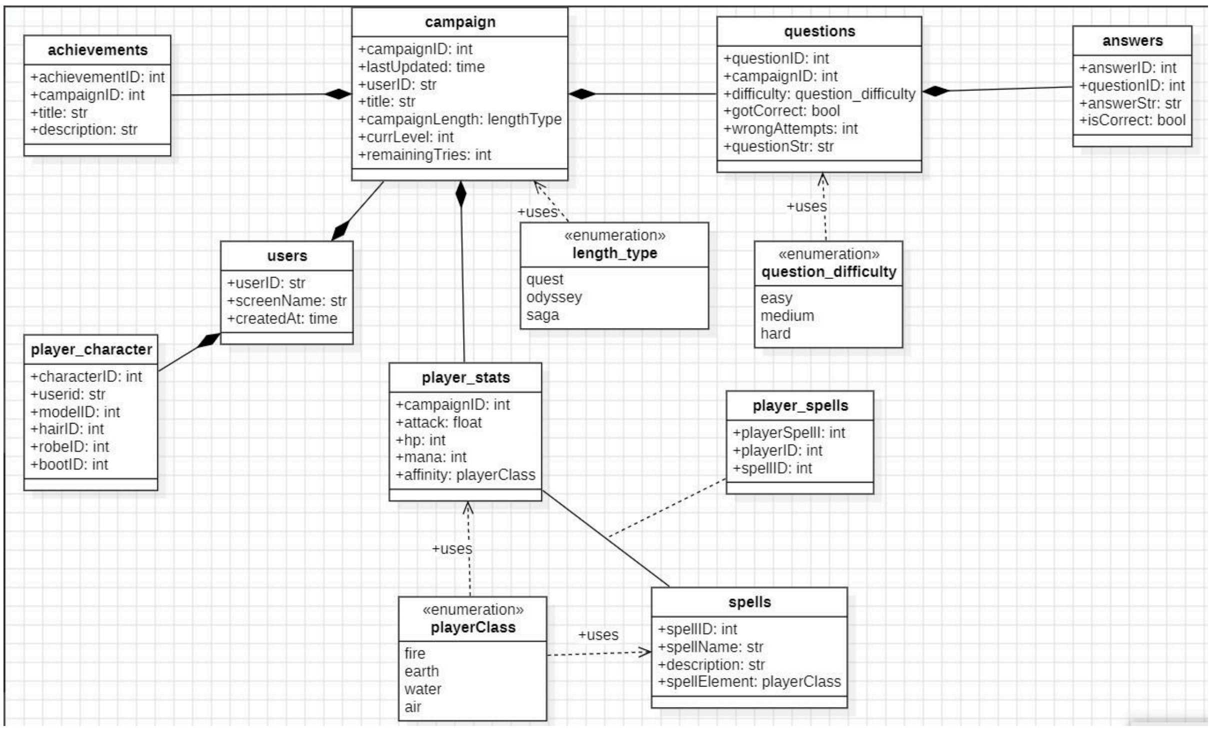
WizdomRun Design Document

System Architecture

The game-based learning application uses client-server interactions to allow the front end to make HTTP requests to the backend. Firebase manages authentication, while the database is independent and managed by a Flask backend. The architecture is similar to a monolithic architecture, as the database takes requests from a single application block where components are tightly coupled. Along with monolithic the system can be modelled by layered architecture, as the Unity in frontend presents the game, which requests logic from the C# frontend, which makes API calls to the data layer (PostgreSQL Database).

- Frontend (Unity in C#): Handles UI, graphics, and game logic
- Backend (Flask on Render): Manages API Requests, handles authentication through Firebase, leverages the LLM service for question generation
- LLM Service (Python): Interfaces with GPT-4o mini to turn notes in PDF format into question-and-answer sets
- Database (PostgreSQL on Neon): Stores users, campaigns, questions, answers, and other game-related data that can be accessed and modified by the backend

Backend UML Class Diagram



The above class diagram shows key entities such as users, campaigns, questions, answers, and game mechanics (e.g., player_stats, spells). Foreign key constraints form relationships between classes and demonstrate composition throughout the database. Enumerations ensure data integrity by creating types for use within the classes.

Design Patterns

Several design patterns are utilized within the application to enhance maintainability, modularity, and scalability:

Backend

- MVC Pattern:
  - Models:
    - Represented by the entities in the database (Users, Campaigns etc.)
  - Views:
    - API endpoints that deliver information (to the frontend in this case)
  - Controllers:
    - Logic is handled by the Flask app itself, controlling and processing requests and responses
- Observer Pattern:
  - Triggers such as check_answer_count and check_correct_answer observe the database until a given event and then call a function to execute specific requirements within the database

Frontend

- Singleton Pattern:
  - Manager Classes:
    - Used to ensure global access to key game components like CampaignManager, which mediates between functions and CampaignService. AuthManager follows the same pattern
- Adapter Pattern:
  - Used to convert objects retrieved from the database through the API endpoints into local variables in Unity
- State Pattern:
  - Used for managing Player and Enemy states (Idle, Attacking, Moving, Dead) in the game mechanics

SOLID Principles

- Single Responsibility Principle:

    - Each class/element has a unique and well-defined responsibility, which performs a singular and specific job. For example:
        - AuthManager/CampaignManager manage only authentication and campaign interactions respectively
        - The user model defines only user traits, with the user's stats, questions, characters etc. all stored in their models
- Open/Closed Principle:
    - API endpoints in the backend are designed in such a way that existing code need not be modified to extend to new functionality
- Liskov Substitution Principle:
    - Unity interfaces allow different states for Player and Enemy to be used interchangeably to control their on-screen actions
- Interface Segregation Principle:
    - Managers in unity interact with service classes to handle a single responsibility:
        - AuthService only handles Firebase authentication, so classes that deal with campaigns don’t have to depend on authentication logic
        - CampaignService manages campaign-related operations, so authentication-related classes don’t depend on campaign logic
        - If another service is introduced (perhaps QuestionService for handling questions), it remains separate and does not introduce unnecessary dependencies into AuthService or CampaignService
- Dependency Inversion Principle:
    - AuthService is a low level class that directly interacts with Firebase which is an external dependency. AuthManager is a high level class that provides a stable interface by abstracting authentication logic for Unity. Dependencies:
        - Unity depends on AuthManager, not Firebase directly
        - AuthService is depended on by Firebase
        - If a new auth provider is used instead of the whole application being dependent on firebase, AuthService needs to be modified and AuthManager will still manage authentication using AuthService

Conclusion

Scalability and maintainability are key areas for the design process for this system. The backend follows REST API principles, while the frontend uses design patterns to enhance the architecture. SOLID principles are applied to ensure that future development can add to existing code instead of replacing it.