Architecture Documentation

Project: FunFlip Game

Authors:

Ashish Ghaskata  
Krishna Raj Bhandari

Suraj Bhatta

Mohammad Adnan Khan

**Documentname:** AB-Funflip Game-Architecture Doc

**Version:** 1.0

**Creation date:** 20.06.2025

**File:** <file location>

Table of Contents

[Table of Figure 1](#_Toc201629462)

[2.1 Description of System Architecture 2](#_Toc201629463)

[2.1.1 Functional Requirements and Non Functional Requirements 2](#_Toc201629464)

[1. List of Requirements 2](#_Toc201629465)

[2.1.2 Prioritization of Non-Functional Requirements 3](#_Toc201629466)

[2.1.3 Architectural Principles 4](#_Toc201629467)

[2.1.4 Interfaces 4](#_Toc201629468)

[2.1.5 Big picture of the system Architecture 6](#_Toc201629469)

[2.2 System Design 8](#_Toc201629470)

[2.2.1 System decomposition 8](#_Toc201629471)

[2.2.2 Design Decisions 8](#_Toc201629472)

[2.2.3 Design Alternatives Considered. 9](#_Toc201629473)

[2.2.4 Documented Reuse of Components in FunFlips 9](#_Toc201629474)

[Activity Diagram 10](#_Toc201629475)

[2.3 Human Machine interface 12](#_Toc201629476)

[2.3.1 Requirements for the H-M Interface 12](#_Toc201629477)

[2.3.2 Design Principles and Style Guide 12](#_Toc201629478)

[2.3.3 Interaction Modeling 12](#_Toc201629479)

# Table of Figure

[Figure 1 High Level System Architecture 6](#_Toc201629396)

[Figure 2 Activity Diagram 9](#_Toc201629397)

[Figure 3 Domain Data Model 10](#_Toc201629398)

[Figure 4 Interaction Modelling 12](#_Toc201629399)

# 2.1 Description of System Architecture

## 2.1.1 Functional Requirements and Non-Functional Requirements

1. List of Requirements

Below are the identified functional and non-functional requirements for the FunFlip educational game. Each requirement is categorized and marked as 'Must' or 'Can' based on priority.

|  |  |  |
| --- | --- | --- |
| **Requirement** | **Must/Can** | **Category** |
| Game must allow category selection (Animals, Fruits, Vegetables) | Must | FR, UX |
| Game must support 3 difficulty levels (Easy, Medium, Hard) | Must | FR |
| Players must be able to flip cards to find matching pairs | Must | FR |
| Audio must play when a card is flipped | Must | FR, UX, Q |
| Unmatched cards must flip back after a delay | Must | FR |
| Game must display turn count | Must | FR |
| Happy sound must play when match is successful | Must | FR, UX |
| Players must see options to replay, return to menu, or quit | Must | FR, UX |
| Voice feedback must differ by category (e.g., animal sounds) | Can | FR, Q |
| The UI must be simple and colourful for ages 4–6 | Must | NFR, UX, Q |
| The game must run offline | Must | NFR, L |
| Sound/music must be toggleable in options | Can | FR, UX |
| Game must respond within 0.5 seconds after user actions | Must | NFR, Q |
| Game must not collect personal data | Must | NFR, L |
| Each component must be modular in Godot | Can | T, NFR |

## 2.1.2 Prioritization of Non-Functional Requirements

The top 3 quality attributes identified for FunFlip are Usability, Accessibility, and Performance. The following table describes how each attribute is defined, implemented, and measured.

Top 3 Quality Attributes

1. Usability
2. Accessibility
3. Performance

Operationalization of Quality Characteristics

|  |  |  |  |
| --- | --- | --- | --- |
| Quality Attribute | Objective (Definition) | How to Achieve It | Measurement Criteria |
| Usability | The game must be simple and navigable by children aged 4–6 | Use large buttons, few navigation steps | Can reach game within 3 clicks; no text reading needed |
| Accessibility | The game must provide voice feedback and sounds to aid learning | Use audio cues when flipping or matching cards | 100% of cards play sound; mute option available |
| Performance | The game must feel responsive to user actions | Optimize Godot scenes and animation speed | Card flip response ≤ 0.5s; zero crashes during tests |

## 2.1.3 Architectural Principles

These are the general rules we followed in designing Fun Flips:

**1️. Strict Layered Architecture**  
The system is divided into clear layers (UI → Scene Loader → Game Logic → Data → Services). Each layer talks only to the one below.

**2. Separation of Concerns & Single Responsibility**  
Each part does just one job: Scene Loader navigates, Game Manager runs gameplay, Data Manager handles data.

**3️. Low Coupling & High Cohesion**  
Components are focused on their task and talk via clean APIs or signals, not by accessing each other’s data.

**4️. Open for Extension, Closed for Modification**  
New content (cards, levels, sounds) can be added without changing existing code.

**5️. Keep It Simple (KISS)**  
The design avoids unnecessary complexity — one reusable Card.tscn, minimal singletons.

**6️. Cross-Cutting Concerns Centralised**  
Usability, accessibility, and performance are addressed across all layers through simple UI, centralized audio handling, preloaded assets, and efficient logic — ensuring consistent quality system-wide.

## 2.1.4 Interfaces

**1. SceneLoader**

- Purpose: Manages scene transitions across the game.  
- Methods exposed:

* Show start screen
* Show category selection screen
* Show level selection screen (with chosen category)
* Show game screen (with chosen category and level)
* Show completion screen
  1. **GameManager**

- Purpose: Controls the core game flow and rules.  
- Methods exposed:

* Start a new game (with category and level)
* Handle card flip
* Pause/resume the game
* Get current score
* End game  
  1. **DataManage**
* Purpose: Provides game data and manages persistence.
* Methods exposed:
* Get available categories
* Get cards for a selected category
* Load saved game progress
* Save game progress  
  1. **AudioPlayer**
* Purpose: Manages playing of sound effects and audio settings.
* Methods exposed:
* Play a sound effect by name
* Set volume level
* Mute/unmute audio
  1. **UICallbacks**
* Purpose: Represents events that the UI triggers for the game or app controller.
* Methods exposed:
* Handle play button pressed
* Handle category selected
* Handle level selected
* Handle back-to-start request

## 2.1.5 Big picture of the system Architecture

A diagram of a software process

AI-generated content may be incorrect.

Figure High Level System Architecture

# 2.2 System Design

## System decomposition

* 1. **Presentation Layer**
* Components: Start Screen, Category Select, Level Select, Game View, Completion Screen
* Responsibility: Displays screens, gathers input, triggers signals for actions  
  1. **Application Controller**
* Component: SceneLoader (Main.gd)
* Responsibility: Manages screen transitions, controls scene flow  
  1. **Game Logic**
* Components: GameManager, Card.gd, ScoreManager, MatchChecker
* Responsibility: Implements game rules — flipping, matching, scoring, win detection  
  1. **Data Layer**
* Component: DataManager
* Responsibility: Loads card sets from categories.json, saves progress  
  1. **System Services (Infrastructure)**
* Components: AudioControl.gd, Godot FileAccess
* Responsibility: Plays sound effects, manages file I/O

## 2.2.2  Design Decisions

These are specific choices made to follow the principles:

1. **Use a 5-layer architecture.**

* To keep responsibilities clean and prevent direct layer skipping.

1. **Store card sets in a JSON file (categories.json).**

* Allows adding new cards without touching game logic.

1. **Use Godot signals for UI → Logic communication.**

* Keeps UI decoupled from gameplay code.

1. **Implement SceneLoader (Main.gd) as the App Controller.**

* Centralises scene management and flow control.

1. **Use a single Card.tscn for all cards.**

* Reduces duplication and simplifies asset management.

1. **Target 60 FPS and simple UI for young players.**

* Performance and accessibility are baked into every layer.

1. **Log major choices as ADRs (Architecture Decision Records).**

* Ensures we remember why we made certain architectural choices.

## 2.2.3 Design Alternatives Considered.

These are options we considered but did not choose, with reasons:

1. **Embed card data directly in GameManager.gd.**

* Rejected: Would make adding new categories require code edits and break open/closed principle.

1. **Have GameManager control scene flow directly.**

* Rejected: This would blur the separation of concerns; SceneLoader (App Controller) is a cleaner solution.

1. **Load card data at runtime on every level load.**

* Rejected: Preloading at startup reduces lag and ensures smooth gameplay for kids.

1. **Multiple Card.tscn scenes per category.**

* Rejected: Unnecessary duplication; harder to maintain.

1. **Allow UI to call Data layer directly.**

* Rejected: Breaks strict layering; would increase coupling and make future changes harder.

## 2.2.4 Documented Reuse of Components in FunFlips

1. **Card.tscn**

* We reused a single Card.tscn scene for all card instances, across all categories and levels.  
  Why:
* Avoids duplicating card logic and visuals.
* Any improvement or bug fix applies globally.
* Reduces memory usage and simplifies scene setup.

1. **AudioControl.gd**

* A single AudioControl.gd script is used as the audio manager for the entire game.  
  Why:
* Provides one place to handle all sound effects and feedback.
* Ensures consistent audio settings and logic across all scenes.
* Makes future updates to audio behavior easy.

1. **SceneLoader (Main.gd)**

* SceneLoader (part of Main.gd) is reused for managing all scene transitions.  
   Why:
* Centralizes navigation logic, keeping it separate from game rules.
* Simplifies maintaining and extending scene flow (e.g., adding new screens).

1. **categories.json**

* A single JSON file contains all card definitions for all categories and levels.  
   Why:
* Easy to add new categories or cards without changing code.
* Keeps card data clean, consistent, and DRY (Don’t Repeat Yourself).
* Allows non-programmers (designers) to extend the game easily.

Interfaces means how components, layers or systems communicate- the connection points or contracts between them.

## Activity Diagram

A diagram of a process

AI-generated content may be incorrect.

Figure Activity Diagram

**2.3 System Architecture and Design – Domain Data Model**

A diagram of a game

AI-generated content may be incorrect.

Figure Domain Data Model

# 2.3 Human Machine interface

Human-Machine Interface (HMI) — FunFlips

## 2.3.1 Requirements for the H-M Interface

* Age-appropriate design: The interface must suit children aged 4–6.
* Simplicity: Minimal steps to start or replay a game (max 3 clicks).
* Touch-friendly: Large buttons, generous touch areas.
* Clear feedback: Visual and sound cues for actions (e.g., flip, match, win).
* Accessibility: High contrast visuals, audio cues, and a mute toggle.

## 2.3.2 Design Principles and Style Guide

* KISS (Keep It Simple and Straightforward): No clutter; only essential controls visible at each step.
* Consistency: Button sizes, colors, and positions are uniform across screens.
* Large interactive elements: Designed for small hands and touch screens.
* Color & sound: Bright, cheerful colors; friendly sounds for positive feedback.
* Readability: Large, easy-to-read fonts with good contrast.

## 2.3.3 Interaction Modeling

* Start screen → Category select → Level select → Game → Completion screen — simple, linear navigation flow.
* Input method: All interactions via touch (tap, long press optional).
* Feedback loop:
  + Tap → immediate visual response (e.g., card flips)
  + Matched pair → sound + visual highlight
  + Game end → completion screen with score
* Scene transitions handled centrally by SceneLoader to keep navigation smooth and consistent.

A diagram of a diagram

AI-generated content may be incorrect.

Figure Interaction Modelling