user has
- `opponent_wins`: Tracks number of wins the opponent has
- `game_settings`: Tracks the current settings and parameters for the current game
- `hand_settings`: Tracks the settings and requirements for the valid dice hands of the current game session (i.e. requirements for a full house, requirements for a yatzee, etc.)

3.1.4.2 Environment Variables

N/A

3.1.4.3 Assumptions

All dependencies, such as GameSettings and GameManager are initialized.

3.1.4.4 Access Routine Semantics

- `finish_game(winner: String, myPlayerFinalStats: Dictionary, OpponentFinalStats: Dictionary)`
  Transition: Updates win count, triggers customization phase.

- `loadGameSetup()`
  Transition: Retrieves and implements the game settings for this session.

3.1.4.5 Local Functions

- `start_next_game()`

- `start_game()`

- `end_game()`

- `_on_settings_ready(_game_settings: Dictionary, _hand_settings: Dictionary)`

- `recieve_game_settings(_game_settings: Dictionary, _hand_settings: Dictionary)`

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
DynamicDiceContainer, networkmanager2p

3.3.3 Syntax
3.3.3.1 Exported Constants

- `N/A`

3.3.3.2 Exported Access Programs

- `setup_player(_myPlayer: bool, initial_health: int, _playerName: String, _hostDevice: bool, _dice_container: Node3D)`
- `update_score(new_score: int)`
- `get_total_score()`
- `get_Last_score()`
- `adjust_health(points: int)`
- `getHealth()`
- `roll_dice()`
- `roll_selected_dice()`
- `pass_roll()`

- setRolls(_rolls: Array)
- getRolls()
- get_dice()
- clearRolls()
- getStats()
- getState()

3.3.4 Semantics

3.3.4.1 State Variables

- score: Contains the players current total score accumulated throughout the current game.
- last_score: Contains the most recent score achieved by the player during the last round.
- health_points: Contains the current health of the player.
- rolls: Tracks the players current dice roll values
- dice: Tracks the players physical dice.
- selected_hand: Tracks the players chosen dice hand for the current round
- myPlayer: checks if this instance of playermanager is for the player themselves, or for the player's opponent
- playerName: Name of the player
- hostDevice: checks if player's device is the current host of the game

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

- setup_player(_myPlayer: bool, initial_health: int, _playerName: String, _hostDevice: bool, _dice_container: Node3D)
  Transition: Initialized the player with their default values and game settings.

- `update_score(new_score: int)`
  Transition: updates the total score of the player to include the new score from the most recent round, and updates the last score to be the new score achieved during that round.

- `get_total_score()`
  Output: Returns the player's total score.

- `get_Last_score()`
  Output: Returns the player's last score from the previous round.

- `adjust_health(points: int)`
  Transition: updates the health of the player.

- `getHealth()`
  Transition: Returns the player's health.

- `roll_dice()`
  Transition: Rolls all the Dice. The rolled dice are tracked, and the new dice roll values retrieved.

- `roll_selected_dice()`
  Transition: Rolls only the dice specified by the player. The rolled dice are tracked, and the new dice roll values retrieved.

- `pass_roll()`
  Transition: Skips the dice roll for that turn

- `setRolls(_rolls: Array)`
  Transition: Sets the values of the player's dice to the specified value

- `getRolls()`
  Output: Returns the dice values

- `get_dice()`
  Output: tells the requester which dice belongs to the player

- `clearRolls()`
  Transition: clears the dice from the board and resets the dice values.

- `getStats()`
  output: Returns a few of the players state variables, providing the player's basic stats for the game (Name, total score, current health, etc)

- `getState()`
  output: Returns ALL of the players state variables tracked by this module, providing all relevant information about the player for the game.

3.3.4.5 Local Functions

- `roll_rolling_or_invalid_dice()`
- `checkIfDiceValidThenRead()`
- `readRolls()`

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`

3.5 MIS of ScoreCalculator Module

3.5.1 Module
ScoreCalculator

3.5.2 Uses
N/A

3.5.3 Syntax
3.5.3.1 Exported Constants

- `N/A`

3.5.3.2 Exported Access Programs

- `initializeValues()`

- `calculate_hand_score() -> Array[int]`

3.5.4 Semantics
3.5.4.1 State Variables

- `SinglesTotal`: Variable used to track the total score of every "singles" type hand for the purposes of bonus score calculation

- `BonusThreshold`: Tracks the maximum bonus score that can be given to the player's score for that round

- `BonusScore`: The bonus score added in addition to the players normal score acquired for that round.

- `BonusUsed`: Checks if the bonus score has been applied yet.

- `BonusExists`: Checks if a bonus score should be applied for this round

3.5.4.2 Environment Variables
N/A
3.5.4.3 Assumptions
The dice rolls are valid, and that no invalid hand type is requested
3.5.4.4 Access Routine Semantics

- `initializeValues()`
  Transition: Initialized the default values for the state variables of this module

- `calculate_hand_score() -> Array[int]`
  Output: detects the type of hand. Then, based on the hand type, it calculates the score before returning the player's score for the round.
  Transition: Depending on the type of hand, and if a bonus score is valid for this round, the state variables may be updated to track the application of the bonus score.

3.5.4.5 Local Functions

- `setupBonus(_bonusHand: Dictionary)`

3.6 MIS of DynamicScoreboard Module

3.6.1 Module
DynamicScoreboard

3.6.2 Uses
N/A

3.6.3 Syntax
3.6.3.1 Exported Constants
N/A

3.6.3.2 Exported Access Programs

- `populate_scoreboard(hand_settings: Dictionary)`

- `updateButtonScore(_score: int)`

- `updateBonusButtonScore(_score: int)`

- `setAllButtonsDisable(state: bool)`

3.6.4 Semantics
3.6.4.1 State Variables

- `lastButton`: Tracks the last button pressed by the player
- `BonusButton`: Tracks the bonus button
- `AllButtons`: Tracks every single button relavent to the hands and their corrisponding scores on the scoreboard.
- `hand_selected`: tracks the currently selected hand
- `bonusExists`: Checks if a bonus exists for the current hand
- `TotalLabel`: Tracks the total overall score for the player.

3.6.4.2 Environment Variables

- `vbox_container`: used to help interact with the UI interface to help display the scoreboard.

3.6.4.3 Assumptions

Assumes that every hand option to be displayed is unique to each other and valid for the rules specified by the game settings

3.6.4.4 Access Routine Semantics

- `populate_scoreboard(hand_settings: Dictionary)`
  Output: Populates the scoreboard with valid hands allowed by the game settings
- `updateButtonScore(_score: int)`
  Transition: Updates the score for a selected hand
- `updateBonusButtonScore(_score: int)`
  Transition: Updates the bonus score that will be added to the score for a hand
- `setAllButtonsDisable(state: bool)`
  Transition: enables or disables all the buttons relevant to scoring for the scoreboard.

3.6.4.5 Local Functions

- `_on_hand_selected(hand: Dictionary, button: Button)`

3.7 MIS of CustomizationMenu Module

3.7.1 Module

CustomizationMenu

3.7.2 Uses

N/A

3.7.3 Syntax
3.7.3.1 Exported Constants
N/A

3.7.3.2 Exported Access Programs

- `show_customization()`
- `wait_customization()`

3.7.4 Semantics
3.7.4.1 State Variables

N/A

3.7.4.2 Environment Variables

N/A

3.7.4.3 Assumptions
Assumes that a full round has been properly completed.

3.7.4.4 Access Routine Semantics

- `show_customization()`
  Output: Shows the customization options available to the player and allows them to select customization options in between games
- `wait_customization()`
  Output: Waits for the opponent to select customization options in between games.

3.7.4.5 Local Functions

- `pass_customization()`
  Description: Skips the customization phase for the player

3.8 MIS of DynamicDiceContainer Module

3.8.1 Module
DynamicDiceContainer

3.8.2 Uses
CustomBaseDie

3.8.3 Syntax

3.8.3.1 Exported Constants

N/A

3.8.3.2 Exported Access Programs

- `roll_dice()`
- `roll_selected_dice()`
- `roll_rolling_or_invalid_dice()`
- `get_dice_values() -> Array[int]`
- `get_dice() -> Array[RigidBody3D]`
- `get_selected_dice() -> Array`
- `get_rolling_dice() -> Array`
- `get_invalid_dice() -> Array`
- `add_dice(dice_count: int, dice_type: int)`
- `move_dice_aside()`
- `move_dice_in_line()`

3.8.4 Semantics

3.8.4.1 State Variables

- `dice_nodes`: Tracks each dice object that is a part of this container module.
- `start_positions`: Specifies the stating positions of the dice

3.8.4.2 Environment Variables

N/A

3.8.4.3 Assumptions

Assumes all dice are of the same type.

3.8.4.4 Access Routine Semantics

- `roll_dice()`
  Transition: Rolls all the dice with a random force and torque
- `roll_selected_dice()`
  Transition: Rolls only the dices specified by the player.

- `roll_rolling_or_invalid_dice()`
  Transition: Rerolls any dice identified as an invalid roll or trapped in an endless roll.

- `get_dice_values() -> Array[int]`
  Output: retrieves the value of each rolled dice

- `get_dice() -> Array[RigidBody3D]`
  Output: retrives each dice object that is a part of the container

- `get_selected_dice() -> Array`
  output: retrieves a list of each die selected by the player

- `get_rolling_dice() -> Array`
  output: gets a list of all dice that is currently rolling

- `get_invalid_dice() -> Array`
  Output: gets a list of all dice that have been rendered invalid, wiether due to it's improper landing position, unreadable value, improper graphic rendering, or other issue.

- `add_dice(dice_count: int, dice_type: int)`
  Transformation: Adds new dice objects to the container

- `move_dice_aside()`
  Transformation: Moves dice not being re-rolled to the side of the board to prevent the dice from interfeering with the reroll

- `move_dice_in_line()`
  Transformation: Move dice once to an organized display on the board to make them easier to read.

3.8.4.5 Local Functions

- `clear_dice()`
  Description: removes each dice object that's part of the contianer.

3.9 MIS of CustomBaseDie Module

3.9.1 Module
CustomBaseDie

3.9.2 Uses
N/A

3.9.3 Syntax

3.9.3.1 Exported Constants
N/A

3.9.3.2 Exported Access Programs

- `populate_scoreboard(hand_settings: Dictionary)`

3.9.4 Semantics
3.9.4.1 State Variables

- `lastButton`: Tracks the last button pressed by the player

3.9.4.2 Environment Variables

- `vbox_container`: used to help interact with the UI interface to help display the scoreboard.

3.9.4.3 Assumptions
Assumes
3.9.4.4 Access Routine Semantics

- `populate_scoreboard(hand_settings: Dictionary)`
  Output: Populates the scoreboard with valid hands allowed by the game settings

3.9.4.5 Local Functions

- `_on_hand_selected(hand: Dictionary, button: Button)`

3.10 MIS of GameSettings Module

3.10.1 Module
GameSettings

3.10.2 Uses
N/A

3.10.3 Syntax
3.10.3.1 Exported Constants
N/A

3.10.3.2 Exported Access Programs

- `populate_scoreboard(hand_settings: Dictionary)`

3.10.4 Semantics

3.10.4.1 State Variables

- `lastButton`: Tracks the last button pressed by the player

3.10.4.2 Environment Variables

- `vbox_container`: used

3.10.4.3 Assumptions

Assumes

3.10.4.4 Access Routine Semantics

- `populate_scoreboard(hand_settings: Dictionary)`
  Output: Populates the scoreboard with valid hands allowed by the game settings

3.10.4.5 Local Functions

- `_on_hand_selected(hand: Dictionary, button: Button)`

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)