Module guide for UNO-Flip

Team24 unomaster
Mingyang Xu
Kevin Ishak
Jianhao Wei
Zain-Alabedeen Garada
Zheng Beng Liang

1. Introduction

Purpose of the Document

This document serves as a blueprint for the software architecture of the UNO-Flip. It outlines the fundamental design and structure of the system, enabling stakeholders to understand how the software components will interact and function as a cohesive whole. The document provides:

1.A clear breakdown of system modules.

2.Definitions of relationships between components and their responsibilities.

3.Guidance for developers, testers, and maintainers during the software lifecycle.

The target audience for this document includes:

1. Developers - To understand module responsibilities, interfaces, and dependencies for implementation.
2. Test Engineers - To design test cases aligned with module functionality and interaction.
3. Stakeholders - To gain an overview of how the system will meet their business and functional needs.
4. System Maintainers - To identify areas for extension or modification during future maintenance phases.

- Scope of the software architecture

- Relationship to related documents (e.g., SRS, design documents)

2. Software Architecture Overview

High-level architecture description

The UNO Flip Online Multiplayer Game is a real-time, cross-platform game designed for web platforms. The system adopts a client-server architecture, ensuring smooth real-time gameplay and state synchronization among multiple players. It supports global matchmaking as well as private lobbies for 2 to 4 players.

Frontend:

Provides a responsive user interface for players to interact with the game. It includes features like card selection, flipping, chat functionality, and real-time status updates.

Developed using Unity Hub for web platforms.

Database:

Stores user profiles, game history, and leaderboard data.

MongoDB is used for flexible and fast document-based data storage.

Backend:

Handles game logic, player matchmaking, room management, and real-time updates.

Built with Node.js and Express for scalable and efficient game state management.

Cloud Hosting:

The application is deployed on cloud platforms like AWS or Firebase for reliability, scalability, and low latency.

Decomposition of the system into modules

1. Game Logic Module

- **Responsibilities**:

Implements the core gameplay mechanics of **UNO Flip**, including:

Managing card rules (e.g., matching colors and numbers, flipping cards).

Turn management for players.

Validating player actions (e.g., whether a move is legal).

Handling special cards like Reverse, Skip, and Draw cards.

Maintains the state of the game, including the deck, discard pile, and each player's hand.

- **Implementation in Unity**:

Unity scripts (written in C#) will handle all game mechanics and rules.

Example classes:

CardManager: Handles the creation and flipping of cards.

GameStateManager: Tracks the current state of the game.

TurnManager: Manages the turn order and player actions.

2. Multiplayer Module

- **Responsibilities**:

Enables real-time multiplayer functionality for online gameplay.

Handles:

Player matchmaking.

Synchronization of game states across clients.

Network communication for player actions, such as playing a card or flipping the deck.

Ensures players remain synchronized even if one experiences network latency.

- **Implementation in Unity**:

Use Unity's **Netcode for GameObjects (NGO)** or third-party solutions like **Photon Unity Networking (PUN)** for real-time multiplayer support.

Example components:

LobbyManager: Manages matchmaking and room creation.

NetworkManager: Synchronizes game states between the server and clients.

PlayerSync: Ensures each player's actions are reflected across all devices.

3. UI Module

Responsibilities:

Provides a user-friendly interface for players to interact with the game.

Includes:

Displaying player hands, deck, and discard pile.

Real-time updates for turn indicators and game status.

In-game chat functionality for player communication.

Ensures responsive design and intuitive navigation.

Implementation in Unity:

Unity’s **UI Toolkit** or **Canvas** system for designing the interface.

Example UI elements:

HandDisplay: Shows the player’s cards in hand.

GameHUD: Displays turn indicators, chat box, and player scores.

MainMenu: Allows players to start a new game or join an existing match.

4. Asset Management Module

- **Responsibilities**:

Manages all visual and audio assets used in the game.

Includes:

2D/3D models for cards and other game objects.

Animations for flipping cards and other visual effects.

Background music and sound effects (e.g., shuffling, flipping, and playing cards).

- **Implementation in Unity**:

Organize assets within Unity's **Asset Management System**.

Use Unity's **Animator Controller** for animations.

Example assets:

Card Sprites: Represents the front and back sides of UNO Flip cards.

Audio Clips: Sounds for card shuffling, game notifications, etc.

Backgrounds: Themed environments for the game board.

5. Backend/Server Module

- **Responsibilities**:

Manages persistent data and ensures smooth communication between clients in a multiplayer environment.

Handles:

Saving player profiles, game history, and leaderboard data.

Real-time data updates for multiplayer games.

Stores data for reconnecting players after disconnection.

- **Implementation**:

Use a lightweight backend solution like **Firebase Realtime Database** or a custom Unity server with **Unity Multiplayer Services**.

Example components:

DatabaseManager: Stores player profiles and game records.

SessionManager: Tracks active game sessions and players.

LeaderboardManager: Updates and retrieves player rankings.

6. AI Module (Optional)

- **Responsibilities**:

Provides AI-controlled opponents for single-player or mixed multiplayer modes.

Ensures AI players can:

Make valid moves.

Simulate strategic behavior based on game context.

- **Implementation in Unity**:

Unity scripts for AI decision-making, using a state machine or behavior tree.

Example classes:

AIPlayer: Controls the AI's logic for playing cards.

AIStrategy: Implements varying difficulty levels for AI players.

Application of the Single-Responsibility Principle

The system adheres to the Single-Responsibility Principle by ensuring each module focuses on a distinct area of functionality:

1. **Game Logic Module**: Handles gameplay mechanics independently from other aspects.
2. **Multiplayer Module**: Focuses solely on enabling real-time interaction between players.
3. **UI Module**: Manages the display and interaction, without interfering with game logic or networking.
4. **Asset Management Module**: Handles visual and audio resources without impacting game logic or UI.
5. **Backend/Server Module**: Manages data storage and communication independently from other modules.
6. **AI Module**: Handles AI logic, separate from real-time multiplayer and game mechanics.

3. Module Design

3.1 Module Breakdown

1. Game Logic Module

- **Description**:
  - Handles all core game mechanics, including card rules, turn management, and game state updates.
- **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.
- **Environment Variables**:
  - `maxPlayers`: Maximum number of players allowed in a game.
  - `flipEnabled`: Boolean to toggle the flip functionality.
- **Exported Functions**:
  - `validateMove(playerId, card)`: Checks if a move is valid.
  - `endTurn(playerId)`: Ends the current player's turn and starts the next.
  - `shuffleDeck()`: Randomizes the card deck.
  - `drawCard(playerId)`: Adds a card to the specified player’s hand.

2. Multiplayer Module

- **Description**:
  - Ensures real-time communication between players and manages game synchronization.
- **State Variables**:
  - `activeGames`: Tracks all ongoing game sessions.
  - `connectedPlayers`: List of currently connected players.
- **Environment Variables**:
  - `serverIP`: IP address of the game server.
  - `timeoutLimit`: Time limit for a player to respond during their turn.
- **Exported Functions**:
  - `createGameRoom(playerId, roomSettings)`: Creates a new game room.
  - `joinGameRoom(playerId, roomId)`: Adds a player to an existing room.
  - `broadcastUpdate(gameId, update)`: Sends game state updates to all players in a room.

3. UI Module

- **Description**:
  - Manages the user interface, ensuring players can interact with the game effectively.
- **State Variables**:
  - `displayedCards`: Tracks the cards currently visible to the player.
  - `turnIndicator`: Highlights the current player's turn.
- **Environment Variables**:
  - `theme`: Current visual theme of the game (e.g., light/dark mode).
  - `screenSize`: Resolution of the player's device.
- **Exported Functions**:
  - `updateCardDisplay(playerId, cards)`: Updates the player's visible hand.
  - `showTurnIndicator(playerId)`: Highlights the active player.
  - `displayMessage(message)`: Shows notifications or chat messages.

---

4. Asset Management Module

- **Description**:
  - Handles all visual and audio assets, ensuring smooth integration into the game.
- **State Variables**:
  - `cardSprites`: Stores front and back images for each card type.
  - `soundEffects`: Stores audio clips for actions like card flips or notifications.

- **Environment Variables**:
    - `assetPath`: Directory where all assets are stored.
- **Exported Functions**:
    - `loadAsset(assetName)`: Fetches the required asset for use.
    - `playSound(effectName)`: Plays a specified sound effect.

---

5. Backend/Server Module

- **Description**:
    - Manages data storage, including player profiles and game history.
- **State Variables**:
    - `userProfiles`: Stores player information, including win/loss statistics.
    - `leaderboard`: Tracks global rankings.
- **Environment Variables**:
    - `databaseURI`: URI for connecting to the database.
- **Exported Functions**:
    - `saveGameResult(gameData)`: Stores the results of a completed game.
    - `fetchLeaderboard()`: Retrieves the current leaderboard.

3.2 Module Relationships

Dependencies and Interactions:

- The **Game Logic Module** depends on the **Multiplayer Module** to broadcast game state changes to all players.
- The **UI Module** communicates with the **Game Logic Module** to fetch and display the current game state.
- The **Asset Management Module** provides resources (e.g., card sprites, sounds) to the **UI Module**.
- The **Multiplayer Module** relies on the **Backend Module** to authenticate players and save game results.

Mapping to SRS Requirements:

- **Requirement 1**: Real-time multiplayer functionality → Handled by the **Multiplayer Module**.
- **Requirement 2**: Accurate gameplay mechanics → Implemented in the **Game Logic Module**.
- **Requirement 3**: User-friendly interface → Fulfilled by the **UI Module**.
- **Requirement 4**: Player profiles and leaderboard → Managed by the **Backend Module**.

3.3 Likely and Unlikely Changes

Anticipated Changes:

1. **Game Logic Module**:
   - Adding new game modes (e.g., timed rounds, tournament play).
   - Modifying card rules (e.g., introducing custom cards or rules).
2. **UI Module**:
   - Redesigning the interface for better accessibility or cross-platform compatibility.
   - Adding support for additional languages.
3. **Multiplayer Module**:
   - Supporting larger game rooms with more than 8 players.
   - Enhancing matchmaking algorithms.

Stable Areas:

1. **Game Logic Module**:
   - Core mechanics like card matching and turn management are unlikely to change.
2. **Backend Module**:
   - Data storage structure for player profiles and game history is stable.
3. **Asset Management Module**:
   - Asset formats (e.g., sprite and sound formats) are not expected to evolve.

3.4 Secrets

1. **Game Logic Module**:
   - The randomization algorithm used for shuffling the deck is encapsulated to prevent predictable outcomes.
   - The logic for validating player moves is hidden to ensure game fairness.
2. **Multiplayer Module**:
   - Network synchronization algorithms and latency compensation methods are kept internal to prevent exploitation.
3. **Backend Module**:
   - Database encryption and authentication mechanisms are encapsulated for security purposes.
4. **Asset Management Module**:
   - Asset compression techniques and preloading mechanisms are hidden to optimize performance.

4. Architectural Diagrams

4.1 UML Package Diagrams

- **Purpose**: Illustrates the modular structure of the system, showing the relationships and dependencies between different modules.
- **Diagram Elements**:
  - **Packages**: Represent major system modules such as `Game Logic`, `UI`, `Multiplayer`, `Asset Management`, and `Backend`.
  - **Dependencies**: Show interactions between modules, such as:
    - `Game Logic` depends on `Multiplayer` for real-time updates.
    - `UI` depends on `Game Logic` to display game state.
  - Example Structure:
    - `UI` → `Game Logic` → `Multiplayer` → `Backend` → `Database`.

4.2 UML Class Diagrams

- **Purpose**: Defines the structure of key classes within each module and their relationships.
- **Key Classes**:
  - **Game Logic Module**:
    - `Card`: Represents a single card with properties (color, type, flip side).
    - `Deck`: Manages the shuffling, drawing, and discard pile.
    - `GameStateManager`: Tracks the overall game state (e.g., current turn, active players).
  - **Multiplayer Module**:
    - `PlayerConnection`: Represents a player's network connection.
    - `RoomManager`: Manages player rooms and game sessions.
  - **UI Module**:
    - `GameBoard`: Handles the visual representation of the game.
    - `PlayerHUD`: Displays player-specific information (e.g., cards, turn indicator).

4.3 State Machine Diagrams

- **Purpose**: Represents the lifecycle of critical components, such as game states or player interactions.
- **Example**:
  - **Game State Machine**:
    - States: `Waiting for Players` → `Game In Progress` → `Game Paused` → `Game Over`.
    - Transitions:
      - `Waiting for Players` → `Game In Progress`: Triggered when the required number of players join.
      - `Game In Progress` → `Game Over`: Triggered when a player wins or the deck is exhausted.

5. Design Details

5.1 Interface Design

- Figma or wireframe designs of user interfaces

- User interaction flows and state transitions

5.2 Database Design

- Entity-relationship diagrams (ERD)

- Database schema and constraints

5.3 Communication Protocols

- API interface definitions

- Data exchange formats and protocols used

6. Design Patterns

List of Design Patterns

Here are the design patterns that can be applied to the **UNO Flip** project:

1. **Singleton Pattern**:
    - **Purpose**: Ensure a single instance of certain classes that manage global states or resources.
    - **Application**:
        - **GameStateManager**: The game state should be managed centrally, ensuring that all players interact with the same instance.
        - **AssetManager**: Handles the loading and management of assets such as card images and sounds, avoiding redundant resource allocation.
2. **Observer Pattern**:
    - **Purpose**: Allow objects to subscribe to changes in a subject, ensuring real-time updates.
    - **Application**:
        - **UI Module**: Observes changes in the `GameStateManager` to update the game board and player HUD dynamically.
        - **Multiplayer Module**: Observes player actions and broadcasts updates to all connected players.
3. **Factory Pattern**:
    - **Purpose**: Create objects without specifying their exact classes.
    - **Application**:

        - **CardFactory**: Dynamically generates different types of cards (e.g., Number Card, Reverse Card, Flip Card) based on input parameters.
4. **State Pattern**:
    - **Purpose**: Encapsulate varying behavior for an object based on its current state.
    - **Application**:
        - **GameState**: Represents different states of the game (e.g., Waiting for Players, Player Turn, Game Over), allowing flexible transitions between states.
5. **Command Pattern**:
    - **Purpose**: Encapsulate requests as objects, allowing undo/redo functionality or deferred execution.
    - **Application**:
        - **PlayerAction**: Encapsulates actions like "Play Card", "Draw Card", and "Flip Deck", which can be executed, stored, or rolled back.

7. External Libraries and Wrappers

Modules that Rely on External Libraries

1. **Multiplayer Module**:
    - **Library**: Unity’s **Netcode for GameObjects** or **Photon Unity Networking (PUN)**.
    - **Purpose**:
        - Real-time synchronization of player actions.
        - Managing matchmaking and lobby creation.
2. **UI Module**:
    - **Library**: Unity's **UI Toolkit**.
    - **Purpose**:
        - Building responsive and interactive user interfaces.
        - Managing transitions, animations, and event handling.
3. **Asset Management Module**:
    - **Library**: Unity's built-in **Resource Management System**.
    - **Purpose**:
        - Efficiently loading and unloading assets like card sprites and background music.
        - Reducing memory usage during gameplay.
4. **Backend/Database Module**:
    - **Library**: **Firebase Realtime Database** or **MongoDB Atlas**.
    - **Purpose**:
        - Storing player profiles, game sessions, and leaderboard data.
        - Providing fast queries for multiplayer interactions.

Wrappers Created to Adapt Libraries

1. **NetworkWrapper**:

    - **Purpose**: Abstracts the underlying multiplayer library (e.g., Photon or Netcode) to provide a unified interface for managing rooms and player connections.
    - **Key Methods**:
        - `createRoom(roomName)`: Creates a new game room.
        - `joinRoom(roomName)`: Joins an existing game room.
        - `sendGameState(data)`: Sends the current game state to all players.
2. **DatabaseWrapper**:
    - **Purpose**: Simplifies interactions with the database, providing higher-level methods for CRUD operations.
    - **Key Methods**:
        - `saveGameSession(sessionData)`: Saves game session details.
        - `getLeaderboard()`: Retrieves the top players' rankings.
3. **AssetLoader**:
    - **Purpose**: Provides a simple interface for loading and caching assets during the game.
    - **Key Methods**:
        - `loadSprite(assetName)`: Loads a sprite by its name.
        - `playAudio(audioName)`: Plays a specific sound effect or background music.

8. Error Handling

- **Common failure scenarios and mitigation strategies**
    - Incorrect card rules enforced due to logic errors in the game engine. Mitigation: Implement unit tests for game rules to ensure correct behavior.
    - Server or client crashes during online multiplayer games. Mitigation: Introduce retry mechanisms and periodic save points to recover the game state.
    - Invalid user inputs, such as selecting an incorrect card or skipping turns. Mitigation: Implement comprehensive input validation and error messages to guide users.
- **Exception-handling mechanisms and recovery processes**
    - Utilize try-catch blocks to handle runtime exceptions in the game logic, such as null references or invalid array accesses.
    - Implement a rollback mechanism to restore the game state to the last valid state upon encountering an error.
    - Log all critical errors with detailed stack traces for debugging purposes, and notify users of recoverable errors with appropriate prompts.

9. Future Extensions and Maintenance

- **Plans for expanding functionality**

    - Add new gameplay modes, such as tournament-style matches or cooperative play.
    - Introduce customizable card decks, allowing users to design their own cards with unique effects.
    - Support cross-platform play between desktop and mobile versions of the game.
- **Areas for potential refactoring or improvement**
    - Optimize the card-rendering engine to improve performance on low-end devices.
    - Refactor the game logic to separate core functionalities from UI elements, enhancing modularity and maintainability.
    - Simplify the multiplayer networking code to reduce latency and improve scalability.
- **Guidelines for maintaining the system over time**
    - Regularly review and update the codebase to ensure compatibility with the latest versions of development tools and libraries.
    - Establish a version control and continuous integration (CI) pipeline for automated testing and deployment.
    - Document new features, bug fixes, and architectural changes in a centralized repository for easy reference.