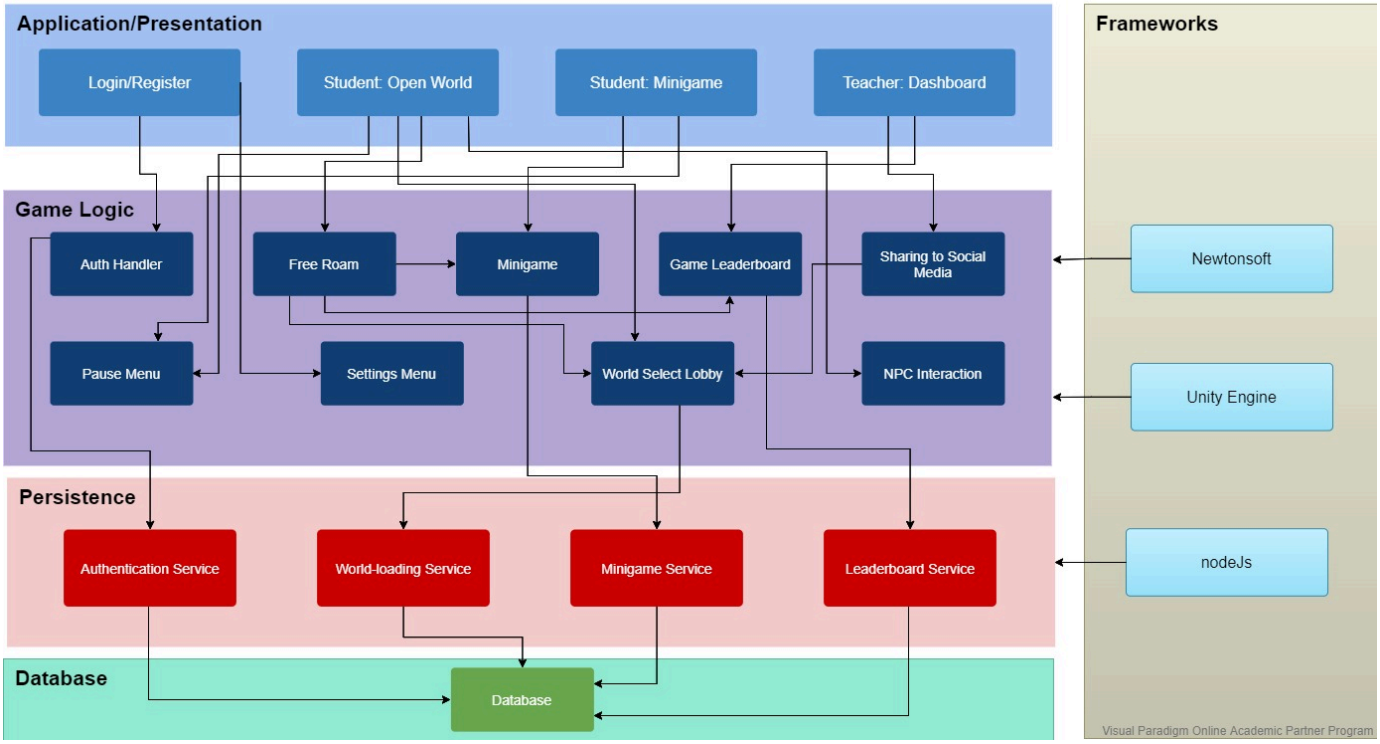
Design Document

Table of Contents

1. Candidate Architecture

Call & Return Architectural Style

Call & Return Architecture
Layered Subtype



The layered subtype of the Call & Return Architectural style enables the application to be segmented into various levels.

We chose this architectural style for a variety of reasons. Firstly, this allows for a ‘separation of concerns’ to be achieved, where each layer only needs to concern itself with the data and controls under its scope. This highlights the testability of adopting this architecture, as the resulting distinct components will be much easier to test.

We will also make use of the concept of ‘layers of isolation’, where changes in one layer will not directly impact other layers. Requests must pass through all the layers sequentially, meaning that the top-most application layer would not have direct access to the database layer, and vice versa. This allows for much looser coupling, allowing for increased flexibility.

Our main logic-handling layers are the topmost application/presentation layer and the game logic layer. The persistence layer is used to abstract the classes required to communicate with the database, allowing for easier flexibility. Furthermore, abstracting these classes allows for improved testability, as we are able to verify the correctness of database calls easily.

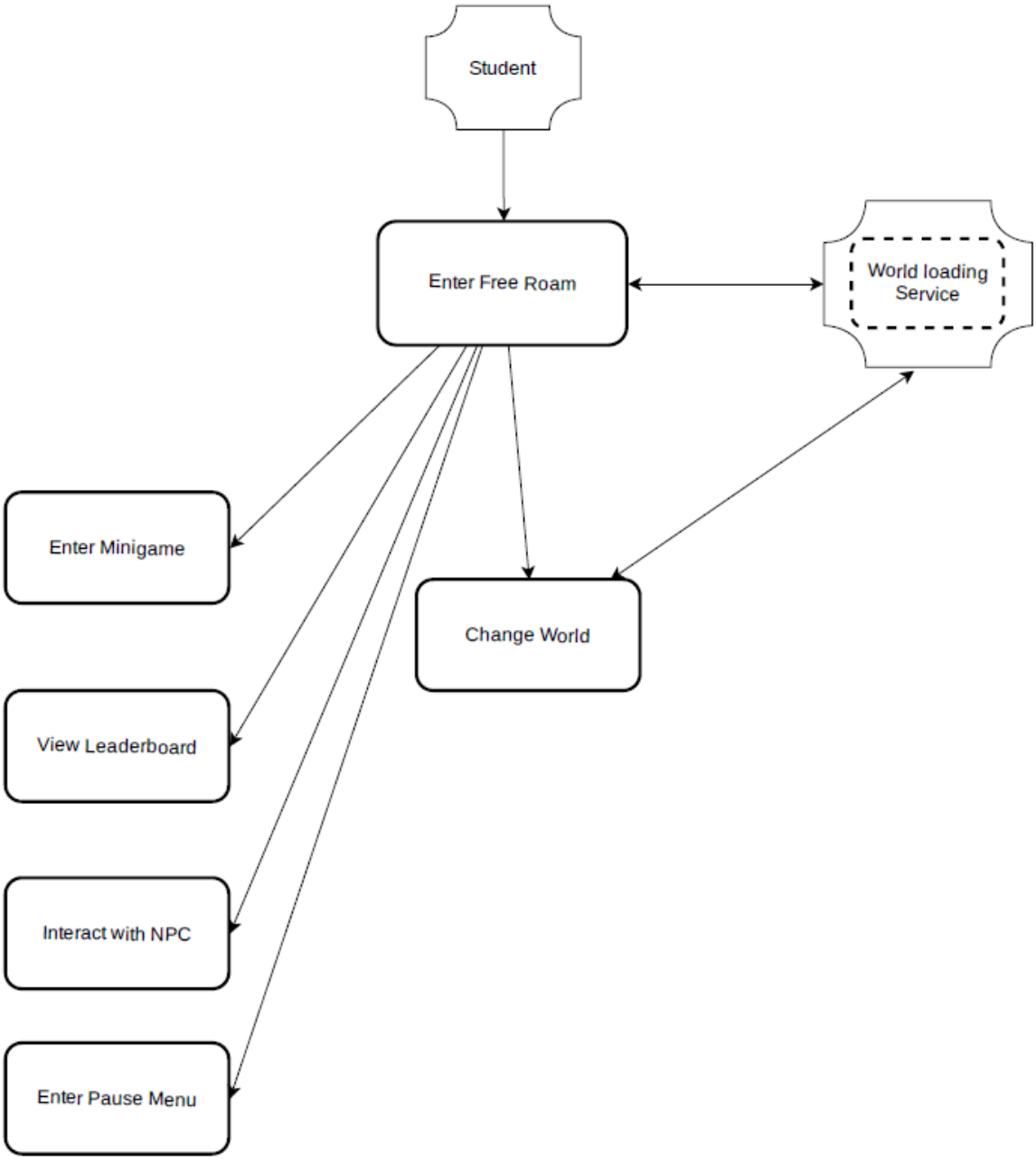
The final database layer represents the databases containing the data. This distinction helps prevent tight coupling, such as if the database was able to communicate with all other layers simultaneously; which would be much more difficult to test and expensive to maintain.

Diagrams below illustrate the data and control flow between the various component within each layer. Only significant components (i.e. components in the Game Logic section of the architectural diagram above) are included in this analysis.

For the purpose of brevity and conciseness, only the directly related components are illustrated in each diagram. The purpose of this would be to draw focus to the immediate components and control/data flow associated with any given component. Associated components are further elaborated on in subsequent sections.

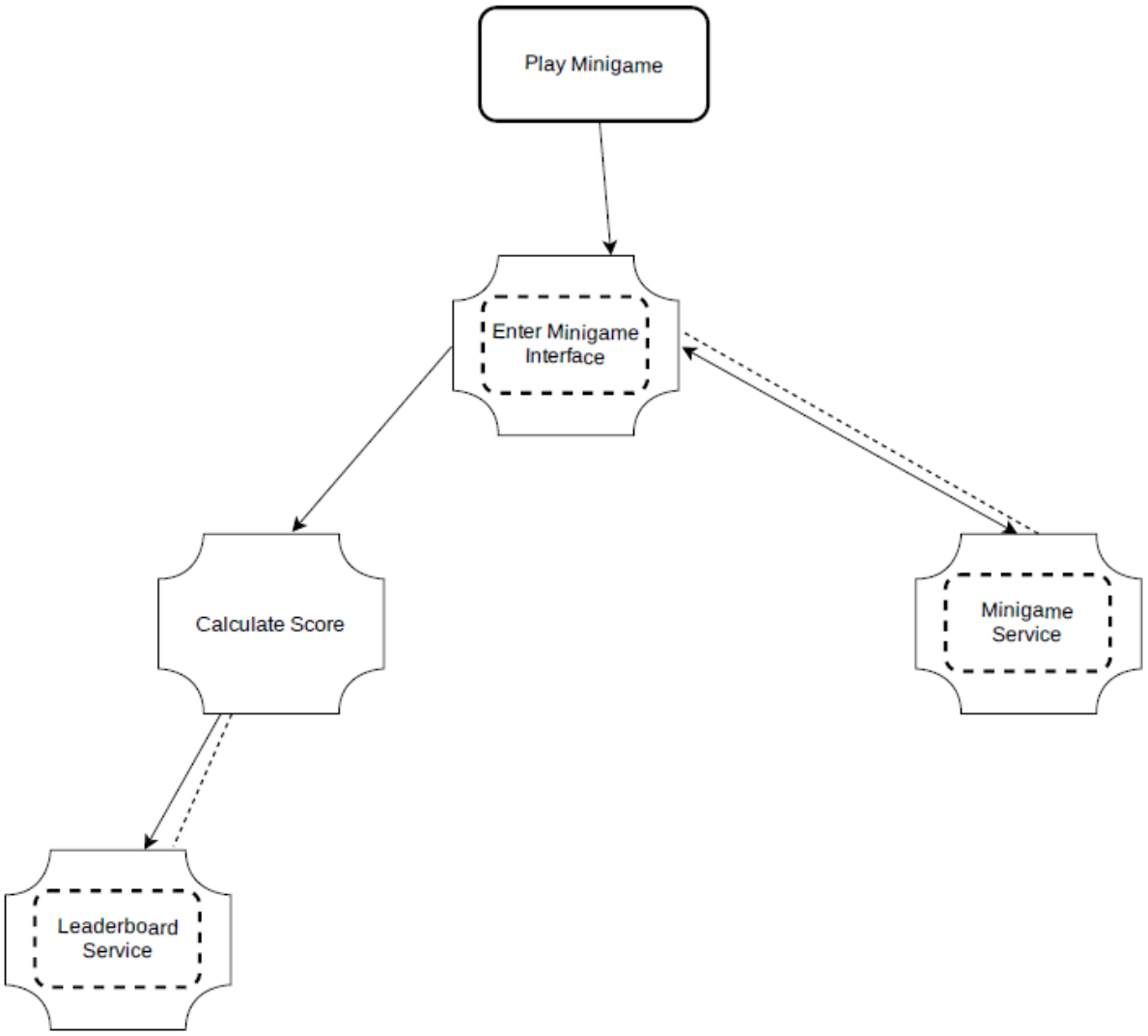
Game Logic Layer:

Free Roam: Architecture Visual Notation Diagram

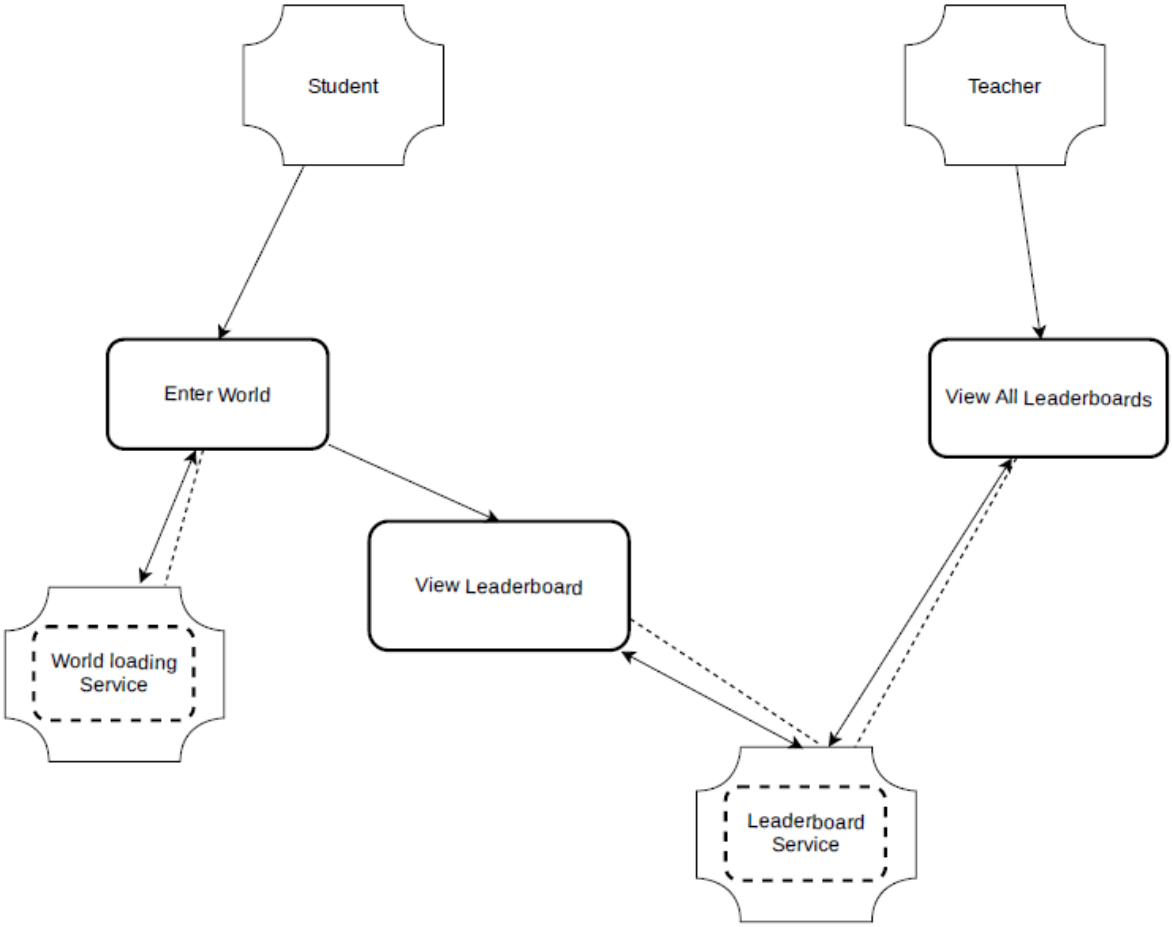


The components shown here are further analyzed and elaborated on in future sections.

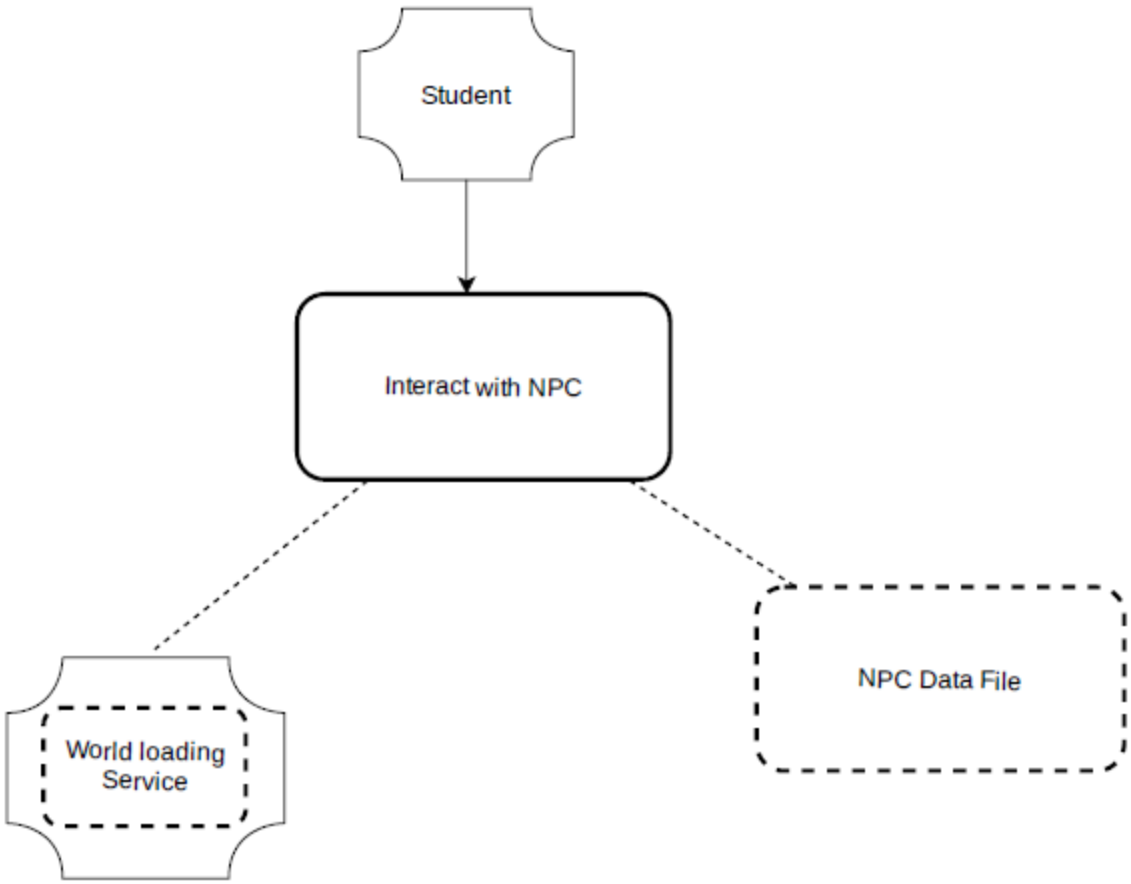
Minigame Component: Architecture Visual Notation Diagram



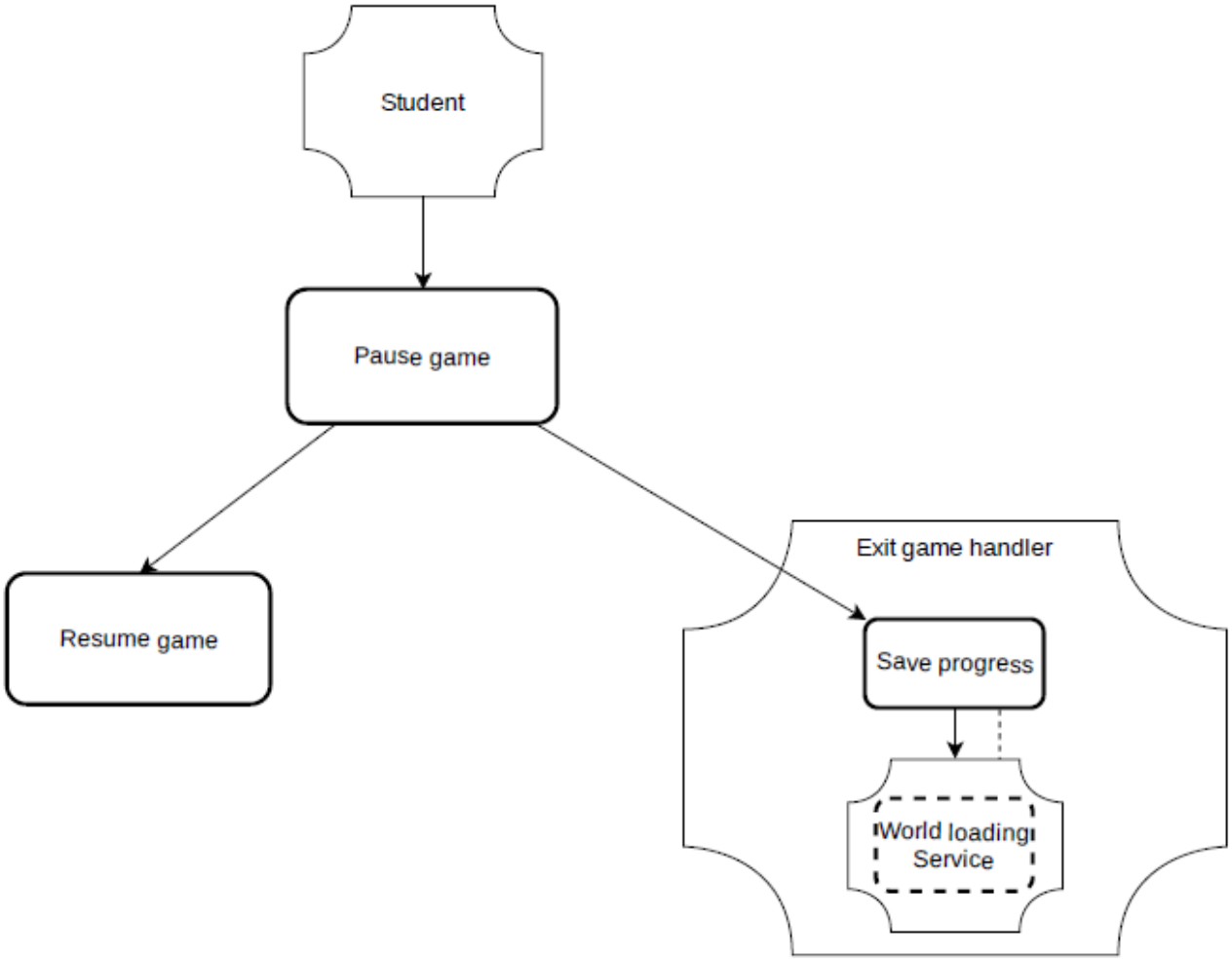
Leaderboard Component: Architecture Visual Notation Diagram



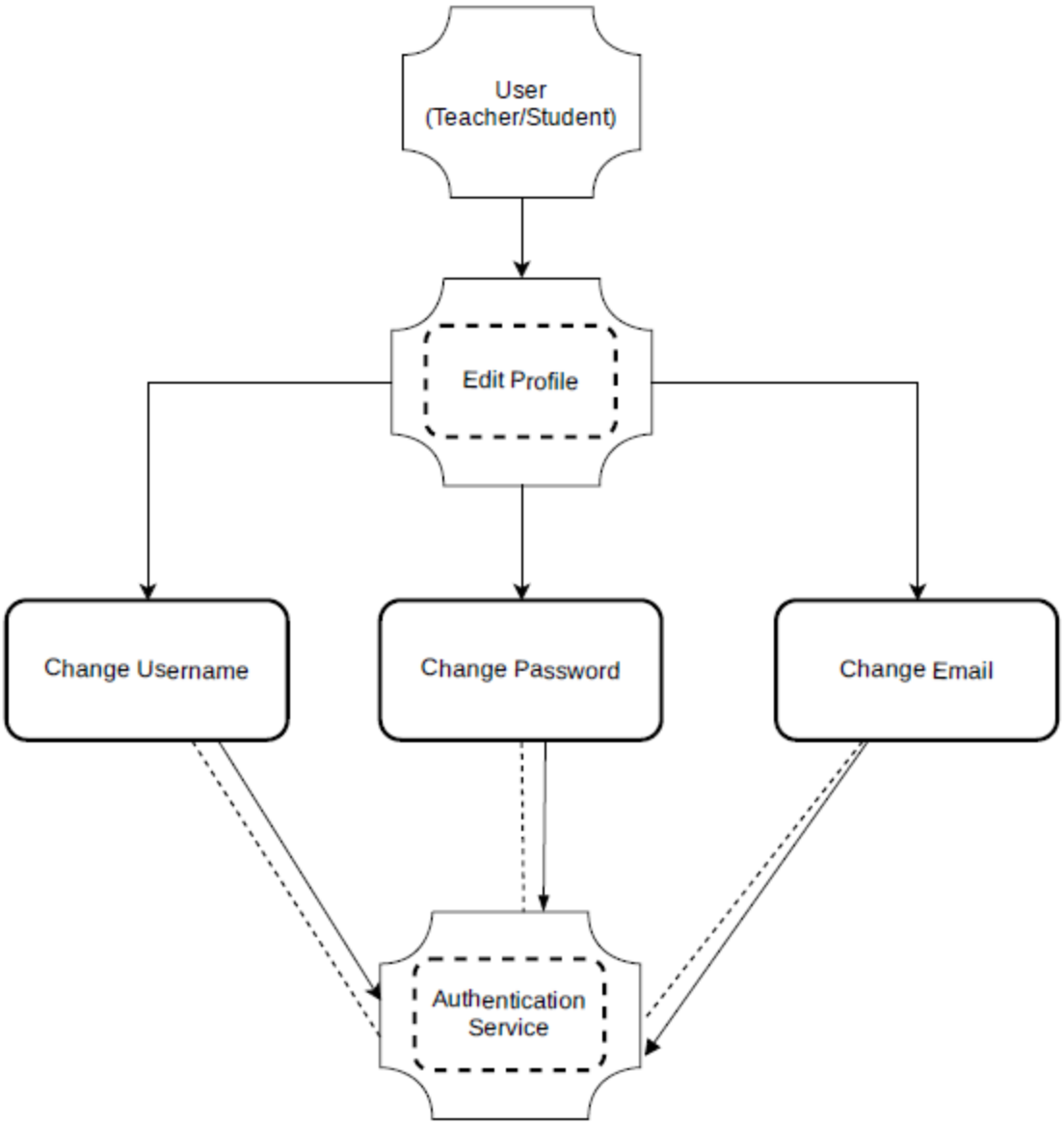
NPC Interaction Component: Architecture Visual Notation Diagram



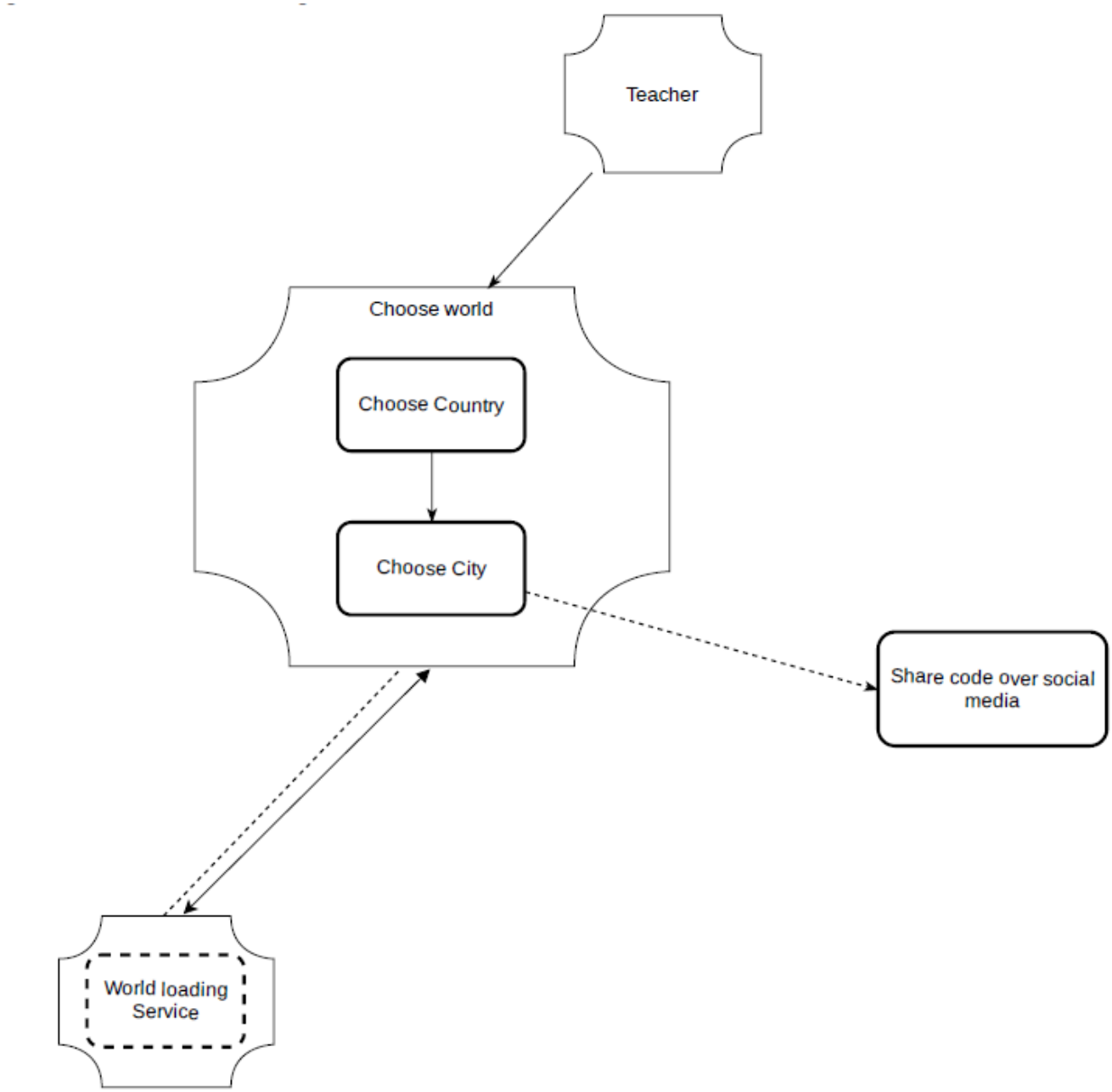
Pause Menu Component: Architecture Visual Notation Diagram



Settings Component: Architecture Visual Notation Diagram



Share to Social Media Component: Architecture Visual Notation Diagram



Tradeoffs and comparisons with other Architectural Styles

Independent Component Architectures (ICAs):

This architectural style was not chosen primarily due to its heavy reliance on message passing as the preferred choice of communication between components. It would be very difficult to determine whether or not messages have been successfully communicated between components if the program was organized in this manner. Ideally, components in ICAs should not be able to transfer control and data between each other, which would be rather unideal for our case.

Client-Server Architecture

Having only a single server with multiple clients may not be very ideal for us, since future changes to the gameplay model may not be able to be adapted to this architecture; flexibility may be compromised.

Event-Based Implicit Invocation

Having an event-based architecture may not be as useful for us, as we have no particular need for other components to express their interest in order to receive information. Data and information should flow in one definitive, organized direction, to prevent undesirable consequences from race conditions, especially since game leaderboards are being updated on the fly.

Data Flow Architectures (DFA)

Data flow focuses on allowing only data to flow between components. CastleSky will require components to exert control over other components as and when required. For instance, the default prohibition of players from accessing certain worlds requires control to flow in one direction, while data input by the player may cause this prohibition to be lifted and require control to flow in the opposite direction. Data flow architectures in general will not suit our purposes very well. Two DFA subtypes and their suitability are outlined below.

Pipe and Filter

At this stage of development, concurrent execution is not a priority, and sacrificing performance (in terms of forcing the lowest clock speed) is not worth it for us.

Batch Sequential

This may not be the best choice for us, since the performance costs are too high

Data Centered Architectures (DCA)

This architectural style focuses on the accessing and updating of widely accessed data. That is, the focus of programs leveraging this style of architecture should be heavily data-centred. Our game, however, does not place such huge reliance and importance on the data - it is simply used to determine what components should be activated. As such, data-centred architectural styles are unlikely to be of much use to us. Modifiability of any representation of data stored would be problematic since updates would have to be rolled out to all clients in order for them to be able to make use of the data. Flexibility would be compromised if we adopt this style.

Repository

Since a passive repository is the data source containing the data, it would not be suitable for it to perform certain actions and be affected by certain triggers, especially since it would be unable to pass control to and from various components.

Blackboard

While mildly more useful than a passive repository, notifications being sent to subscribers may not be of all that much use to us, since components should be connected in an orderly, definitive fashion to avoid problems arising from possible race conditions. Race conditions may occur as a result of notifications being received at slightly different times, causing data to be represented incorrectly.

Virtual Machines

A virtual machine architecture would be rather unnecessary since our game is designed to be capable of running on all major operating systems. Furthermore, educators that subscribe to our game-based education tool are likely to distribute school-funded computing equipment to students, rendering virtual machine-based architecture unnecessary.

Call & Return

This style seemed to be the most promising architectural style for us, especially considering the perspective of developers that will need to develop and maintain the game.

Main Program and Subroutine

While technically still feasible, we felt that adopting a main program and subroutine substyle would heavily limit flexibility in the future, especially if changing requirements dictate integration with other programs and software to be necessary. This would require components to communicate, pass data, and control to each other, which would be a stretch when choosing this substyle.

Layered

As such, the architectural style we chose was the Call and Return style, leveraging on the Layered substyle. This style provides us with the most amount of flexibility while still allowing us to maintain readable and easily testable code.

2. Rationale / Reasons for candidate architecture selection

Our selection of candidate architecture is done specifically to target our Non-Functional Requirements chosen in our System Requirements Specifications, namely **Flexibility, Testability, and Maintainability.**

Flexibility

The layered subtype of the Call & Return Architectural style allows for each layer to be independent and not affect other layers. This represents loose coupling within the layers, allowing for much greater flexibility in terms of allowing modifications of individual components.

Testability

The layered architectural style also allows for a “separation of concerns” to be achieved, wherein each layer only needs to concern itself with the data and controls under its scope. This helps to maintain the Testability component of our software, as each component is distinct and easier to test.

Maintainability

When there are multiple layers in such an architectural style, the concept of ‘layers of isolation’ will greatly contribute to our software’s Maintainability. Changes made in one layer will not directly impact the performance of another. As such, with this layered architectural style, we are able to correct defects and make changes to one layer at any point in time.

3. Subsystem Interface Design (per subsystem)

External Subsystem (Main Game External Application Handler)

CastleSky is designed with the ability to integrate with external software and applications in mind. Although currently only having support for one particular interface, the subsystem is designed to be able to support future integrations with other software. Additional interfaces will be able to be created to facilitate communication between other applications and CastleSky - another demonstration of CastleSky’s flexibility.

To allow easy sharing of content from teachers to students, this subsystem will contain interfaces to WhatsApp and Telegram. The implementation will create a seamless connection between teachers and students as they would be able to easily communicate any updates on the two platforms. These updates can be anything, from new worlds being released to upcoming deadlines. The two interfaces are:

Whatsapp Interface

1. **Name**
   The interface shall be named WhatsApp Broadcaster
2. **Functionality**
   The goal of this interface is to enable teachers to share information with a WhatsApp group or contacts in a convenient manner. With a click of a button, the teacher will be redirected from our platform to the WhatsApp Desktop app or web app. The teacher will then be required to choose the contacts or groups they would like to share with.
3. **Parameters**
   This subsystem will require the context of the message, for instance, if it is a new release or deadline reminder. Additionally, the world’s access code would be required as well. Based on the context the subsystem will generate an URL-encoded text, which is required by WhatsApp.
4. **Return Objects**
   The interface will return a string indicating if the operation was a success or failure. If a failure has occurred, the string will contain an error message.

Backend Subsystem (Game Backend)

This subsystem was developed to provide a unique yet comprehensive backend subsystem for the purpose of facilitating communication between the CastleSky main game and the database. This backend generally follows the model-view-controller architectural style typical of web applications, but we have differentiated and isolated the various functionalities into interfaces.

Authentication Interface

1. **Name**
   The interface shall be named AuthenticationManager.
2. **Functionality**
   a. This interface will give or deny access to users based on their credential validity. First, it will check if the email provided is registered in the database. Then, it will perform hashing on the given password and verify if it matches the one in the database. If both checks are successful, the interface will see if the account is verified.
   b. This interface will also allow for user registration by verifying the email address uniqueness and subsequently creating an account

3. **Parameters**
    a. In the event of user login, both a username and password must be provided before the interface can perform its checks with the database.
    b. In the event of user registration, username, password, email address and class name must be provided before the interface can perform its checks with the database. Input validation such as password complexity requirements will be performed on the front-end.
4. **Returned Objects**
    a. In the event of a successful login, the interface will return a json object containing the logged in user's particulars, as well as a status indicating if the operation was a success or failure. If a failure has occurred, the string will contain an error message.
    b. In the event of successful registration, a status indicating success is returned. The user will be redirected back to the login page.

Scoring Interface

1. **Name**
   The interface shall be called the CityScoreInterface
2. **Functionality**
   This interface contains various methods required to perform CRUD operations with student scores. This includes posting scores to the leaderboard after students finish playing a game, as well as obtaining consolidated data containing the scores. These will typically be used to display the leaderboard, as well as the summary reports for the teachers.
3. **Parameters**
   Generally, posting to the database requires the parameters UserId and Score to be supplied, while querying the database for all scores will tend to require some combination of CountryIds, WorldIds and CityIds.
4. **Returned Objects**
   A status message indicating the success or failure of the called method. Depending on the method, a list of Scores may be returned as well.

Minigame Data Interface

1. **Name**
   The interface shall be called the MinigameDataInterface
2. **Functionality**
   This interface shall contain various methods to perform CRUD operations on any minigame related **data** in the database. This is typically used when loading data from the database for the minigame.
3. **Parameters**
   When loading game data for the database, for now, only the cityId will be required as parameters. However in future implementations, if the contents of the minigame questions are desired to be dependent on the student's proficiency (which can be obtained through some action performed using the student's userId), this may be easily passed in as a parameter as well.
4. **Returned Objects**
   The requested minigame data, which could include questions and answers, or even game data assets.

Minigame Subsystem (Minigame Main Game & Main Game Minigame)

In CastleSky, each minigame is designed to be innately separate from the Main Game. We discovered that this provides a level of Flexibility and Extensibility that would be very difficult to replicate if minigames were developed in the Main game itself. While each minigame will be designed independently and would be capable of functioning alone, integration with the Main Game of CastleSky would require every minigame to be compatible with our Minigame Access Interface. This allows external developers to add much more functionality to CastleSky in the form of minigames, which further highlights the Flexibility of CastleSky.

Minigame Access Interface

1. **Name**
   This interface shall be called the MinigameAccessInterface
2. **Functionality**
   This interface validates and facilitates the movement between the Main game and the Minigame
3. **Parameters**
   Methods in this interface tend to commonly accept the UserId of the player, as well as other key attributes required to play the minigame.
4. **Return Objects**
   Generally, a status code indicating the success or failure of data validation will be returned. The player will be sent to the Minigame if validation succeeds.

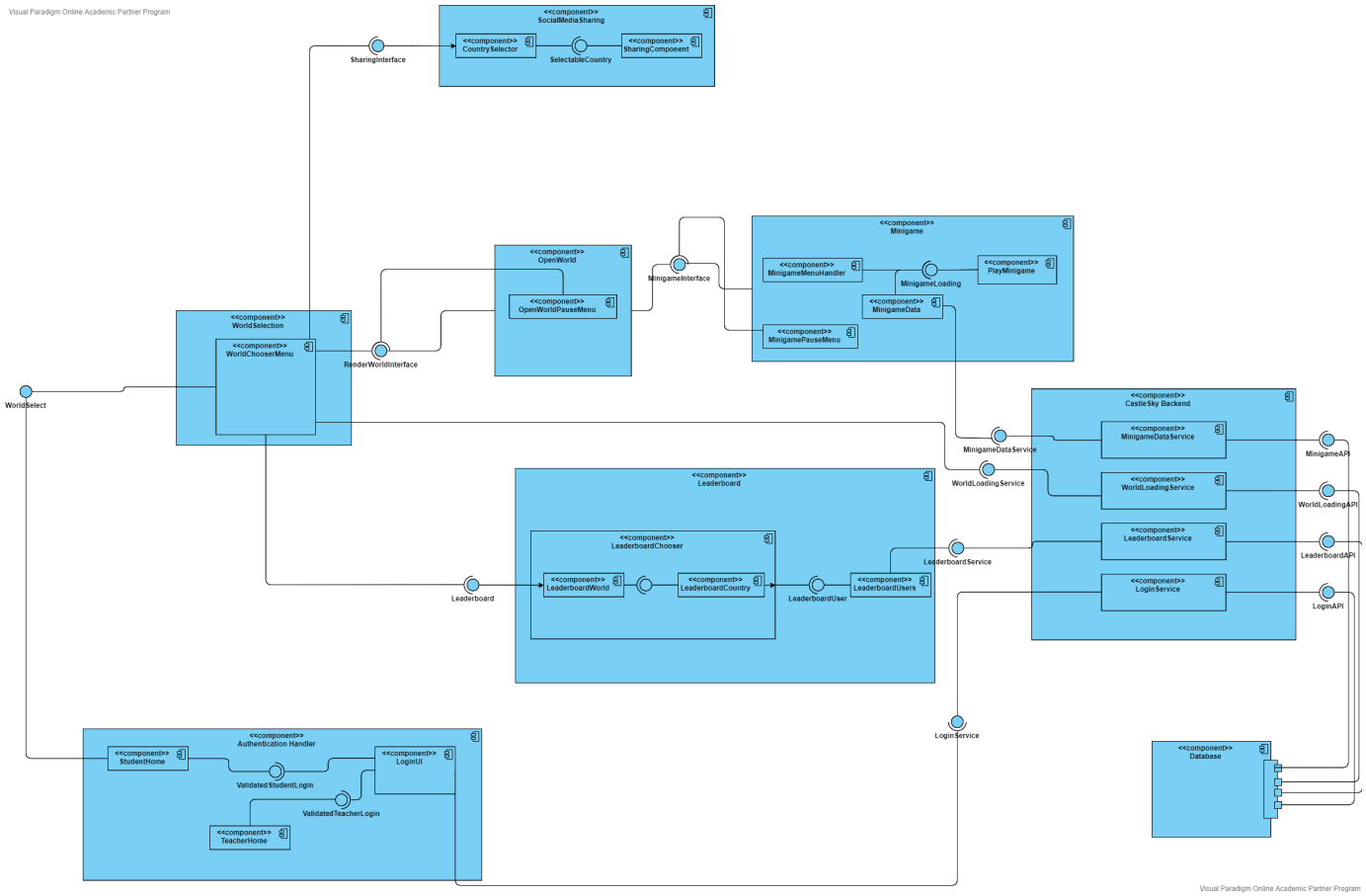
Leaderboard Interface

1. **Name**
   The interface shall be named LeaderboardManager.
2. **Functionality**
   The objective of this interface is to perform queries on the database and populate the leaderboard based on the filter given. Whenever a

user chooses to access the leaderboard, they will be provided with a dropdown menu to choose from worlds they have access to. The interface will then query the database using the user's input and sort the result based on the scores. This data will only be populated and retrieved when associating with a particular minigame.

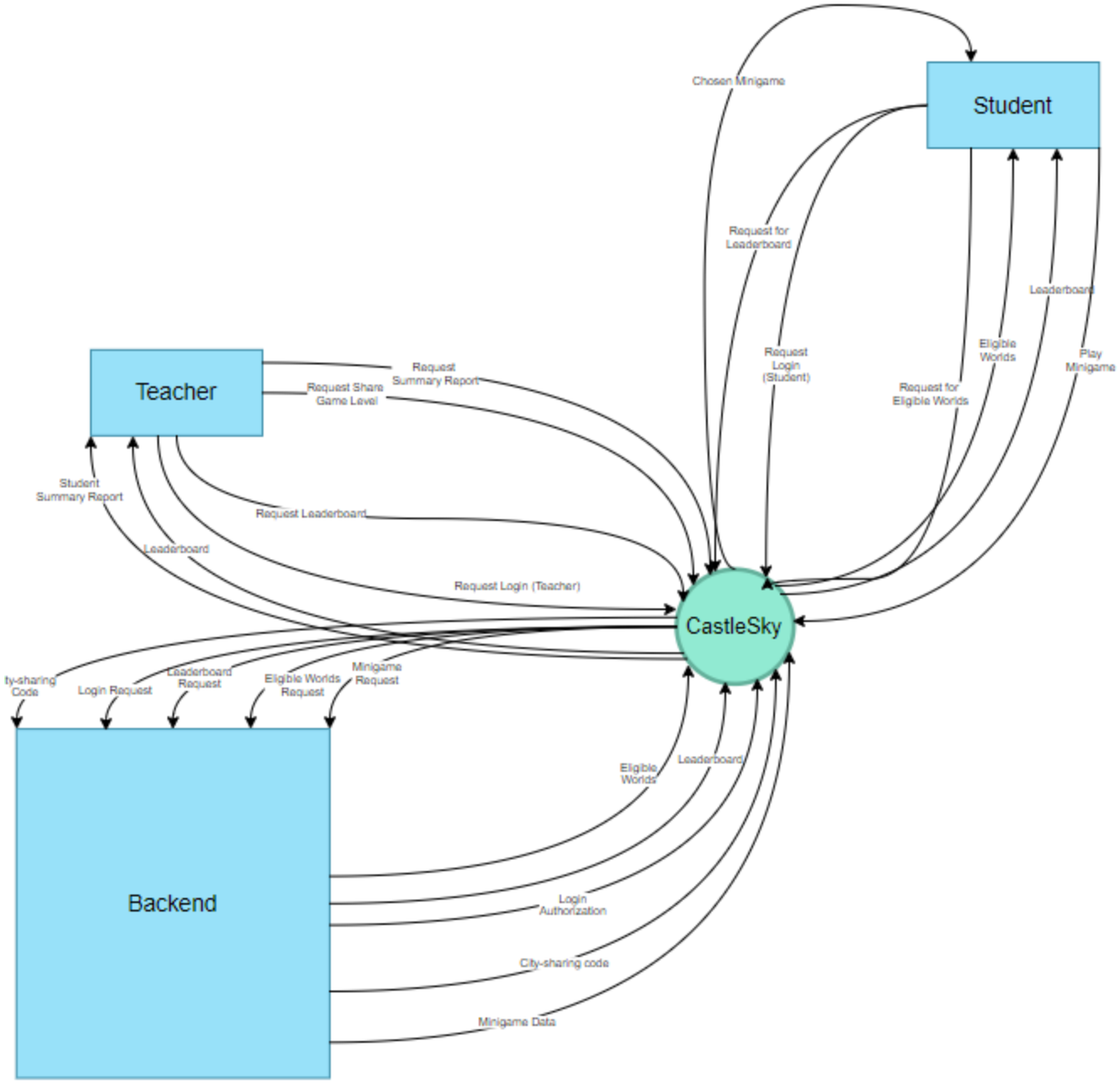
3. **Parameters**
   The user may indicate which world's leaderboard they would like to view, otherwise, the leaderboard for the current minigame will be displayed
4. **Returned Objects**
   The interface will return an array consisting of usernames and their respective scores in the given world.

Component Diagram



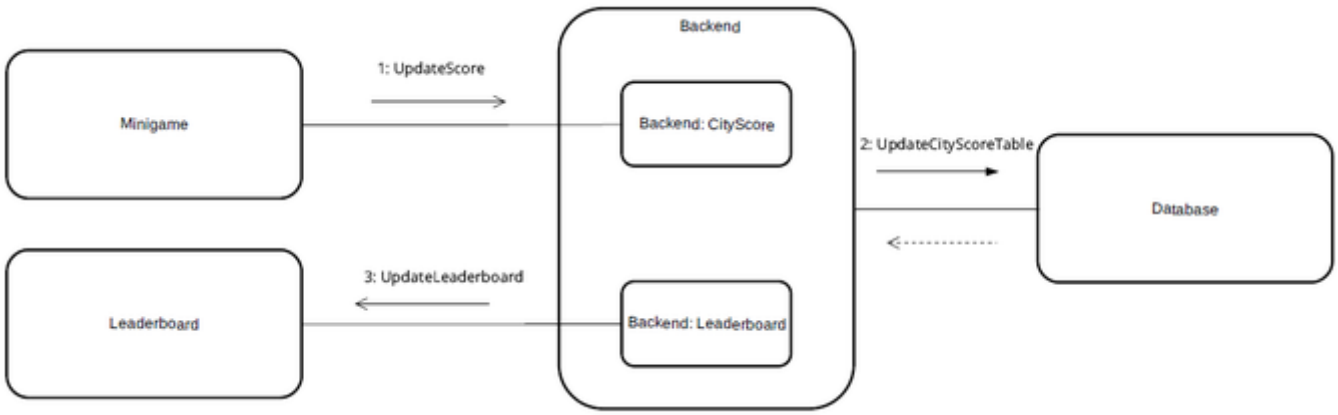
Context Diagram

This context diagram illustrates the interactions between the various controlling classes and the main game.



Communication Diagram

This communication diagram illustrates the communication between the backend components and several of the active game components (specifically the minigame and leaderboard)


Backend
Minigame
1: UpdateScore
Backend: CityScore
2: UpdateCityScoreTable
Database
Leaderboard
3: UpdateLeaderboard
Backend: Leaderboard