Module Interface Specification for *Uno Flip Remix*

Team 24

Mingyang Xu
Kevin Ishak
Jianhao Wei
Zain-Alabedeen Garada
~~Zheng Beng Liang~~

April 2, 2025

1 Revision History

Date	Developer	Notes
January 12th, 2025	Kevin Ishak	Initialize template, add rough draft of section 3,4 and 5
January 13th, 2025	Jianhao Wei	Add modules into section 6
January 13th, 2025	Zain-Alabedeen Garada	Add modules into section 6
January 17th, 2025	Zheng Bang Liang	Added all modules for Behavior Hiding Modules
January 17th, 2025	Jianhao Wei, Kevin Ishak, Zain-Alabedeen Garada	Modify behavior hiding modules that Zheng added and add the rest of the MIS modules
January 17th, 2025	Jianhao Wei	Modify section 3 and wrote section 4, 5, 16. Communicate with other members and wrote section 17
January 28th, 2025	Jianhao Wei	Make changes based on peer feedbacks. Please see commits and issue trackers for detail
April 1st, 2025	Kevin Ishak	Edits based on TA feedback for the final report revision

2 Symbols, Abbreviations and Acronyms

The following table summarizes the symbols and abbreviations used throughout this document. These definitions are consistent with those listed in the Module Guide (MG). For the complete reference, see the full MG at this link.

Symbol / Abbreviation	**Definition**
AC	Anticipated Change
UC	Unlikely Change
MG	Module Guide
MIS	Module Interface Specification
SRS	Software Requirements Specification
UI	User Interface
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
DAG	Directed Acyclic Graph
HUD	Heads-Up Display
UNO Flip	A variant of the classic UNO game featuring a two-sided deck

Contents

3 Introduction

UNO Flip is a modern twist on the traditional UNO card game, incorporating an innovative double-sided card deck with "light" and "dark" sides. Players must adapt their strategy dynamically as the game flips between these two sides, creating unpredictable and engaging game play.

The goal of this project is to develop a digital version of UNO Flip that faithfully replicates the physical game play experience while introducing improvements such as rule automation, multiplayer networking, real-time animations, and a modern user interface.

This Module Interface Specification (MIS) provides detailed interface specifications for each software module described in the Module Guide (MG). It serves as a design reference for implementation and testing.

Complementary documents include:

- The Software Requirements Specification (SRS), which defines functional and non-functional system requirements.
- The Module Guide (MG), which outlines the modular decomposition and design philosophy.

4 Notation

The structure of the MIS for modules follows the documentation standards described in Hoffman and Strooper (1), with adaptations based on Ghezzi et al. (2). The mathematical notation follows conventions from Chapter 3 of Hoffman and Strooper (1995). This MIS also builds on the SFWRENG 4G06 GitHub template, available at this link.

The following tables summarize the primitive, object, and domain-specific data types used by the UNO Flip 3D software. These types form the basis for module specifications throughout this document.

Primitive Data Types

Data Type	Notation	Description
Boolean	boolean	Logical value representing `true` or `false`.
Integer	int	Whole numbers in the range $[-2^{63}, 2^{63} - 1]$.
Floating Point	float	Real numbers with fractional components using IEEE 754 32-bit representation.
Serialized Data Stream	serializedData	Binary-encoded data used for inter-module or inter-device communication.

Derived Object Data Types

Object Type	Notation	Description
String	String	A sequence of Unicode characters.
Generic Array	Array[Type]	A collection of elements of the specified type.
Dictionary	dictionary	Key-value mapping where each key corresponds to a specific element.
Graphics Object	GraphicObject	Object describing visual representation, used by the UI module.

Domain-Specific Data Types

Custom Type	Notation	Description
Card	Card	Represents an individual card with color, number, and action attributes.
Deck	Deck	A stack of `Card` objects supporting draw and shuffle operations.
Player	Player	Entity with a hand of cards, name, and status.
Game State	GameState	Tracks all gameplay data: turn order, decks, hands, discard piles, etc.
Move	Move	Encodes a player's chosen card and action.
Turn Direction	TurnDirection	Enum representing clockwise or counterclockwise direction.
Action Type	ActionType	Enum for card actions: `Draw2`, `Skip`, `Flip`, etc.
Color	Color	Enum for card colors: `Red`, `Blue`, `Green`, `Yellow`, `Wild`.

UNO Flip 3D 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

This section outlines the modular decomposition of the UNO Flip Remix system, categorized into three types: Hardware-Hiding, Behaviour-Hiding, and Software Decision modules. These categories follow the modular design principles established in the Module Guide (MG). Each module listed below is a leaf module from the MG hierarchy and is associated with an interface specification in this document.

5.1 Hardware-Hiding Modules

- **Backend/Server Module:** Serves as a virtual hardware abstraction layer between Unity clients and the network server. It is responsible for input/output processing and enabling communication across clients in a multiplayer environment.

5.2 Behaviour-Hiding Modules

- **Card Effect Module:** Executes the effects of special cards and updates the game state accordingly.
- **Turn Management Module:** Manages the order of player turns, including handling special conditions like "Reverse" or "Skip" cards.
- **User Interface Module:** Displays the game state to the user and accepts user inputs through interactive UI elements.
- **Save/Load Module:** Allows saving and restoring the game state from persistent storage.
- **Animation Module:** Provides animations for card movements, flips, and visual effects that enhance gameplay.
- **Output Module:** Renders visual and textual game outputs, including scoreboards and notifications.

5.3 Software Decision Modules

- **Multiplayer Networking Module:** Handles matchmaking, game state synchronization, and reliable communication between clients using Unity and a TCP-based server.
- **Verification Output Module:** Validates game logic and output correctness, ensuring rule compliance before and after each turn.
- **Input Module:** Converts raw user inputs into standardized data structures usable by the game logic and UI.
- **Control Logic Module:** Manages the overall game control flow, including transitions between states, coordination between modules, and input/output sequencing.

6 MIS of Backend Server Module

6.1 Module

BackendServer

6.2 Uses

InputModule, OutputModule, MultiplayerNetworkingModule

6.3 Syntax

6.3.1 Exported Constants

- MAX_CLIENTS: `int` — Maximum number of clients supported by the server.
- DEFAULT_PORT: `int` — Default port used for incoming socket connections.

6.3.2 Exported Access Programs

- startServer(port: int) → boolean
- acceptConnection() → Connection
- broadcastMessage(msg: String) → void
- shutdownServer() → void

6.4 Semantics

6.4.1 State Variables

- serverSocket: Stores the active server socket instance.
- clients: A list of active client connections.
- gameSessions: Stores game session state mapped to connected players.

6.4.2 Environment Variables

- OSNetworkStack: The operating system's network protocol stack (e.g., TCP/IP).
- UnityEngine.Platform: Unity's internal networking environment used to interface with the server.

6.4.3 Assumptions

- The host machine allows socket binding on the specified port.
- Unity clients conform to the expected messaging protocol.

6.4.4 Access Routine Semantics

- startServer(port) → boolean
  Transition: Binds a socket to the specified port and begins listening for incoming client connections.
  Output: Returns `true` if the server starts successfully; otherwise, `false`.
- acceptConnection() → Connection
  Transition: Accepts a new client connection request and appends it to the active client list.
  Output: Returns a new `Connection` object representing the connected client.
- broadcastMessage(msg) → void
  Transition: Sends the message `msg` to all connected clients via TCP.
- shutdownServer() → void
  Transition: Closes all active client connections and releases the server socket.

6.4.5 Local Functions

- validateMessageFormat(msg: String) → boolean
  Description: Checks if the incoming message string conforms to the expected JSON protocol.
- removeInactiveClients() → void
  Description: Iterates over the client list and removes disconnected or timed-out clients.
- logConnectionEvent(event: String) → void
  Description: Appends server-side events to a log for debugging and traceability.

7 MIS of Card Effect Module

7.1 Module

Card Effect

7.2 Uses

Hardwire Hiding

7.3 Syntax

7.3.1 Exported Constants

- DRAW_TWO_EFFECT: Specifies the effect identifier for a "Draw Two" card.
- SKIP_TURN_EFFECT: Specifies the effect identifier for a "Skip" card.
- FLIP_DECK_EFFECT: Specifies the effect identifier for a "Flip" card.

7.3.2 Exported Access Programs

- reverseDirection(playerId: int)
- skipTurn(playerId: int)
- triggerDrawCards(playerId: int, cardCount: Int)
- flipDeck()

7.4 Semantics

7.4.1 State Variables

- currentEffect: Track the current effect being applied
- effectQueue: Store the effects that are waiting to be applied

7.4.2 Environment Variables

- The special card type that current game environment allowed.

7.4.3 Assumptions

- All card effects are predefined.
- The "Flip" card effect toggles the entire game state between "light" and "dark" sides.

7.4.4 Access Routine Semantics

- reverseDirection(playerId: int, previousPlayerId: int): $\rightarrow$ void
  Transition: Reverse the direction the game is being played to the previous player by re-assigning the previous player next opportunity
- skipTurn(playerId: int, nextPlayerId: int) $\rightarrow$ void
  Transition: Skip the opportunity for the specified player to play and assignment the opportunity to the next player

- triggerDrawCards(playerId: int, cardCount: Int) → void
  Transition: Let the player specified to draw another card into their database and add the card count to their totalPower variable.

- flipDeck() → void
  Transition: Changes the deckSide variable and updates the game state to reflect the flipped deck.

7.4.5 Local Functions

- calculateNextPlayer(direction: String) → int
  Description: Determines the next player in the turn sequence after applying a "Skip" or "Reverse" effect.

- applyChainEffect(effectQueue: Array[String]) → void
  Description: Resolves multiple card effects in sequence defined in the input string array

- toggleDeckSide() → void
  Description: Switches the game state between "light" and "dark" sides during a "Flip" card effect.

8 MIS of Turn Management Module

8.1 Module

Turn Management

8.2 Uses

Input, Card Effect

8.3 Syntax

8.3.1 Exported Constants

None

8.3.2 Exported Access Programs

- validateMove(playerId: int, cardid: int)

- endTurn(playerId: int)

- shuffleDeck()

- drawCard(playerId: int)

8.4 Semantics

8.4.1 State Variables

- currentPlayer: Tracks the player whose turn it is
- deck: Represents the stack of remaining cards in the game.
- discardPile: Stores played cards
- playerHands: Stores each player's cards

8.4.2 Environment Variables

- maxPlayers: Maximum number of players allowed in a game
- flipEnabled: Boolean to toggle the flip functionality

8.4.3 Assumptions

- The number of players, game rules, player restrictions are preloaded
- The game environment is known

8.4.4 Access Routine Semantics

- validateMove(playerId: int, cardId: int) → boolean
  Output: Checks if a move is valid
- endTurn(playerId:int) → void
  Transition: Ends the current player's turn and starts the next
- shuffleDeck() → void
  Transition: Randomizes the card deck
- drawCard(playerId: int) → void
  Transition: Adds a card to the specified player's hand

8.4.5 Local Functions

- shuffleProcess(original: Array[String]) → Array[String]
  Description: Contain the random algorithm to shuffle the deck
- CardModifier(cardId: int) → void
  Description: Contain algorithm to draw different card to screen

9 MIS of User Interface Module

9.1 Module

User Interface

9.2 Uses

Output, Turn Management

9.3 Syntax

9.3.1 Exported Constants

- DEFAULT_THEME: Specifies the default theme for the game UI (e.g., light mode).
- FONT_STYLE: Default font style used across UI elements.
- ASSET_PATH: Directory path where assets are stored
- DEFAULT_CARD_SPRITE: Specifies the default card sprite to use if none is provided.

9.3.2 Exported Access Programs

- updateCardDisplay(playerId: int, cardId: int)
- showTurnIndicator(playerId: int)
- displayMessage(message: String)
- loadScene(type: String)
- loadAsset(assetName: String)
- unloadAsset(assetName: String)
- playSound(effectName: String)

9.4 Semantics

9.4.1 State Variables

- displayedCards: Tracks the cards currently visible for each player.
- turnIndicator: Indicates which player's turn it is.
- messageQueue: Stores pending notifications or chat messages to be displayed.
- theme: Specifies the current visual theme in light mode or dark mode.

- loadedAssets: Tracks assets currently loaded into memory.
- audioSettings: Stores configuration for playing audio
- assetCache: Cache for frequently accessed assets to improve performance.
- assetDirectory: Path to the directory containing all assets

9.4.2 Environment Variables

- The resolution of the device being used.

9.4.3 Assumptions

- The UI module assumes that game state updates from the multiplayer networking and turn management modules are reliable.
- All required assets are preloaded by the Save/Load module.
- Multiplayer synchronization ensures accurate real-time updates across all connected devices.
- All assets are correctly named and stored in the specified directory.
- The module assumes sufficient memory and storage are available for caching assets.
- Dependencies for visual and audio formats are preinstalled on the system.

9.4.4 Access Routine Semantics

- updateCardDisplay(playerId: int, cardId: int) → void
  Transition: Updates the player's visible hand to reflect the current state of their cards.
- showTurnIndicator(playerId: int) → void
  Transition: Highlights the current player's turn using visual indicators.
- displayMessage(message: String) → void
  Transition: Displays a notification or chat message on the game screen.
- loadScene(type: String) → void
  Transition: Load specific type of background with animation to the user interface
- loadAsset(assetName: String) → void
  Transition: Loads the specified asset from the asset directory into memory and returns a reference.

- unloadAsset(assetName: String) → void
  Transition: Removes the specified asset from memory to free up resources.

- playSound(effectName: String) → void
  Transition: Plays the specified sound effect from the audio assets directory.

9.4.5 Local Functions

- applyTheme(themeId: int) → void
  Description: Configures and applies the selected theme for the game UI.

- renderMessageQueue(messages: Array[String]) → void
  Description: Processes and displays pending messages in the queue.

- adjustUILayout() → void
  Description: Dynamically adjusts the layout based on the screen resolution and device type.

- cacheAsset(assetName: String) → void
  Description: Adds the specified asset to the cache for quick retrieval.

- clearCache() → void
  Description: Clears the asset cache to free up memory

- validateAsset(assetName: String) → void
  Description: Checks if the specified asset exists and is accessible.

10 MIS of Save/Load Module

10.1 Module

Save/Load

10.2 Uses

Hardwire Hiding

10.3 Syntax

10.3.1 Exported Constants

None

10.3.2 Exported Access Programs

- save(info: String, description: String)
- retrieve(description: String)
- delete(description: String)
- changeDesc(originalDesc: String, updateDesc: String)

10.4 Semantics

10.4.1 State Variables

- ifFull: Track if the database is full
- dict: The dictionary that stores the array index correspond with descriptions
- infoArray: The array that stores all the information

10.4.2 Environment Variables

None

10.4.3 Assumptions

The string and description stored does not contain any special characters

10.4.4 Access Routine Semantics

- save(info: String, description: String) → void
  Transition: Save the information into the database with description
- retrieve(description: String) → String
  Output: Return the information by its description
- delete(description: String) → void
  Transition: Delete the information in the database by its description
- changeDesc(originalDesc: String, updateDesc: String) → void
  Transition: change the description of a piece of information into another

10.4.5 Local Functions

- returnIndex(description: String) → int
  Description: Return the index of the infoArray based on the description.

11 MIS of Animation Module

11.1 Module

Animation

11.2 Uses

User Interface, Card Effect, Save/Load

11.3 Syntax

11.3.1 Exported Constants

None

11.3.2 Exported Access Programs

- move(cardId: int, distance: int, direction: String)
- flip(cardId: int)
- select(cardId: int)
- appear(cardId: int)
- disappear(cardId: int)

11.4 Semantics

11.4.1 State Variables

- cardSide: Track side the card is on
- cardColor: Track the color of the card
- cardPosition: Track the position of the card
- show: Track if the card is shown on the screen

11.4.2 Environment Variables

None

11.4.3 Assumptions

Each card has a unique id

11.4.4 Access Routine Semantics

- move(cardId: int, distance: int, direction: String) → void
  Transition: Move the card with specific id by a set amount of pixels with horizontal or vertical direction

- flip(cardId: int) → void
  Transition: Flip the card with specific id to show the opposite face

- select(cardId: int) → void
  Transition: Show the animation when the card is selected by the user

- appear(cardId: int) → void
  Transition: Show the card with specific id to the user screen

- disappear(cardId: int) → void
  Transition: Make the card with specific id to disappear from the user screen

11.4.5 Local Functions

- getCardInfo(id: int) → void
  Description: Get the info of the card to local state variables

- applyVisualElements(id: int) → void
  Description: Apply the visual effect to the user screen based on the id provided and update local state variables

12 MIS of Output Module

12.1 Module

Output

12.2 Uses

Card Effect

12.3 Syntax

12.3.1 Exported Constants

None

12.3.2 Exported Access Programs

- render(info: String, font: int, color: String, location: int)
- showCardEffect(id: int, effectNum: int)

12.4 Semantics

12.4.1 State Variables

None

12.4.2 Environment Variables

None

12.4.3 Assumptions

Each card has a unique id

12.4.4 Access Routine Semantics

- render(info: String, font: int, color: String, location: int) → void
  Transition: Display the information onto the screen with the font, color and location specified
- showCardEffect(id: int, effectNum: int) → void
  Transition: Using Card Effect module to show flip, skip or draw two on specific card

12.4.5 Local Functions

- checkEdge(font: int, location: int) → boolean
  Description: Check if the information displayed exceeds the boundary of the screen

13 MIS of Multiplayer Networking Module

13.1 Module

Multiplayer Networking

13.2 Uses

Verification Output, Save/Load, Animation

13.3 Syntax

13.3.1 Exported Constants

serverID: The serial number of the game room upon user request

13.3.2 Exported Access Programs

- createGameRoom(playerId: int, roomSettings: Array[String])
- joinGameRoom(playerId: int, roomId: int)
- broadcastUpdate(gameId: int, update: String)

13.4 Semantics

13.4.1 State Variables

- activeGames: Tracks all ongoing game sessions.
- connectedPlayers: List of currently connected players.

13.4.2 Environment Variables

- serverIP: IP address of the game server.
- timeoutLimit: Time limit for a player to respond during their turn.

13.4.3 Assumptions

- The connection between server and other machines can be established successfully
- The encryption and decryption methods are known

13.4.4 Access Routine Semantics

- createGameRoom(playerId: int, roomSettings: Array[String]) → void
  Transition: Creates a new game room by a specific user with specific setting
- joinGameRoom(playerId: int, roomId: int, publicKey: int) → int
  Transition: Adds a specific player to an existing room by its ID and public key for encryption and decryption purpose purposes
  Output: Return the public key of the server for encryption and decryption purposes
- broadcastUpdate(gameId: int, update: String) → void
  Transition: Sends game state updates to all players in a room.

13.4.5 Local Functions

- encryption(information: String, publicKey: int) → String
  Description: Contain encryption algorithm to encrypt data before sending using public key from user

- decryption(information: String, privateKey: int) → String
  Description: Contain decryption algorithm to decrypt data after receiving using the private key of game room

14 MIS of Verification Output Module

14.1 Module

Verification Output

14.2 Uses

None

14.3 Syntax

14.3.1 Exported Constants

None

14.3.2 Exported Access Programs

- captureOutput(playerId: int, info: String)

- validateOutput(info: String)

14.4 Semantics

14.4.1 State Variables

- outputBuffer: Temporarily store the incoming input received for later use

- validatedOutput: Store the input that has been validated by the module for later transmission

14.4.2 Environment Variables

- the validation algorithm the device is running on

14.4.3 Assumptions

- All output devices conform to Unity's input standard.
- The validation algorithm must make sure that there is no error or discrepancy occurring after the validation

14.4.4 Access Routine Semantics

- captureInput(info: String) → void
  Transition: Capture and save the information into the output buffer
- validateInput(info: String) → String, boolean
  Transition: validate the output from the outputBuffer using existing algorithms
  Output: Return the original output and a boolean indicating if the input can be validated

14.4.5 Local Functions

- algorithmDatabase(input: String, type: int) → String
  Description: Contain the algorithm that converts the input string to the format that can be used by other modules and return the converted input string
- serialization(input: String) → serializedData
  Description: Contain the algorithm to convert the input string into serialized data for inter-module or internet communications

15 MIS of Input Module

15.1 Module

Input

15.2 Uses

Hardwire Hiding

15.3 Syntax

15.3.1 Exported Constants

None

15.3.2 Exported Access Programs

- captureInput(playerId: int, info: String)
- validateInput(info: String)
- convertInput(info: String, type: String)

15.4 Semantics

15.4.1 State Variables

- inputBuffer: Temporarily store the incoming input received for later use
- validatedInput: Store the input that has been validated by the module for later transmission

15.4.2 Environment Variables

- The version of supporting device that the software is running on
- the validation algorithm the device is running on

15.4.3 Assumptions

- All input devices conform to Unity's input standard.
- The validation algorithm must make sure that there is no error or discrepancy occurring after the validation

15.4.4 Access Routine Semantics

- captureInput(playerId: int, info: String) → void
  Transition: Capture and save the information into the input buffer
- validateInput(info: String) → String, boolean
  Transition: validate the input from the inputBuffer using existing algorithms
  Output: Return the original input and a boolean indicating if the input can be validated
- convertInput(action: String, type: String) → String
  Output: Convert the input from validatedInput into specific format that can be used by other modules

15.4.5 Local Functions

- algorithmDatabase(input: String, type: int) → String
  Description: Contain the algorithm that converts the input string to the format that can be used by other modules and return the converted input string

- serialization(input: String) → serializedData
  Description: Contain the algorithm to convert the input string into serialized data for inter-module or internet communications

16 Exception Handling Strategies

The exception handling is critical for our software since it directly impacts the user experience of our software. It is our responsibility to ensure that our customers have a good experience with our software. To prevent exception from happening in our software, we implement the following 4 strategies:

- **Limit Erroneous User Input:** We design the user interface such that the user input is bounded within a certain range to limit erroneous user input that might crash the software. We also include the input verification in Input module to ensure all the information that passed to the software are legitimate

- **Wrap External Resources:** We have design all of our function in our modules to wrap the resources and libraries they use from the global space of the software. This ensures that the exception in third-party software does not impact the integrity of own software.

- **Cleaning up resources:** We have implemented the mechanism to clean up unused resource promptly and reliably to make sure the exceptions do not occurs due to cache overload

- **Limiting Errors Instead of Handling Errors:** Instead of designing exception handling mechanism, we make sure our software is carefully designed and tested to reduce the chance of exception happening.

By implementing these strategies, we can reduce the chance for exception happening and limit the need of the exception handling mechanisms.

References

[1] D. M. Hoffman and P. Strooper, *Software Design: With C++ and Java*, Addison-Wesley, 1995.

[2] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, 2nd ed., Prentice Hall, 2003.