Module Interface Specification for The Crazy Tens

Team #25, The Crazy Four
Ruida Chen
Ammar Sharbat
Alvin Qian
Jiaming Li

November 13, 2025

1 Revision History

Date	Version	Notes
Nov 12th	Rev-1	Module M1-M11
Nov 12th	Rev-1	Module M12-M22
Nov 13th	Rev-1	Fix consistency
Nov 13th	Rev-1	Fix correlation
Nov 13th	Rev-1	Fix consistency

2 Symbols, Abbreviations and Acronyms

See MG Documentation at MG

Contents

3 Introduction

The following document details the Module Interface Specifications for The Crazy Tens

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at https://github.com/The-Crazy-Four-Games/Crazy-Eights-Game.

4 Notation

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 The Crazy Tens.

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 The Crazy Tens 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, The Crazy Tens 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**	**Level 3 (Leaf Modules)**
Hardware-Hiding Module		M20 (Server OS) M21 (Client Runtime) M22 (PostgreSQL)
Behaviour-Hiding Module	(Core Domain Logic)	M15 M16 M17 M18 M19
Software Decision Module	Backend (Server)	M1 M2 M3 M4 M5 M6
	Frontend (Client)	M7 M8 M9 M10 M11 M12 M13 M14

Table 1: Module Hierarchy

6 MIS of API Module (M1)

6.1 Module

API (Service Layer)

6.2 Uses

- M3 Matchmaking Module
- M4 Authentication Module
- M5 Repository Module

6.3 Syntax

6.3.1 Exported Constants

None.

6.3.2 Exported Access Programs

Name	In	Out	Exceptions
signup	creds: UserCredentials	user: User	AuthError, ValidationError
login	creds: UserCredentials	token: AuthToken	InvalidCredentials, ValidationError
createGame	token: AuthToken, options: GameOptions	session: GameSession	AuthError, LobbyError
getProfile	token: AuthToken	profile: UserProfile	AuthError, NotFound
updateProfile	token: AuthToken, profile: UserProfile	profile: UserProfile	AuthError, ValidationError, NotFound
deleteProfile	token: AuthToken	void	AuthError, NotFound

6.4 Semantics

6.4.1 State Variables

None. This module is stateless.

6.4.2 Environment Variables

- **RequestAdapter**: An environment capable of receiving and routing service requests.
- **ResponseAdapter**: An environment capable of sending responses back to the caller.

6.4.3 Assumptions

- The **RequestAdapter** is running and correctly routes to this module's services.
- Modules M3, M4, and M5 are available.
- `AuthToken` is an opaque type verifiable by M4.

6.4.4 Access Routine Semantics

signup(*creds*)

- transition: Validates *creds*. Calls `M4.registerUser(creds.username, creds.password)`. On success, calls `M5.createUser(data)`.
- output: Returns the newly created `User` abstract data type.
- exception: `ValidationError` if credentials format is invalid. `AuthError` if user already exists.

login(*creds*)

- transition: Validates *creds*. Calls `M4.loginUser(creds.username, creds.password)` to generate `AuthToken`.
- output: Returns the generated `AuthToken`.
- exception: `ValidationError` if credentials format is invalid. `InvalidCredentials` if authentication fails.

createGame(*token, options*)

- transition: Calls `M4.verifyToken(token)` to get a UserID. On success, calls `M3.createLobby(UserID` `options)`.
- output: Returns the `GameSession` abstract data type.
- exception: `AuthError` if *token* is invalid. `LobbyError` if lobby creation fails.

getProfile(*token*)

- transition: Calls `M4.verifyToken(token)` to get a UserID. On success, calls `M5.getUserProfile(Use`
- output: Returns the `UserProfile` abstract data type.
- exception: `AuthError` if *token* is invalid. `NotFound` if the user profile does not exist.

updateProfile(*token, profile*)

- transition: Calls `M4.verifyToken(token)` to get a UserID. Validates *profile* data. On success, calls `M5.updateUserProfile(UserID, profile)`.
- output: Returns the updated `UserProfile`.
- exception: `AuthError`. `ValidationError`. `NotFound`.

deleteProfile(*token*)

- transition: Calls `M4.verifyToken(token)` to get a UserID. On success, calls `M5.deleteUser(UserID)`
- output: `void`.
- exception: `AuthError`. `NotFound`.

6.4.5 Local Functions

None.

6.4.6 Considerations

- The secret of this module is the definition of the service layer endpoints and the data serialization format (e.g., JSON) used for transport.
- The web server implementation (e.g., Express) *uses* this module, it is not *part of* this module.

7 MIS of Real-time Gateway Module (M2)

7.1 Module

Real-time Gateway

7.2 Uses

- M4 Authentication Module
- M15 Game Action Module

7.3 Syntax

7.3.1 Exported Constants

None.

7.3.2 Exported Access Programs

Name	In	Out	Exceptions
handleConnection	socket: ClientSocket	void	SessionError
handleJoinGame	socket: ClientSocket, data: JoinGameData	void	SessionError, NotFound
handleSubmitMove	socket: ClientSocket, data: MoveData	void	InvalidMove, NotYourTurn, SessionError
registerGameSession	gameID: GameID, session: GameSession	void	LobbyError
emitGameState	socket: ClientSocket, state: GameState	void	
broadcastGameState	roomID: GameID, state: GameState	void	

7.4 Semantics

7.4.1 State Variables

- **activeGames**: Map$< GameID, GameSession >$ — A map holding the live `GameSession` objects for all currently active games.

7.4.2 Environment Variables

- **RealtimeAdapter**: The environment (e.g., a WebSocket server) that manages client connections.
- **ClientSocket**: An opaque handle representing a single client connection.

7.4.3 Assumptions

- The **RealtimeAdapter** is running and forwards events to these handlers.
- `AuthToken` (inside `JoinGameData`) is verifiable by M4.
- M19 is available to process game logic.

7.4.4 Access Routine Semantics

handleConnection(*socket*)

- transition: Registers the new *socket* with the **RealtimeAdapter**. Attaches handlers for `handleJoinGame`, `handleSubmitMove`, etc.
- output: `void`.
- exception: `SessionError` if the connection handshake fails.

handleJoinGame(*socket, data*)

- transition: 1. Calls `M4.verifyToken`(*data.authToken*) to get a `UserID`. 2. Retrieves the `GameSession` from **activeGames** using `data.gameID`. 3. Associates the *socket* with the `GameSession` and `UserID`.
- output: Calls `emitGameState`(*socket, currentState*) to send the current state to the joining player.
- exception: `SessionError` if `authToken` is invalid. `NotFound` if the `GameSession` does not exist in **activeGames**.

handleSubmitMove(*socket, data*)

- transition: Identifies `UserID` and `GameSession` from the *socket*. Calls `M19.validateAction`(*data, gameState*) to check legality. If valid, calls `M19.executeAction`(*data, gameState*) to get the `newGameState`. Updates the `GameSession` in **activeGames** with `newGameState`.
- output: Calls `broadcastGameState`(*gameID, newGameState*) to send the new state to all players in that session.
- exception: InvalidMove. NotYourTurn. SessionError.

registerGameSession(*gameID, session*)

- transition: Called by M3. Adds the new *session* to the **activeGames** map.
- output: `void`.
- exception: `LobbyError` if the *gameID* is already active.

7.4.5 Local Functions

None.

7.4.6 Considerations

- The secret of this module is the management of stateful connections and the mapping of `ClientSocket` handles to active `GameSession`s.
- This module acts as the server-authoritative state synchronizer, delegating all game logic to M19.

8 MIS of Matchmaking Module (M3)

8.1 Module

Matchmaking

8.2 Uses

- M4 Authentication Module
- M15 Game Engine Module
- M2 Real-time Gateway Module

8.3 Syntax

8.3.1 Exported Constants

None.

8.3.2 Exported Access Programs

Name	In	Out	Exceptions
createLobby	token: AuthToken, options: GameOptions	lobby: Lobby	LobbyError
joinLobby	token: AuthToken, lobbyID: LobbyID	void	LobbyFull, LobbyNotFound, AuthError
startMatch	token: AuthToken, lobbyID: LobbyID	gameID: GameID	LobbyNotFound, NotLobbyHost, AuthError, GameCreationError

8.4 Semantics

8.4.1 State Variables

- **lobbies**: Map¡LobbyID, Lobby¿ — Holds all active, waiting-for-players `Lobby` objects [cite: 639-640].

8.4.2 Environment Variables

None.

8.4.3 Assumptions

- Any `AuthToken` is verifiable by M4.
- Modules M15 and M2 are available when `startMatch` is invoked.

8.4.4 Access Routine Semantics

createLobby(*token, options*)

- transition: Calls `M4.verifyToken`(*token*) to get `UserID`. Generates a unique `LobbyID`. Creates a new `Lobby` object (setting `hostID = UserID`). Adds this new object to the **lobbies** map[cite: 636].
- output: Returns the newly created `Lobby`.
- exception: `AuthError`. `LobbyError`.

joinLobby(*token, lobbyID*)

- transition: Calls `M4.verifyToken`(*token*) to get `UserID`. Looks up the `Lobby` in **lobbies**. Verifies `Lobby.status == 'waiting'` and `Lobby.players.length < MAX_PLAYERS`. If valid, appends `UserID` to the `Lobby.players` array[cite: 636].
- output: `void`.
- exception: `LobbyFull`. `LobbyNotFound`. `AuthError`.

startMatch(*token, lobbyID*)

- transition: 1. Calls `M4.verifyToken`(*token*) to get `UserID`. 2. Looks up the `Lobby` in **lobbies** and verifies `UserID` is the host. 3. Calls `M15.createGame`(*lobby.players, options*) to get a new `GameState`[cite: 669]. 4. Creates a new `GameSession` object. 5. Calls `M2.registerGameSession`(*lobbyID, newGameSession*) to hand off the session to the real-time server [cite: 630-631]. 6. Removes the `Lobby` from the **lobbies** map[cite: 636].
- output: Returns the `GameID` (which may be the `LobbyID`).
- exception: `LobbyNotFound`. `NotLobbyHost`. `AuthError`. `GameCreationError`.

8.4.5 Local Functions

None.

8.4.6 Considerations

- The secret of this module is the `Lobby` data structure and the management of the **lobbies** map.
- This module bridges the stateless API (M1) and the stateful game session (M2) by handling the pre-game lobby state [cite: 636-637].

9 MIS of Authentication Module (M4)

9.1 Module

Authentication

9.2 Uses

- M5 Repository Module

9.3 Syntax

9.3.1 Exported Constants

None.

9.3.2 Exported Access Programs

Name	In	Out	Exceptions
registerUser	username: string, password: string	User	UserExists, ValidationError
loginUser	username: string, password: string	AuthToken	InvalidCredentials, ValidationError
verifyToken	token: AuthToken	UserID	TokenExpired, InvalidCredentials
manageGuestSession	-	AuthToken	SessionError

9.4 Semantics

9.4.1 State Variables

None. This module is stateless.

9.4.2 Environment Variables

- **CryptoLibrary**: An instance of the password hashing library (e.g., bcrypt).
- **JWT_SECRET_KEY**: The secret key used for signing and verifying JSON Web Tokens (AuthToken), read from a secure environment.

9.4.3 Assumptions

- The **CryptoLibrary** is properly configured.
- The **JWT_SECRET_KEY** is securely provided to the environment.
- Module M5 is available for database operations.

9.4.4 Access Routine Semantics

registerUser(*username, password*)

- transition: Validates *username* and *password* formats. Calls `M5.findUserByUsername(username)` to check for existence. Hashes and salts the *password* using **CryptoLibrary**. Calls `M5.createUser(username, hashedPassword)`.
- output: Returns the newly created `User` object.
- exception: `UserExists` if the username is already taken. `ValidationError` if inputs are malformed.

loginUser(*username, password*)

- transition: Calls `M5.findUserByUsername(username)` to retrieve the stored user hash. Compares the plaintext *password* with the stored hash using **CryptoLibrary**. If they match, generates a new `AuthToken` (JWT) signed with **JWT_SECRET_KEY** containing the `UserID`.
- output: Returns the newly generated `AuthToken`.
- exception: `InvalidCredentials` if the user is not found or the password does not match. `ValidationError` if inputs are malformed.

verifyToken(*token*)

- transition: Validates the *token*'s signature and expiration using **JWT_SECRET_KEY**. If valid, parses the `UserID` from the token payload.
- output: Returns the `UserID` extracted from the token.
- exception: `TokenExpired` if the token is past its expiry date. `InvalidCredentials` if the token signature is invalid or the token is malformed.

manageGuestSession( )

- transition: Generates a temporary `AuthToken` (JWT) with a special ”guest” `UserID` or a temporary unique identifier.
- output: Returns the `AuthToken` for the guest session.
- exception: `SessionError` if token generation fails.

9.4.5 Local Functions

None.

9.4.6 Considerations

- The secret of this module is the password hashing algorithm (bcrypt), salt generation, JWT structure, and the **JWT_SECRET_KEY**.
- M1 relies on this module for handling user authentication endpoints.
- M2 relies on `verifyToken` to authenticate WebSocket connections.

10 MIS of Repository Module (M5)

10.1 Module

Repository

10.2 Uses

- M22 Database Module

10.3 Syntax

10.3.1 Exported Constants

None.

10.3.2 Exported Access Programs

Name	In	Out	Exceptions
findUserByUsername	username: string	User	RecordNotFound, DatabaseConnectionError
createUser	data: UserData	User	UniqueConstraintViolation, DatabaseConnectionError
saveGameResult	result: GameResult	void	DatabaseConnectionError
getUserProfile	userID: UserID	UserProfile	RecordNotFound, DatabaseConnectionError
updateUserProfile	userID: UserID, data: UserProfile	UserProfile	RecordNotFound, DatabaseConnectionError
deleteUser	userID: UserID	void	RecordNotFound, DatabaseConnectionError

10.4 Semantics

10.4.1 State Variables

- **dbConnectionPool**: A connection pool managing active connections to the M22 database.

10.4.2 Environment Variables

- **DatabaseInstance (M21)**: The instance of the PostgreSQL database software (M22) on which this module executes queries.

10.4.3 Assumptions

- The **DatabaseInstance** is running and accessible.
- A database connection string is securely provided to the environment.
- The database schema (tables, columns, relations) has been initialized and matches the queries defined within this module.

10.4.4 Access Routine Semantics

findUserByUsername(*username*)

- transition: Acquires a connection from **dbConnectionPool**. Executes a SQL `SELECT` query to find the user by *username*.
- output: Returns the `User` object if found.
- exception: `RecordNotFound` if no user with *username* is found. `DatabaseConnectionError` if the query fails.

createUser(*data*)

- transition: Acquires a connection from **dbConnectionPool**. Executes a SQL `INSERT` query to create a new user with *data*.
- output: Returns the newly created `User` object (e.g., with database-generated ID).
- exception: `UniqueConstraintViolation` if the username already exists. `DatabaseConnectionError` if the query fails.

saveGameResult(*result*)

- transition: Acquires a connection from **dbConnectionPool**. Executes a SQL `INSERT` query to store the *result* in the game history table.
- output: `void`.
- exception: `DatabaseConnectionError` if the query fails.

getUserProfile(*userID*)

- transition: Acquires a connection from **dbConnectionPool**. Executes a SQL `SELECT` query to retrieve the user's profile based on *userID*.

- output: Returns the `UserProfile` object.
- exception: `RecordNotFound` if *userID* is not found. `DatabaseConnectionError` if the query fails.

updateUserProfile(*userID, data*)

- transition: Acquires a connection from **dbConnectionPool**. Executes a SQL `UPDATE` query to modify the user's profile matching *userID* with new *data*.
- output: Returns the updated `UserProfile` object.
- exception: `RecordNotFound` if *userID* is not found. `DatabaseConnectionError` if the query fails.

deleteUser(*userID*)

- transition: Acquires a connection from **dbConnectionPool**. Executes a SQL `DELETE` query to remove the user matching *userID*.
- output: `void`.
- exception: `RecordNotFound` if *userID* is not found. `DatabaseConnectionError` if the query fails.

10.4.5 Local Functions

None.

10.4.6 Considerations

- The secret of this module is the database schema, all SQL queries, and connection pooling.
- Other modules (M1, M4) are completely unaware of SQL. They call abstract functions like `getUserProfile`.
- If the database is migrated from PostgreSQL (M22) to another system, only M5 needs to be rewritten; all other modules remain unchanged.

11 MIS of Audit Module (M6)

11.1 Module

Audit

11.2 Uses

- M20 Operating System Module

11.3 Syntax

11.3.1 Exported Constants

None.

11.3.2 Exported Access Programs

Name	In	Out	Exceptions
log.info	message: string	void	LogWriteError
log.warn	message: string	void	LogWriteError
log.error	message: string	void	LogWriteError

11.4 Semantics

11.4.1 State Variables

- **loggerInstance**: An instance of the configured logging library (e.g., Winston).

11.4.2 Environment Variables

- **LogStorage**: The destination for log output, typically a file on the M20 filesystem.

11.4.3 Assumptions

- The **loggerInstance** is successfully initialized when the module is loaded.
- The **LogStorage** (filesystem) provided by M20 is writable.

11.4.4 Access Routine Semantics

log.info(*message*)

- transition: Uses the **loggerInstance** to format the *message* as an 'info' level entry (adhering to the hidden log format) and write it to **LogStorage**.

- output: `void`.
- exception: `LogWriteError` if writing to **LogStorage** fails.

log.warn(*message*)

- transition: Uses the **loggerInstance** to format the *message* as a 'warn' level entry and write it to **LogStorage**.
- output: `void`.
- exception: `LogWriteError` if writing to **LogStorage** fails.

log.error(*message*)

- transition: Uses the **loggerInstance** to format the *message* as an 'error' level entry and write it to **LogStorage**.
- output: `void`.
- exception: `LogWriteError` if writing to **LogStorage** fails.

11.4.5 Local Functions

None.

11.4.6 Considerations

- The secret of this module is the log format, the storage location (e.g., file path), and the log retention policy.
- This module is used by other backend modules (M1, M2, M4, M5) to log important system events for debugging and security auditing.

12 MIS of Real-time Client Module (M7)

12.1 Module

Real-time Client

12.2 Uses

- M2 Real-time Gateway Module

12.3 Syntax

12.3.1 Exported Constants

None.

12.3.2 Exported Access Programs

Name	In	Out	Exceptions
connect	-	void	ConnectionFailed
disconnect	-	void	
on	eventName: 'gameStateUpdate', callback: (state) =¿ void	void	
emit	eventName: 'submitMove', move: Move	void	ConnectionFailed

12.4 Semantics

12.4.1 State Variables

- **socket**: Socket — The `Socket.io-client` instance.
- **isConnected**: bool — Flag indicating the connection status.

12.4.2 Environment Variables

- **BrowserRuntime (M21)**: The client's web browser environment providing WebSocket APIs.

12.4.3 Assumptions

- The M2 server is running and its URL is accessible to the client.
- The browser environment (M21) supports WebSockets.

12.4.4 Access Routine Semantics

connect( )

- transition: Initializes and establishes the WebSocket connection to M2. Sets **socket** to the new instance and **isConnected** to `true` on success.
- output: `void`.
- exception: `ConnectionFailed` if the connection times out or is rejected.

disconnect( )

- transition: Closes the active WebSocket connection. Sets **isConnected** to `false` and **socket** to `null`.
- output: `void`.

on(*eventName, callback*)

- transition: Registers an event listener on the **socket** instance. When M2 emits an event matching *eventName* (e.g., 'gameStateUpdate'), the *callback* is invoked with the data payload.
- output: `void`.

emit(*eventName, move*)

- transition: Serializes and sends the *move* data to the M2 server over the **socket** connection, under the *eventName* (e.g., 'submitMove').
- output: `void`.
- exception: `ConnectionFailed` if **isConnected** is `false`.

12.4.5 Local Functions

None.

12.4.6 Considerations

- The secret of this module is the WebSocket connection state and reconnection logic.
- It is the client-side counterpart to M2.
- UI modules (e.g., M11, M12) use this module to receive state updates and send user actions.

13 MIS of Application Shell Module (M8)

13.1 Module

Application Shell

13.2 Uses

- M21 Browser Runtime Module
- M9 Authentication Client Module
- M10 Lobby View Module
- M11 Game Board View Module
- M14 Profile View Module

13.3 Syntax

13.3.1 Exported Constants

None.

13.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
Render	props: ReactProps	JSX.Element	RouteNotFound

13.4 Semantics

13.4.1 State Variables

- **currentUser**: User — null — Stores the state of the currently logged-in user.
- **currentRoute**: string — The active route from the browser's URL.

13.4.2 Environment Variables

- **BrowserRuntime (M21)**: The browser environment providing the DOM for rendering and the URL History API for routing.

13.4.3 Assumptions

- The React library is loaded in the M21 environment.
- The browser supports the History API.
- Modules M9, M10, M11, and M14 are available to be rendered as children.

13.4.4 Access Routine Semantics

Render(*props*)

- transition: Reads the URL path from the **BrowserRuntime (M20)** to update **currentRoute**. Reads the authentication status to update **currentUser**. Renders the global layout (header, footer). Selectively renders a child module (M9, M10, M11, or M14) based on **currentRoute** and **currentUser**.
- output: Returns a React Element (`JSX.Element`) for the **BrowserRuntime (M20)** to render to the DOM.
- exception: `RouteNotFound` if **currentRoute** does not match any entry in the application's routing table.

13.4.5 Local Functions

None.

13.4.6 Considerations

- The secret of this module is the application routing table and the global layout structure.
- This module acts as a controller view, deciding which page (M10, M11, M14) to display based on URL and authentication state.

14 MIS of Authentication Client Module (M9)

14.1 Module

Authentication Client

14.2 Uses

- API Module M1
- Browser Runtime Module M21 (Client Runtime)

14.3 Syntax

14.3.1 Exported Constants

None.

14.3.2 Exported Access Programs

Name	In	Out	Exceptions
handleLogin	-	void	AuthUIError
handleSignup	-	void	AuthUIError
handleLogout	-	void	
Render	props: ReactProps	JSX.Element	

14.4 Semantics

14.4.1 State Variables

- **username**: string — Stores the value from the username input field.
- **password**: string — Stores the value from the password input field.
- **isLoading**: bool — True if an API request (to M1) is in progress.
- **error**: string — Stores error messages from M1 (e.g., ”Invalid credentials”).

14.4.2 Environment Variables

- **BrowserRuntime (M21)**: The browser environment providing DOM rendering and storage.
- **AuthStorage**: The client-side storage mechanism (e.g., `localStorage`) used to persist the `AuthToken`.

14.4.3 Assumptions

- Module M1's authentication endpoints are available.
- This module is rendered by M8 (Application Shell).

14.4.4 Access Routine Semantics

handleLogin( )

- transition: Sets **isLoading** to `true`. Reads **username** and **password** from state. Calls `M1.login`. On success, stores the returned `AuthToken` in **AuthStorage**, sets **isLoading** to `false`, and updates global auth state. On failure, sets **isLoading** to `false` and populates **error**.
- output: `void`.
- exception: `AuthUIError` (represented in the **error** state) if M1 fails.

handleSignup( )

- transition: Sets **isLoading** to `true`. Reads **username** and **password**. Calls `M1.signup`. Manages success or failure similar to `handleLogin`.
- output: `void`.
- exception: `AuthUIError` (represented in the **error** state) if M1 fails (e.g., user exists).

handleLogout( )

- transition: Removes the `AuthToken` from **AuthStorage**. Updates global auth state (e.g., sets `currentUser` to `null`).
- output: `void`.

Render(*props*)

- transition: Reads all **State Variables** to determine UI.
- output: Returns a `JSX.Element` containing login/signup forms, inputs, and buttons. UI reflects **isLoading** (e.g., spinner) and **error** (e.g., error message) states.

14.4.5 Local Functions

None.

14.4.6 Considerations

- The secret of this module is how and where the `AuthToken` is stored on the client (e.g., `localStorage` vs. cookie).
- This module is responsible for both the UI of the forms and the client-side logic of communicating with M1.

15 MIS of Lobby View Module (M10)

15.1 Module

Lobby View

15.2 Uses

- M1 API Module
- M7 Real-time Client Module
- M21 Browser Runtime Module

15.3 Syntax

15.3.1 Exported Constants

None.

15.3.2 Exported Access Programs

Name	In	Out	Exceptions
Render	props: ReactProps	JSX.Element	
handleCreateGame	-	void	CreateGameError
handleJoinGame	lobbyID: LobbyID	void	JoinGameError

15.4 Semantics

15.4.1 State Variables

- **lobbiesList**: Lobby[] — An array of available game lobbies.
- **selectedLobby**: LobbyID — null — The ID of the lobby currently selected in the UI.
- **isLoading**: bool — True if a create or join operation is in progress.

15.4.2 Environment Variables

- **BrowserRuntime (M21)**: The browser environment providing DOM rendering.

15.4.3 Assumptions

- Modules M1 and M7 are available and configured.
- This module is rendered by M8 (Application Shell).

15.4.4 Access Routine Semantics

Render(*props*)

- transition: Reads **lobbiesList**, **selectedLobby**, and **isLoading** from state.
- output: Returns a `JSX.Element` that renders the UI for listing, creating, and joining game lobbies. Renders a loading indicator if **isLoading** is true.

handleCreateGame( )

- transition: Sets **isLoading** to `true`. Calls `M1.createGame` to create a new lobby. On success, receives a `newLobbyID` and calls `handleJoinGame(newLobbyID)`.
- output: `void`.
- exception: `CreateGameError` (displayed in UI) if the M1 call fails.

handleJoinGame(*lobbyID*)

- transition: Sets **isLoading** to `true`. Calls `M7.emit('joinGame', { lobbyID: lobbyID, ... })`. On success, the M7/M2 connection will trigger a state change that M8 will use to render M11.
- output: `void`.
- exception: `JoinGameError` (displayed in UI) if M7 fails to join.

15.4.5 Local Functions

None.

15.4.6 Considerations

- The secret of this module is the UI layout for displaying, creating, and joining games.
- It coordinates user actions, calling M1 for lobby creation and M7 for joining a real-time session.

16 MIS of Game Board View Module (M11)

16.1 Module

Game Board View

16.2 Uses

- M21 Browser Runtime Module
- M12 Move Controller Module

16.3 Syntax

16.3.1 Exported Constants

None.

16.3.2 Exported Access Programs

Name	In	Out	Exceptions
Render	props: ReactProps	JSX.Element	None

16.4 Semantics

16.4.1 State Variables

- **clientGameState**: GameState — The current game state object (hands, deck, discard pile).
- **validMoves**: Card[] — An array of cards in the player's hand that are legal to play.

16.4.2 Environment Variables

- **BrowserRuntime (M21)**: The browser environment providing DOM rendering and CSS.

16.4.3 Assumptions

- This module is rendered by M8 when a game is active.
- The **clientGameState** and **validMoves** are provided (likely as props).
- Event handlers from M12 are attached to the rendered elements.

16.4.4 Access Routine Semantics

Render(*props*)

- transition: Reads **clientGameState** and **validMoves** from state/props.
- output: Returns a `JSX.Element` that renders the main game interface, including the player's hand, the discard pile, and the deck. It visually highlights any cards in the hand that are also present in the **validMoves** list.

16.4.5 Local Functions

None.

16.4.6 Considerations

- The secret of this module is the DOM/CSS structure and animation logic used to render the game board.
- This is primarily a "dumb" rendering component; it displays state and delegates user input handling to M12.

17 MIS of Move Controller Module (M12)

17.1 Module

Move Controller

17.2 Uses

- Real-time Client Module (M7)
- Browser Runtime Module (M21)

17.3 Syntax

17.3.1 Exported Constants

None.

17.3.2 Exported Access Programs

Routine Name	In	Out	Exceptions
handlePlayCard	card: Card	void	InvalidMoveUI
handleDrawCard	-	void	InvalidMoveUI

17.4 Semantics

17.4.1 State Variables

- **selectedCard**: Card
- **uiFeedback**: string

17.4.2 Environment Variables

- **BrowserRuntime (M21)**: Provides user input events from the UI.

17.4.3 Assumptions

- Module M7 is connected.
- These access programs are bound to UI elements rendered by M11.

17.4.4 Access Routine Semantics

handlePlayCard(*card*)

- transition: Performs client-side pre-validation. If invalid, sets **uiFeedback**. If valid, calls `M7.emit('submitMove', { action: 'play', payload: card })`.
- output: `void`.
- exception: `InvalidMoveUI` (captured in **uiFeedback** state).

handleDrawCard( )

- transition: Performs client-side pre-validation. If invalid, sets **uiFeedback**. If valid, calls `M7.emit('submitMove', { action: 'draw' })`.
- output: `void`.
- exception: `InvalidMoveUI` (captured in **uiFeedback** state).

17.4.5 Local Functions

- `clientSidePreValidation(...)`: Local logic to pre-check moves.

17.4.6 Considerations

- The secret of this module is the client-side input handling logic and pre-validation rules.
- It decouples the M11 view from the M7 client service.

18 MIS of Scoreboard View Module (M13)

18.1 Module

The Scoreboard View Module handles the graphical display of player scores, game progress, and ranking updates. It receives score data from the Scoring Module and displays it in real time.

18.2 Uses

- M17 Scoring Module
- M18 Base Conversion Module

18.3 Syntax

18.3.1 Exported Constants

None.

18.3.2 Exported Access Programs

Routine Name	**In**	**Out**	**Exceptions**
renderScores	ScoreList	None	None
updateScore	PlayerID, Integer	None	None
toggleBaseDisplay	None	None	None

18.4 Semantics

18.4.1 State Variables

- **playerScores**: mapping of PlayerID to score values.
- **currentBase**: numeric base currently used for display (decimal or dozenal).

18.4.2 Environment Variables

- UI rendering environment; display panel for real-time updates.

18.4.3 Assumptions

- The Scoring Module provides consistent and valid score data.

18.4.4 Access Routine Semantics

renderScores(*ScoreList*)

- transition: draws the scoreboard table using the latest score values.
- output: `None`.

updateScore(*PlayerID, Integer*)

- transition: refreshes the score of a single player and triggers a UI redraw.
- output: `None`.

toggleBaseDisplay( )

- transition: switches between base-10 and base-12 visual formats.
- output: `None`.

18.4.5 Local Functions

- `convertBase(value, base)`: converts a score integer into the appropriate base representation.

18.4.6 Considerations

- The scoreboard must remain visually synchronized and readable regardless of player count or base selection.

19 MIS of Profile View Module (M14)

19.1 Module

Profile View Module

19.2 Uses

- M9 Authentication Client Module
- M17 Scoring Module

19.3 Syntax

19.3.1 Exported Constants

None.

19.3.2 Exported Access Programs

Routine Name	In	Out	Exceptions
loadProfile	PlayerID	ProfileData	DatabaseReadError
updateProfile	PlayerID, ProfileData	None	DatabaseWriteError
renderProfile	ProfileData	None	None

19.4 Semantics

19.4.1 State Variables

- **currentProfile**: cached profile data of the active player.
- **sessionStats**: recent gameplay summary for quick access.

19.4.2 Environment Variables

- Backend database connection or local storage; user interface display frame.

19.4.3 Assumptions

- Each player has a unique identifier.
- Profile data is fetched before rendering.

19.4.4 Access Routine Semantics

loadProfile(*PlayerID*)

- transition: retrieves stored profile data from database or cache.
- output: `ProfileData`.

updateProfile(*PlayerID, ProfileData*)

- transition: commits new statistics or preferences to persistent storage.
- output: `None`.

renderProfile(*ProfileData*)

- transition: displays the user's avatar, username, and score summary.
- output: `None`.

19.4.5 Local Functions

- `formatStats()`: formats match statistics for display.

19.4.6 Considerations

- This module must protect user data integrity and minimize latency when loading or updating profile information.

20 MIS of Game Engine Module (M15)

20.1 Module

Game Engine

20.2 Uses

- M16 Rules Module
- M17 Scoring Module
- M19 Game Actions Module

20.3 Syntax

20.3.1 Exported Constants

None.

20.3.2 Exported Access Programs

Routine Name	In	Out	Exceptions
initializeGame	PlayerList	GameState	InvalidSetupException
processTurn	Action	GameState	InvalidActionException
checkWinCondition	GameState	Boolean	None
reshuffleDeck	None	None	EmptyDeckException

20.4 Semantics

20.4.1 State Variables

- **currentState**: current configuration of the game.
- **activePlayer**: player ID whose turn is in progress.
- **drawPile, discardPile**: sets of remaining and played cards.

20.4.2 Environment Variables

- Game state repository, player actions, and random seed generator.

20.4.3 Assumptions

- Each player performs one valid action per turn.
- Randomness is seeded for reproducibility.

20.4.4 Access Routine Semantics

initializeGame(*PlayerList*)

- transition: distributes cards, sets starting player, and creates the discard pile.
- output: `GameState`.

processTurn(*Action*)

- transition: validates and executes one action, then triggers scoring update.
- output: `GameState`.

checkWinCondition(*GameState*)

- output: evaluates if any player has no cards remaining.

reshuffleDeck( )

- transition: moves discard cards back into draw pile and randomizes order.
- output: `None`.

20.4.5 Local Functions

- `advanceTurn()`: calculates next player index.

20.4.6 Considerations

- The module must maintain consistency across all players and prevent race conditions during state transitions.

21 MIS of Rules Module (M16)

21.1 Module

Rules Module

21.2 Uses

- M15 Game Engine Module
- M17 Scoring Module

21.3 Syntax

21.3.1 Exported Constants

- **MAX_HAND_SIZE** = 10
- **BASE_DOZENAL** = 12

21.3.2 Exported Access Programs

Routine Name	**In**	**Out**	**Exceptions**
isValidMove	Card, Card	Boolean	None
isWildCard	Card	Boolean	None
getAllowedSuits	Card	SuitList	None
applyRuleVariant	GameState, VariantID	GameState	InvalidVariantException

21.4 Semantics

21.4.1 State Variables

- Current rule set identifier (classic or dozenal).

21.4.2 Environment Variables

- Access to game state and current discard card.

21.4.3 Assumptions

- Game state is valid when rules are applied.

21.4.4 Access Routine Semantics

isValidMove(*Card, Card*)

- output: returns true if the new card matches suit or rank or follows wild rules.

isWildCard(*Card*)

- output: determines if a card can change the active suit.

getAllowedSuits(*Card*)

- output: returns all valid suits for a wild card declaration.

applyRuleVariant(*GameState, VariantID*)

- transition: modifies the rule behavior based on selected configuration.
- output: `GameState`.

21.4.5 Local Functions

- `compareRanks(a,b)`: helper for rank matching.

21.4.6 Considerations

- Rules must remain modular and easy to extend for new variants without changing engine code.

22 MIS of Scoring Module (M17)

22.1 Module

Scoring Module

22.2 Uses

- M15 Game Engine Module
- M18 Base Conversion Module

22.3 Syntax

22.3.1 Exported Constants

- **WIN_BONUS** = 100
- **CARD_PENALTY** = 10

22.3.2 Exported Access Programs

Routine Name	In	Out	Exceptions
calculateScore	PlayerState	Integer	None
updateScores	GameState	ScoreList	None
getWinner	GameState	PlayerID	None

22.4 Semantics

22.4.1 State Variables

- Mapping of PlayerID to total scores.

22.4.2 Environment Variables

- Game state from Game Engine; base system from Base Conversion Module.

22.4.3 Assumptions

- Game Engine correctly identifies the end of a round.

22.4.4 Access Routine Semantics

calculateScore(*PlayerState*)

- output: computes total score for a player after round ends.

updateScores(*GameState*)

- transition: recalculates and persists the current scoreboard.
- output: `ScoreList`.

getWinner(*GameState*)

- output: returns the player ID with the highest score.

22.4.5 Local Functions

- `convertScoreToBase(score, base)`: converts numeric score into base-12 if required.

22.4.6 Considerations

- The scoring algorithm must remain deterministic and verifiable for test reproducibility.

23 MIS of Base Conversion Module (M18)

23.1 Module

Base Conversion Module

23.2 Uses

- M17 Scoring Module

23.3 Syntax

23.3.1 Exported Constants

- **DIGITS** = [0–9, A, B]

23.3.2 Exported Access Programs

Routine Name	In	Out	Exceptions
decimalToDozenal	Integer	String	None
dozenalToDecimal	String	Integer	InvalidFormatException

23.4 Semantics

23.4.1 State Variables

- None; stateless conversion utility.

23.4.2 Environment Variables

- None.

23.4.3 Assumptions

- Input values are within valid numerical range.

23.4.4 Access Routine Semantics

decimalToDozenal(*Integer*)

- output: converts base-10 integer to base-12 string representation.

dozenalToDecimal(*String*)

- output: parses base-12 string into a base-10 integer.

23.4.5 Local Functions

- `mapDigit(symbol)`: converts symbol to corresponding numeric value.

23.4.6 Considerations

- Module must handle both positive and zero values accurately; negative numbers optional.

24 MIS of Game Actions Module (M19)

24.1 Module

Game Actions Module

24.2 Uses

- M15 Game Engine Module
- M16 Rules Module

24.3 Syntax

24.3.1 Exported Constants

None.

24.3.2 Exported Access Programs

Routine Name	**In**	**Out**	**Exceptions**
createAction	ActionType, Parameters	Action	InvalidActionType
validateAction	Action, GameState	Boolean	InvalidMoveException
executeAction	Action, GameState	GameState	ActionExecutionError
undoAction	Action, GameState	GameState	None

24.4 Semantics

24.4.1 State Variables

- **pendingActions**: a queue of unexecuted player actions.
- **lastAction**: most recent action for rollback or replay.

24.4.2 Environment Variables

- Backend execution environment.

24.4.3 Assumptions

- Each action follows the command pattern and can be validated independently.
- The game engine (M15) ensures single-threaded execution for action safety.

24.4.4 Access Routine Semantics

createAction(*ActionType, Parameters*)

- output: constructs an action object from parameters (e.g., “play card 8 spade”).

validateAction(*Action, GameState*)

- output: checks if the action is allowed under current rules (by calling M16).

executeAction(*Action, GameState*)

- transition: applies changes to game state (by calling M15) and notifies observers.
- output: `GameState`.

undoAction(*Action, GameState*)

- transition: reverses the last applied change for testing or debugging.
- output: `GameState`.

24.4.5 Local Functions

- `serializeAction()`: converts an action into a string or JSON for replay logging.

24.4.6 Considerations

- This module improves maintainability by isolating gameplay logic into self-contained actions, enabling undo/redo and deterministic testing.

25 MIS of Operating System Module (M20)

25.1 Module

Operating System Module

25.2 Uses

None.

25.3 Syntax

25.3.1 Exported Constants

None.

25.3.2 Exported Access Programs

Routine Name	In	Out	Exceptions
scheduleTask	Function, Delay	None	None
readFile	Path	String	IOError
writeFile	Path, String	None	IOError

25.4 Semantics

25.4.1 State Variables

- System scheduler queue.

25.4.2 Environment Variables

- Host OS file system and process manager.

25.4.3 Assumptions

- OS provides basic thread safety and asynchronous task execution.

25.4.4 Access Routine Semantics

scheduleTask(*Function, Delay*)

- transition: executes a callback after a specified delay.
- output: `None`.

readFile(*Path*)

- output: retrieves content from a local file path.

writeFile(*Path, String*)

- transition: writes content to a file.
- output: `None`.

25.4.5 Local Functions

- `validatePath(path)`: checks file system accessibility.

25.4.6 Considerations

- This module ensures portability across Windows, macOS, and Linux.

26 MIS of Browser Runtime Module (M21)

26.1 Module

Browser Runtime Module

26.2 Uses

None.

26.3 Syntax

26.3.1 Exported Constants

None.

26.3.2 Exported Access Programs

Routine Name	In	Out	Exceptions
registerEvent	String, Callback	None	None
renderElement	HTMLElement	None	None
storeLocalData	Key, Value	None	StorageException

26.4 Semantics

26.4.1 State Variables

- Local storage cache, active event listeners.

26.4.2 Environment Variables

- Browser environment (HTML5, Web APIs).

26.4.3 Assumptions

- All browser APIs are available in the execution environment.

26.4.4 Access Routine Semantics

registerEvent(*String, Callback*)

- transition: binds a function to a specified DOM event.
- output: `None`.

renderElement(*HTMLElement*)

- transition: draws a UI element on the screen.
- output: `None`.

storeLocalData(*Key, Value*)

- transition: writes data into browser storage for persistence.
- output: `None`.

26.4.5 Local Functions

- `serialize(obj)`: converts objects into storable string format.

26.4.6 Considerations

- Must be compatible with modern browsers and responsive frameworks.

27 MIS of Database Module (M22)

27.1 Module

Database Module

27.2 Uses

None.

27.3 Syntax

27.3.1 Exported Constants

None.

27.3.2 Exported Access Programs

Routine Name	**In**	**Out**	**Exceptions**
connect	ConnectionString	Boolean	DatabaseConnectionError
query	SQLStatement	ResultSet	QueryError
insertRecord	Table, Data	Boolean	InsertError
updateRecord	Table, Data	Boolean	UpdateError

27.4 Semantics

27.4.1 State Variables

- **dbConnection**: current active database session.
- **cache**: optional in-memory data cache.

27.4.2 Environment Variables

- Database server or local SQLite environment.

27.4.3 Assumptions

- Database connection string is valid and accessible.

27.4.4 Access Routine Semantics

connect(*ConnectionString*)

- transition: establishes a session with the database server.
- output: `Boolean`.

query(*SQLStatement*)

- output: executes read operations and returns result sets.

insertRecord(*Table, Data*)

- transition: inserts a new entry into the specified table.
- output: `Boolean`.

updateRecord(*Table, Data*)

- transition: modifies existing records based on key values.
- output: `Boolean`.

27.4.5 Local Functions

- `sanitizeInput()`: prevents SQL injection attacks.

27.4.6 Considerations

- Database operations must remain atomic and logged to ensure integrity and traceability.

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.

28 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)