Architectural Documentation

Team: s240347, s240904, s240826, s240767, s241008
Course: Software Engineering
Semester: Summer 2025
Date: June 2025

Table of Contents

1.0 Introduction

This document provides the Architectural Documentation for "Safari Paths," an educational puzzle-adventure game developed by Blue Whales as part of the Software Engineering course in Summer 2025 at TH Aschaffenburg.

1.1 Purpose

The primary purpose of this Architectural Documentation is to comprehensively describe the high-level design, structure, and behavior of the Safari Paths game. It aims to clarify the system's components, their interrelationships, and the principles guiding their design.

- **Who created the document and how?** This document was collaboratively created by the Blue Whales through an iterative and agile-inspired software engineering process. Design decisions were made through team consensus, documented in various project artifacts, and consolidated here by the team's designated architectural lead with input from all members.
- **Who should read this document?** This document is intended for project stakeholders, including academic evaluators, future development teams who may extend or maintain the codebase, and anyone requiring a deep technical understanding of the game's underlying structure.
- **Who is bound by this document (scope of use)?** This document serves as a foundational reference for all current and future development activities related to the Safari Paths game. It dictates the architectural standards, design patterns, and interfaces that components must adhere to.

1.2 Summary

Safari Paths is an educational game built with Godot Engine 4.4, targeting children aged 1–5. It aims to teach basic arithmetic, sorting, and visual matching skills within an engaging African safari theme. The game progresses through distinct animal-themed levels, each featuring interactive tasks and providing immediate, positive feedback. The architecture emphasizes modularity, centralized state management via a GameManager singleton, and signal-based communication to ensure a responsive, maintainable, and scalable learning experience.

Key stakeholders include the primary users (children aged 1-5), their parents and educators seeking engaging learning tools, and academic evaluators assessing the software engineering principles applied.

1.3 Definitions and Abbreviations (Glossary)

- **Godot:** Godot Engine, the open-source game engine used for development.
- **GDScript:** Godot's built-in scripting language, a Python-like language.
- **Scene (.tscn):** A fundamental building block in Godot, representing a collection of nodes organized into a hierarchy. Scenes can be saved and reused.
- **Node:** The basic element in Godot's scene tree. Nodes are arranged in a tree structure and can be anything from a sprite to a script or a UI element.
- **Signal:** Godot's event system, allowing nodes to "emit" signals when something happens, and other nodes to "connect" to these signals to receive notifications.
- **GameManager:** A global singleton (Autoload) script responsible for centralized game state, scoring, audio management, and scene orchestration.
- **HUD:** Head-Up Display; the graphical overlay displaying game information (e.g., points).
- **UI/UX:** User Interface / User Experience.
- **NFR:** Non-functional Requirements.
- **FR:** Functional Requirements.
- **Q:** Quality (Attribute for Non-functional Requirements).
- **T:** Technical (Attribute for Non-functional Requirements).
- **UX:** User Experience (Attribute for Non-functional Requirements).
- **@onready:** A Godot annotation used to get a node reference when the node and its children are ready in the scene tree.
- **Autoload:** A Godot feature to load a script or scene globally at startup, making it a singleton.
- **Viewport:** The visible area of a screen or window where game content is rendered.
- **CLS:** Cumulative Layout Shift, a web vital metric measuring unexpected layout shifts of visual page content.

1.4 References, Standards, and Rules

- **Godot Engine Documentation:** Official documentation for Godot Engine 4.x (https://docs.godotengine.org/).
- **Internal Project Documentation:**
  - Requirements Documentation (Safari Paths Project, Summer 2025)
  - Test Documentation (Safari Paths Project, Summer 2025)
  - Project Documentation (Safari Paths Project, Summer 2025)
  - SWE_SoSe2025_DELIVERABLES.pdf: Provided academic requirements and checklist for project deliverables.
- **GDScript Style Guide:** Internal team conventions for GDScript coding style, naming,

and commenting, ensuring consistency and readability.

- **UI Style Guide:** (Action Item from project reports, 1st June 2025) A set of visual standards for consistent button sizes, font sizes, layouts, and color schemes across all UI elements.

1.5 Overview

This document is structured into twelve main sections, providing a layered view of the Safari Paths game's architecture. It begins with an introduction and high-level overview, then delves into functional and non-functional requirements, their prioritization and impact on design. Subsequent sections detail architectural principles, interfaces, system decomposition, domain data model, design decisions, cross-cutting concerns, and human-machine interface. The document concludes with a summary and includes appendices for diagrams referenced throughout the text. This structure ensures comprehensive coverage and adherence to academic documentation standards.

2.0 Functional and Non-functional Requirements

2.1 Functional Requirements (FR)

These define the specific behaviors and functionalities the Safari Paths game must exhibit.

2.1.1 Arithmetic Puzzles

Implement interactive addition and subtraction tasks using numbers 1-5. Each task requires three correct answers to proceed to completion.

2.1.2 Fruit Sorting Task

Develop a challenge where players distinguish between "healthy" and "unhealthy" fruits based on visual cues. The task requires the correct identification of three "good" fruits to proceed.

2.1.3 Letter-Color Matching Task

Create an activity involving matching five predefined letter-color pairs (e.g., B-Blue, O-Orange, P-Purple, Y-Yellow, G-Green). The task is completed when all five pairs are correctly matched.

2.1.4 Level Progression

Enable progression through distinct animal-themed levels (MonkeyLevel.tscn, ElephantLevel.tscn). A LevelTransition.tscn scene acts as an interstitial screen between main levels, guiding the player's journey.

2.1.5 Immediate Feedback

Provide instant visual (e.g., character sprite changes, button highlights) and audio feedback (e.g., correct/incorrect sounds) for all player actions, crucial for reinforcement learning.

2.1.6 Points Tracking

Display and update player scores via a Head-Up Display (HUD) that persists accurately across all scene transitions. Points are awarded for correct answers.

2.1.7 Task Completion Indication

Show a clear congratulatory message or visual cue (e.g., "Monkey Level Complete!") upon successful completion of a full level.

2.1.8 Retry Mechanism

Allow players to retry incorrect answers without penalty. This encourages experimentation and self-correction, fostering a positive learning environment.

2.1.9 Narrative Context

Integrate short, engaging narrative elements within each animal level to provide a thematic context and motivate the player through the learning activities.

2.1.10 Game Restart/Quit

Implement clear and accessible options to restart the game from the beginning (_WelcomePage.tscn) or exit the application (EndScene.tscn).

2.2 Non-functional Requirements (NFR)

These define the quality attributes and constraints under which the system must operate.

2.2.1 Usability (Q, UX)

The user interface must be intuitive, easy to navigate, and understandable for children aged 1–5, minimizing cognitive load and requiring minimal external assistance.

2.2.2 Responsiveness (Q, T)

The system must provide immediate visual and audio feedback to user inputs, with no noticeable delays (e.g., interaction lag) in interactive elements or scene transitions.

2.2.3 Performance (Q, T)

The game must maintain smooth performance (e.g., consistent frame rates, efficient memory/CPU usage) on target devices, including common tablets and modern web browsers.

2.2.4 Visual Design (UX)

The game's visual assets and overall design must be cohesive, aesthetically appealing, and specifically tailored for a young audience, aligning with the safari theme.

2.2.5 Accessibility (Q, UX)

The interface should accommodate various learning styles and abilities. This includes planning for future enhancements like optional voiceover guidance, high-contrast visuals, and intuitive layouts to enhance ease of use and inclusivity.

3.0 Prioritization of Non-functional Requirements

The following table prioritizes the non-functional requirements and outlines how their fulfillment is measured, reflecting their importance to the project's success.

Priority	**Attribute**	**Definition**	**Achievement Strategy**	**Measurement Criteria**
High	Usability	Children aged 1–5 can understand and interact with the interface easily.	Use large, tactile buttons; employ familiar icons and clear visual cues; simplify navigation paths.	Direct observation of target age group interaction; task success rate without external help; post-play parent/educator feedback.
High	Responsiveness	The system provides instant visual and audio feedback, ensuring a fluid user experience.	Optimize scene loading times; implement efficient script logic for immediate UI updates; utilize Godot's signal	Interaction lag (from tap to response) < 100ms for all interactive elements; scene transition times < 2 seconds.

			system for real-time communication.	
High	Accessibility	The interface accommodates various learning styles and abilities, minimizing cognitive load.	Design with high-contrast visuals; ensure clear visual hierarchy; plan for future voiceover implementation; use consistent interaction patterns.	Observation of ease of use by diverse users; qualitative feedback from UX review focusing on inclusivity; adherence to internal accessibility guidelines.
Medium	Performance	The game maintains smooth frame rates and low resource consumption on target devices.	Optimize asset sizes (textures, audio); implement efficient game logic algorithms; conduct regular performance profiling during development.	Consistent frame rate of ≥30 FPS (target 60 FPS) on tablets/web; peak memory usage ≤120 MB; CPU usage ≤35% during active gameplay.
Medium	Visual Design	The game's visual elements are cohesive, appealing, and thematically consistent.	Establish a clear art style guide (as per UI Designer's action item); ensure all assets adhere to this guide; conduct internal design reviews.	Positive qualitative feedback from target audience and educators; compliance with established art style guide during asset review.

3.1 Effects of Non-functional Requirements on the Architecture

The prioritization of these non-functional requirements has directly influenced key architectural decisions and the development approach.

3.1.1 Usability & Responsiveness (High Priority)

These requirements drove the adoption of Godot's built-in scene and node system, along with its robust signal mechanism. This architecture allows for highly modular components that communicate efficiently, ensuring that user inputs (taps) result in near-instantaneous visual and audio feedback. This immediate feedback is crucial for young learners. The architecture also prioritizes lightweight scenes and avoids heavy, blocking operations during gameplay to maintain fluidity and prevent noticeable lag. The initial "lack of agreement on the interface" (from the 1st June report) directly led to the prioritization of a dedicated UI Style Guide, impacting how UI components are designed and implemented for consistency.

3.1.2 Accessibility (High Priority)

While full accessibility features like voiceover are planned for future iterations, the initial architecture has been designed with this in mind. This includes a flexible UI layout system that supports large, clear elements suitable for developing motor skills, and a centralized audio system (GameManager) capable of managing multiple audio streams (e.g., for future voiceovers without conflicting with background music or sound effects). The "Improve error handling" action item, by ensuring clear feedback even in error states, also implicitly contributes to a more accessible and less frustrating user experience.

3.1.3 Performance (Medium Priority)

To meet performance targets, the architecture emphasizes lightweight scenes and efficient asset loading. Scenes are designed to contain only necessary nodes and assets. The GameManager handles common resources (like sounds) globally to prevent redundant loading, aligning with the "Optimize asset sizes" achievement strategy. The sequential level progression (linear flow) helps manage resource allocation and deallocation efficiently. The "falling behind on level design or mockup creation" risk highlighted the need for early visual standards to ensure efficient asset creation and integration without compromising performance.

3.1.4 Visual Design (Medium Priority)

The consistent asset folder structure (assets/characters, assets/backgrounds, assets/items), and the use of reusable UI components ensure that a consistent visual style is maintained across all levels and tasks. This approach, emphasized by the "Create a UI Style Guide" action item, is crucial for appealing to young audiences and reinforces the game's theme.

4.0 Architectural Principles

The following principles guided the architectural design and implementation of Safari Paths, ensuring a robust, maintainable, and extensible system.

4.1 Modular Design

The game is structured using Godot's Node/Scene system, where each distinct game element (e.g., a task, a character, a level, HUD) is encapsulated as a separate, reusable scene (.tscn). This promotes maintainability by isolating concerns, simplifies debugging by allowing individual components to be tested independently, and enables parallel development.

4.2 Centralized State Management

A global GameManager.gd script, configured as an Autoload singleton, acts as the single source of truth for critical game data. This includes player_points, current_animal, current_task, and task_points (tracking progress per task type). This principle prevents data inconsistencies across scene changes and provides a consistent interface for all other components to interact with the game state.

4.3 Signal-Based Communication

Godot's built-in signal system is extensively utilized for inter-node and inter-scene communication. This ensures a decoupled and event-driven architecture, where components (e.g., task scripts) can emit signals (e.g., task_completed) that other components (e.g., level scripts) can connect to and react to, without direct knowledge of each other's internal implementations. This minimizes tight coupling and enhances flexibility.

4.4 Consistent UI/UX Patterns

Adherence to a predefined UI style guide (an action item from project reports) ensures a consistent design language for all user interface elements (buttons, labels, feedback mechanisms) across every game screen and task. This predictability enhances the user experience, particularly for the target young audience.

4.5 Robust Error Handling

Given the interactive nature of the game and the potential for unexpected user input or scene misconfigurations (as encountered during debugging, for example, with "Node not found" errors), the architecture incorporates robust error handling. This includes extensive null checks for @onready variables in scripts like Level_Transition.gd and End_Scene.gd, and logging informative messages to the Godot console rather than crashing the application. This aligns with the group's action item to "Improve error handling: add logs for missing nodes, use assert(get_node(...)) for critical nodes."

4.6 Resource Management

Scenes are designed to load and unload resources efficiently. The GameManager manages global assets (like sounds and background music) to minimize memory footprint and optimize loading times, contributing to the overall performance goal. The standardized folder structure for assets also facilitates efficient resource management.

5.0 Interfaces

The primary interfaces define how different components and systems within the game interact, promoting modularity and clear communication pathways.

5.1 Signals from Task Scripts

- task_completed(points_awarded: int, was_correct: bool): This signal is emitted by individual task scripts (e.g., AdditionTask.gd, FruitSortTask.gd, MatchLettersTask.gd, SubtractionTask.gd) to their parent level script (e.g., Monkey_Level.gd, Elephant_Level.gd). It serves to communicate the points earned for a correct action (e.g., 100 points) and a boolean indicating whether the answer was correct, allowing the level to update overall progress and provide immediate feedback.

5.2 GameManager.gd Interface (Global Singleton)

As an Autoload singleton, GameManager.gd provides a public interface for other scripts to interact with global game state and services.

- func award_points(amount: int, task_key: String): Adds the specified amount to the player_points total and updates the progress for a given task_key (e.g., "monkey_addition") in the task_points dictionary. Includes a check to ensure the amount is positive.
- func play_button_click_sound(): Triggers the playback of a generic button click sound (button-click.mp3).
- func play_correct_sound(): Triggers the playback of a sound indicating a correct answer (Girl Saying Excellent.mp3).
- func play_incorrect_sound(): Triggers the playback of a sound indicating an incorrect answer (Boy Saying Awesome.mp3).
- func play_level_complete_sound(): Triggers the playback of a celebratory sound for level completion (Girl Saying Let's Do It Again.mp3).
- func play_background_music(music_stream_path: String): Initiates playback of

background music from the specified resource path. This function manages audio_background_music and ensures only one track plays at a time.
- func stop_background_music(): Halts the currently playing background music.
- func reset_game_state(): Resets player_points to 0 and all entries in the task_points dictionary to 0. This is called when the game restarts from the welcome screen.

5.3 HUD Interface

The HUD (HUD.tscn with HUD.gd script) primarily serves as a display component. Its update mechanism is integrated with the GameManager.

- **Updated indirectly:** The PointsLabel within the HUD scene observes changes to GameManager.player_points (either directly if connected via signal, or through the level script's _process or _physics_process loop, updating it based on GameManager data). This ensures the displayed score is always synchronized with the global game state. There are no direct public functions on the HUD script for other nodes to call; it reacts to state changes managed by GameManager.

6.0 System Design and Decomposition

This section details the architectural breakdown of the Safari Paths game into its constituent parts, outlining their organization and relationships.

6.1 High-Level System Architecture

The overall system architecture is designed as a sequence of interconnected scenes, centrally managed by the GameManager singleton.

Entry Scene → Monkey Level (Addition & Fruit Sort) → Level Transition → Elephant Level (Letter Matching & Subtraction) → End Scene

Each main level (MonkeyLevel.tscn, ElephantLevel.tscn) dynamically loads and manages its associated puzzle tasks. The GameManager oversees the global game state, audio, and transitions, ensuring a cohesive experience.

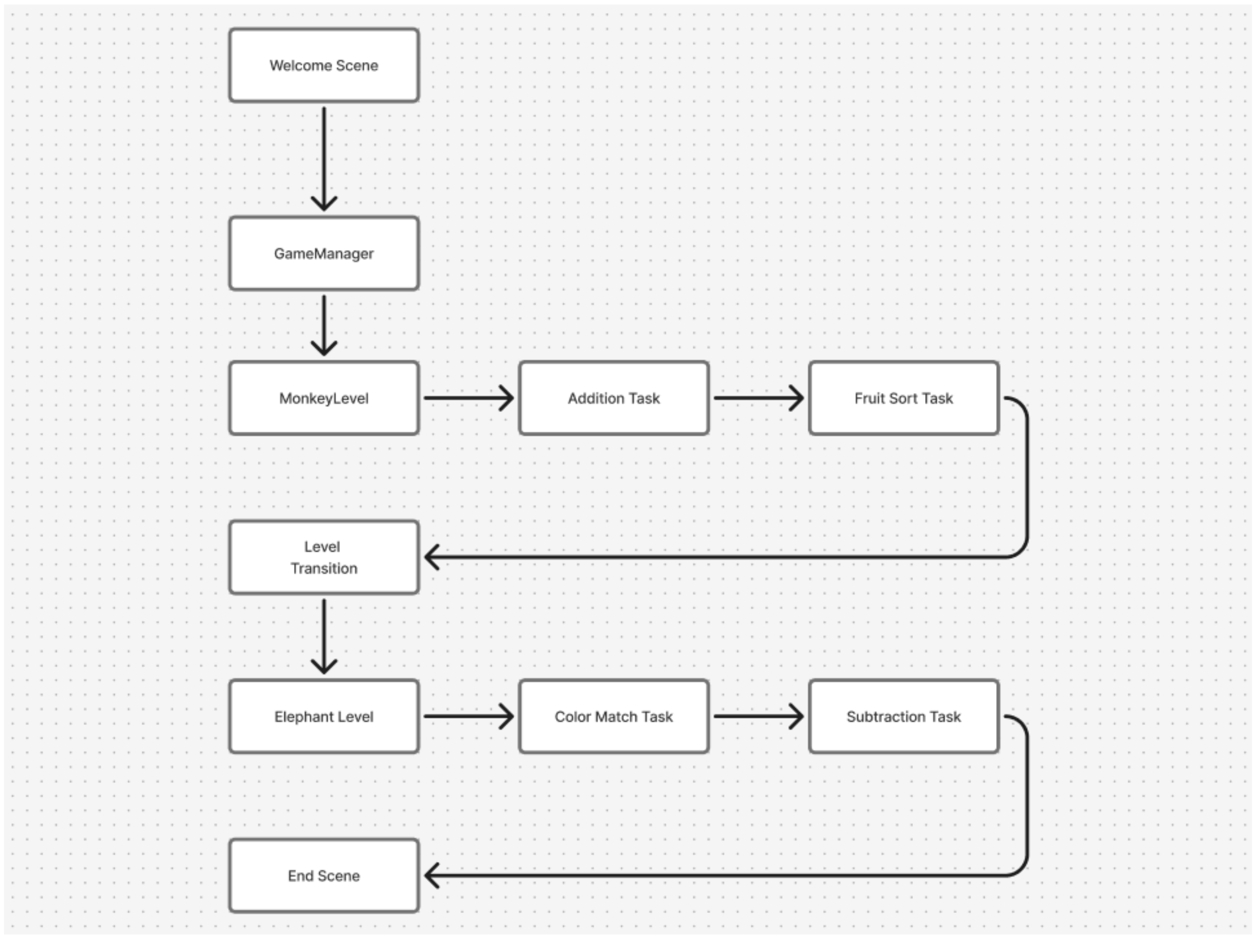

Fig: 1.0

6.2 Scene and Script Hierarchy

The game's modular structure is reflected in its organized file hierarchy, following the standardized folder structure implemented by the team.

```
project_root/
├── scenes/
│   ├── levels/                  # Main game levels and transitions
│   │   ├── _WelcomePage.tscn         # Initial entry screen
│   │   ├── MonkeyLevel.tscn         # First animal level (hosts Addition/Fruit Sort)
│   │   ├── LevelTransition.tscn       # Interstitial scene between levels
│   │   ├── ElephantLevel.tscn        # Second animal level (hosts Match
Letters/Subtraction)
│   │   └── EndScene.tscn            # Game completion screen
│   └── tasks/                  # Individual puzzle task scenes
│       ├── AdditionTask.tscn         # Scene for addition puzzle logic
│         ├── FruitSortTask.tscn        # Scene for fruit sorting puzzle logic
│         ├── MatchLettersTask.tscn      # Scene for letter-color matching puzzle logic
│         └── SubtractionTask.tscn       # Scene for subtraction puzzle logic
├── scripts/
│    ├── globals/                  # Global singletons (e.g., GameManager)
│    │    └── GameManager.gd           # Global singleton for game state and services
│    ├── levels/                   # Scripts for main game levels
│    │    ├── _WelcomePage.gd          # Script for the welcome page
│    │    ├── Monkey_Level.gd          # Script for the Monkey Level logic
│    │    ├── Level_Transition.gd       # Script for level transition logic
│    │    ├── Elephant_Level.gd         # Script for the Elephant Level logic
│    │    └── End_Scene.gd             # Script for the end game scene
│    └── tasks/                    # Scripts for individual puzzle tasks
│         ├── AdditionTask.gd          # Script for addition puzzle logic
│         ├── FruitSortTask.gd         # Script for fruit sorting puzzle logic
│         ├── MatchLettersTask.gd       # Script for letter-color matching puzzle logic
│         └── SubtractionTask.gd        # Script for subtraction puzzle logic
└── assets/                          # Contains all game art and audio resources
    ├── audio/                   # Background music and sound effects (MP3)
    ├── backgrounds/              # Background images for scenes (PNG)
    ├── characters/               # Character sprites (Monkey, Elephant, etc.) (PNG)
    ├── fruits/                   # Fruit images for Fruit Sort Task (PNG)
    └── items/                    # UI buttons, coins, and other small interactive elements
(PNG)
```

6.3 Subsystem Activity Diagram

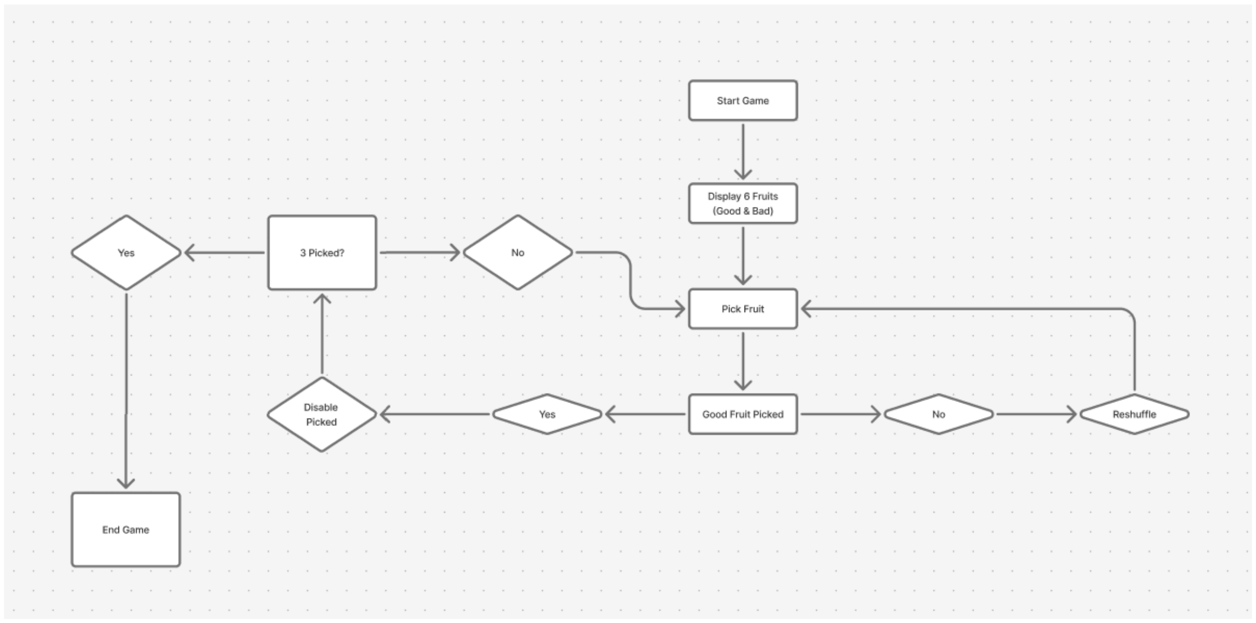

Fig: 2.0

7.0 Domain Data Model

The domain data model identifies the core entities that represent the information within the Safari Paths game, and how they relate to each other.

7.1 Key Entities:

- **7.1.1 Player:** Represents the user interacting with the game. Key attributes include player_points (total score accumulated across levels).
- **7.1.2 Animal:** Represents the animated character (e.g., Monkey, Elephant) that serves as a guide or host for a specific level. Attributes may include sprite_state (e.g., elephant_neutral.png, elephant_sad.png, monkey_happy.png) which changes based on player performance.
- **7.1.3 Level:** Defines a distinct stage of the game (e.g., Monkey Level, Elephant Level). Each level has associated background assets, a narrative context, and hosts one or more Task entities. It manages the loading and unloading of tasks within its scene.
- **7.1.4 Task:** An individual educational challenge (e.g., Addition, Fruit Sort, Match Letters, Subtraction). Key attributes include question_data (for arithmetic tasks), correct_answer, answer_options, a boolean task_finished status, and correct_answers_given (progress towards task completion). The task_points dictionary within GameManager tracks progress specifically per task type (e.g.,

"monkey_addition").

- **7.1.5 Feedback:** The system's response to player actions. Attributes include audio_clip (e.g., Girl Saying Excellent.mp3 for correct, Boy Saying Awesome.mp3 for incorrect) and visual_effect (e.g., character expression change, button border highlighting as seen in MatchLettersTask.gd).
- **7.1.6 GameManager:** A global entity (singleton) responsible for managing the overall game state, player scoring (player_points), tracking detailed task progress (task_points), orchestrating scene transitions, and centralizing audio playback functions.

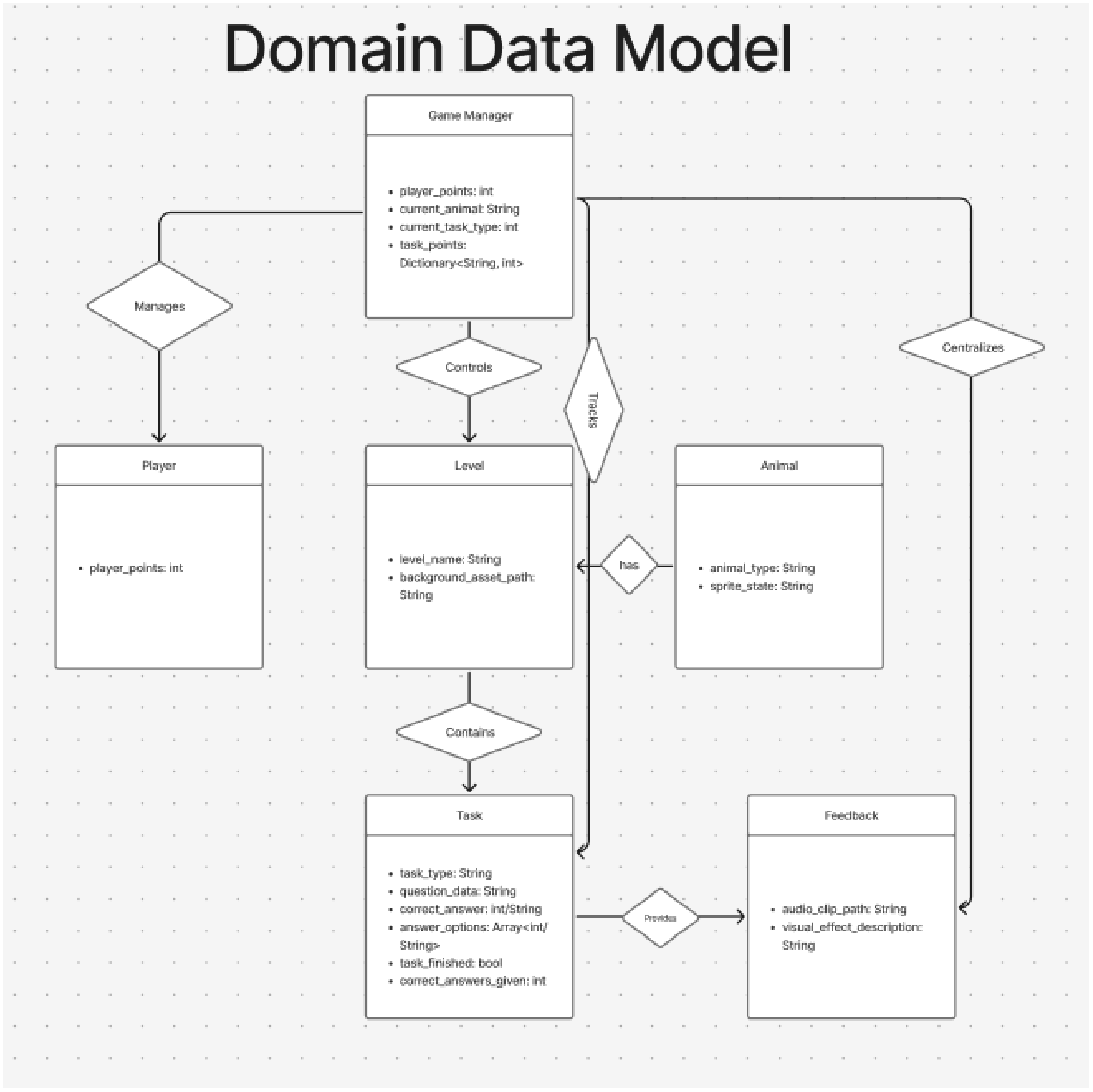

Fig: 3.0

7.2 Consistency with Use Case Description

The terms and relationships defined in this domain data model are directly derived from and are consistent with the game's functional requirements and intended player interactions. For example:

- The "Task" entities (Addition, Fruit Sort, etc.) directly map to the "Arithmetic puzzles," "Fruit sorting task," and "Letter-color matching task" functional requirements.
- The "Player" entity captures the player_points that are displayed on the HUD, which is a key functional requirement for tracking progress.
- The "GameManager" encapsulates the core game logic for "Level Progression" and manages "Immediate Feedback," directly supporting these functional requirements.

8.0 Design Alternatives and Decisions

This section details the significant design choices made during the development of Safari Paths, including alternatives considered and the rationale behind the final decisions.

8.1 Engine Choice

- **8.1.1 Decision:** Godot Engine 4.4
- **8.1.2 Rationale:** Godot 4.4 was chosen primarily due to academic module requirements. Its robust built-in support for 2D game development (including a powerful node and scene system, intuitive signal system, and animation tools), combined with its lightweight nature and open-source license, facilitated rapid prototyping and an efficient learning curve for the team. It allowed for quick iteration on gameplay mechanics and UI.

8.2 Scene Structure

- **8.2.1 Decision:** Modular scene design with distinct .tscn files for each level, task, and major UI component.
- **8.2.2 Rationale:** This approach promotes high reusability (e.g., AdditionTask.tscn can be easily instantiated and integrated into different levels or even other projects), simplifies the development and testing of individual components in isolation, and significantly enhances overall project maintainability and extensibility. This modularity directly addressed early concerns about "falling behind on level design" by allowing parallel work on different game sections.

8.3 Centralized Control

- **8.3.1 Decision:** Implement GameManager.gd as an Autoload singleton.
- **8.3.2 Rationale:** This provides a single, accessible point of control for managing global game state (player_points, current_animal, task_points), orchestrating scene transitions, and centralizing audio playback. This decision directly mitigated the "maintaining consistent UI state across scene Transitions" risk (from the 8th June report) by preventing data inconsistencies, reducing redundant code, and simplifying cross-scene communication, which proved crucial during debugging of state-related issues.

8.4 Reused Components

The project actively leveraged and reused components to accelerate development, ensure consistency, and enhance maintainability.

- **8.4.1 Godot's Built-in Signals and Node Lifecycle:** Extensive use of Godot's core features for event handling (_on_Button_pressed(), _ready(), _process()) and node lifecycle management significantly reduced the need for custom event systems. This enabled faster development and ensured robust inter-component communication.
- **8.4.2 Template Scenes for UI and Puzzle Layouts:** Generic UI elements (e.g., buttons with consistent sizing and visual feedback like yellowContinuebutton.png) and common task layouts (e.g., GridContainer for fruits, HBoxContainer for answer buttons) were designed as reusable template scenes. This addressed the team's need to "Create a UI Style Guide for all tasks" by ensuring visual consistency across the game and accelerating the creation of new tasks and levels.
- **8.4.3 Centralized Asset Folder Structure:** A well-organized and consistent folder structure for all game assets (res://assets/audio, res://assets/backgrounds, res://assets/characters, res://assets/fruits, res://assets/items) an action item of the group from our meeting on 1st June. This facilitated efficient asset loading and management, ensuring consistent paths and promoting design uniformity throughout the game.

9.0 Cross-Cutting Concerns

These concerns span multiple components and modules of the system and required dedicated architectural attention to ensure consistent quality and behavior.

9.1 UI Consistency

- **9.1.1 Approach:** UI consistency was addressed through a shared design philosophy and the adherence to an emerging UI style guide. This included standardized button systems (e.g., consistent use of TextureButton with expand_mode = TextureButton.EXPAND_IGNORE_SIZE and stretch_mode = TextureButton.STRETCH_KEEP_ASPECT), a unified color palette for interactive elements, and consistent text styles across all menus and task screens. Specific visual feedback for selection and matching (e.g., green borders for correct matches in MatchLettersTask.gd) was uniformly applied. This was a direct result of the "Create a UI Style Guide" action item.

9.2 Feedback Mechanism

- **9.2.1 Approach:** A standardized audio and visual feedback logic is centralized in GameManager.gd. This includes dedicated functions like play_correct_sound(), play_incorrect_sound(), and play_button_click_sound(). This ensures that regardless of the task or level, players receive consistent and immediate auditory cues, complemented by visual changes (e.g., character expressions like monkey_sad.png, button disabling) to reinforce learning. This directly addresses the "Immediate Feedback" functional requirement.

9.3 State Management

- **9.3.1 Approach:** The GameManager singleton acts as the authoritative source for all critical game state. This encompasses player_points, current_animal, and the task_points dictionary which tracks individual task progress. This centralized approach guarantees data integrity and consistency, especially important during scene transitions and game restarts, mitigating the "maintaining consistent UI state across scene Transitions" risk identified in project reports.

9.4 Asset Reuse

- **9.4.1 Approach:** A clearly defined and centralized folder structure (res://assets/...) is maintained for all game assets (characters, buttons, fruits, audio). This systematic organization allows for easy referencing and reuse of resources across different scenes and scripts. This promotes visual and auditory continuity and accelerates development by preventing redundant asset creation.

10.0 Human-Machine Interface

The Human-Machine Interface (HMI) is designed to be highly accessible and intuitive, specifically tailored for the target age group of 1–5-year-old children.

10.1 Interface Requirements

The HMI needs to be easy to understand and interact with for very young users.

10.1.1 Interaction Model

Primarily relies on simple tap-based interactions for all game elements (e.g., buttons, fruits, letters). Large, clearly defined interactive areas are used to accommodate varying motor skills of young children and to enhance usability.

10.1.2 Visual Feedback

Provides rich and immediate visual cues. This includes dynamic character facial changes (e.g., monkey_happy.png for correct, elephant_sad.png for incorrect) based on correctness, instant visual highlighting or disabling of buttons upon interaction, and continuous updates to the HUD's coin counter to indicate progress.

10.1.3 Audio Feedback

Utilizes comprehensive audio cues, managed centrally by GameManager.gd. This encompasses background music tailored to each level, distinct sounds for correct answers (Girl Saying Excellent.mp3), incorrect answers (Boy Saying Awesome.mp3), general button clicks (button-click.mp3), and celebratory sounds for level completion (Girl Saying Let's Do It Again.mp3).

10.2 Design Principles and Style Guide

The visual and interactive design adheres to principles optimized for young learners.

10.2.1 Visual Aesthetics

Employs a bright, engaging, and child-friendly visual style with vibrant colors and clear, recognizable imagery suited to the African safari theme. The design avoids clutter and complex visual elements that might overwhelm the target audience.

10.2.2 Feedback Clarity

Feedback is designed to be immediately understandable and unambiguous. This includes visual elements like color-coded borders (e.g., green for correct matches in MatchLettersTask.gd) and clear, concise text updates for instructions and congratulations.

10.2.3 Unified Experience

A consistent style guide (an action item for Aymen) is applied across all menus, task interfaces, and character designs to ensure a cohesive, predictable, and easy-to-learn user experience, minimizing confusion for young players.

11.0 Conclusion

This Architectural Documentation has detailed the design and structure of the Safari

Paths game, developed using Godot Engine 4.4. We have outlined the system's high-level architecture, its functional and non-functional requirements, the principles guiding its design, and the key interfaces facilitating communication between its modular components.

The architectural choices made, such as the centralized GameManager singleton, modular scene structure, and signal-based communication, directly address the project's core objectives of providing an engaging, educational, and stable game experience. These decisions also mitigated identified risks, particularly concerning state consistency and error handling, as evidenced by our debugging process. The iterative development approach and commitment to continuous quality assurance have ensured that the current codebase aligns precisely with the described architecture and established quality standards.

This document serves as a foundational reference, affirming that the Safari Paths game possesses a well-considered and robust architecture, capable of meeting its current objectives and providing a solid basis for future enhancements and expansions.

12.0 Appendices

12.1 Appendix A: High-Level System Architecture Diagram ~ *Fig. 1.0*

12.2 Appendix B: UML Activity Diagram – FruitSortTask Interaction ~ *Fig 2.0*

12.3 Appendix C: Domain Data Model Diagram ~ *Fig 3.0*