NANYANG TECHNOLOGICAL UNIVERSITY

CZ3003 - Software System Analysis & Design

# Candidate Architecture and Subsystem Interface Design

Project Name: Game of Thrones
Group Name: Team TWO
Lab group: TDDP1
Date of Submission: 07/09/2021

| **Group Member** | **Matriculation Number** |
|---|---|
| Chee Jia Yuan | U1921773A |
| Chong Jie Sheng | U1920968D |
| Ernest Ang Cheng Han | U1921310H |
| Joshua Toh Sheng Jie | U1921471F |
| Koh Tzi Yong | U1920076C |
| Lek Zhi Ying | U1922765K |
| Leong Hao Zhi | U1920469K |
| Li Zheng Jun, Jefferson | U1922429B |
| Lim Guo Quan | U1920769A |
| Loo Yi Ying Phoebe | U1921301A |
| Remus Neo Keng Long | U1922206F |
| Tan Wei Lun | U1822950L |

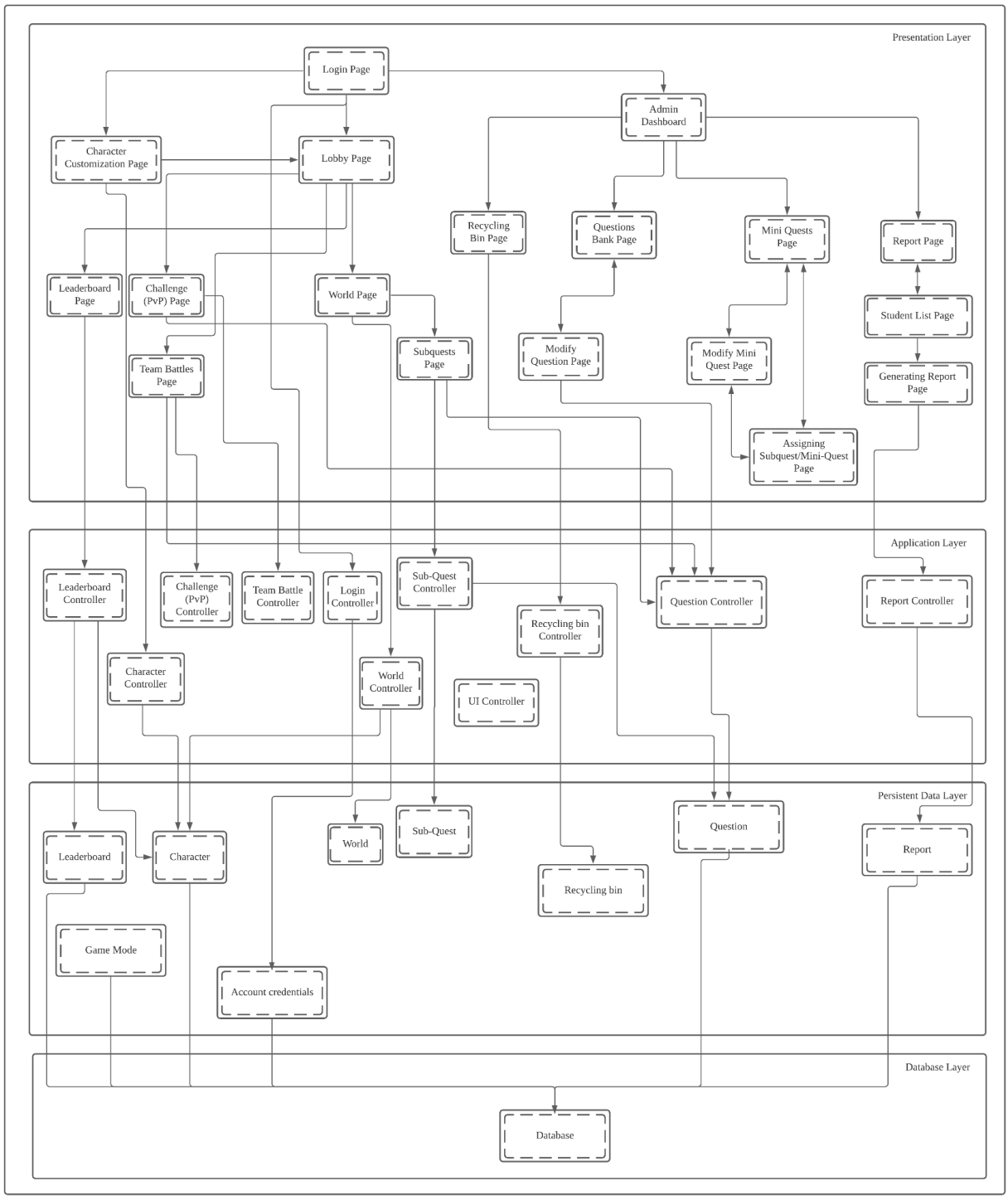
# Table of Contents

# 1. Introduction

Architecture design exposes the structure of a system while hiding the implementation details. The architecture also focuses on how the elements and components within a system interact with one another.

A good architecture design consists of four software design principles: modularity, single responsibility principle, principle of least knowledge, and open close principle. Collectively, these principles will result in a comprehensive design with a clear separation of concerns, low coupling and high cohesion.

# 2. Candidate Architecture




Given that there is no architectural design that is satisfactory over every design consideration, our group chose to use the layered-architecture design.

There are three distinct advantages of using layered-architecture style:

1. Support designs based on increasing level of abstraction
2. Support enhancement of functionality by introducing new components
3. Support reuse of lower layer components while upper layer components varies

The Presentation Layer includes all the boundary classes that make up the user interfaces. These classes are responsible for displaying the information that is relevant to users.The user interface of our game can be split into 2 categories - for students and teachers. For both students and teachers, the user interface starts from the login page. For students, this then extends to the character customization page and the lobby page. From the lobby page, it leads to other features that are available for students, including the world page, subquest pages, leaderboard page, challenge (pvp) page and team battles page. For teachers, the login page will extend to the administrative dashboard on a website. From the dashboard, it leads to features available for teachers, including questions page, modifying questions page, recycling bin page, modifying mini quest page, assigning subquest page, student list page and generating report page.

The Application Logic Layer includes all the control classes that form the logic of the program. These control classes are called by the boundary classes in the presentation layer to perform functions on the entities involved. In addition, they will retrieve relevant information from the database and external APIs.

The Persistent Data Layer includes all the entity classes, which contains the attributes of each entity. The entities are stored in a database and include questions, answers, students, teachers, leaderboard etc.

The Database Layer includes data managed by the program. Searches, insert, delete and update operations are executed here. Example operations include storage of students' answers and modification of questions by teachers.

# 3. Rationale for Candidate Architecture

## 3.1 Advantages

### 3.1.1 Encapsulation and Decoupling

The autonomous layers in a layered architecture allow for separation of concerns by loosely coupling components. By organizing the layers in this manner, the components within the specific layers would also adhere to the Single Responsibility Principle, dealing with only one facet of a feature. For example, the login page in the presentation layer is only involved in the display while the application logic layer is responsible for the authentication process. This allows for easier addition of new features, increasing flexibility of our program.

### 3.1.2 Ease of Testing

To test a component, layers other than the one containing the component of interest can be mocked, which is to apply a dummy piece of code that is verified to be called correctly as part of the test, or stubbed, which refers to applying a dummy piece of code that allows the program to run, with no concern for the results. This allows isolated testing within a business component, as well as mocking of a business layer to test specific screen functionality.

### 3.1.3 Ease of development

The Single Responsibility Principle of each component also makes it easier for a large group of developers like us to split tasks and not repeat work processes unnecessarily.

## 3.2 Limitations

### 3.2.1 Elements of Database-Centric architecture

Critics of this architecture have argued that it is very database-centric. As everything flows down layers, it's often the database that's the last layer. However, as our program is currently being developed on a relatively small scale, the limitations of database-centricity in a layered architecture can be minimized as there aren't many downstream clients.

### 3.2.2 Performance

There is a possibility of creating too many layers, resulting in the problem of lasagna architecture and causing an unnecessary increase in code complexity. This results in higher overhead, reducing performance. However, this can be easily overcome by limiting the number of layers we use to only the necessary ones.

## 3.3 Conclusion

Since the weaknesses of the layered architecture can be minimized for this project, the benefits heavily outweigh the drawbacks, the layered architecture seems to be the best starting point as the candidate architecture system for this project.

# 4. Evaluation of Alternative Architecture Styles

## 4.1 Independent Components

In an independent component architecture, the design of the software is decomposed into separate logical components with well-defined communication interfaces with methods, events and properties. It has a high level of abstraction and this allows for component reusability.

As a result, independent component architecture is suited for systems which have separate environments which are not context specific, often replaceable and independent. This brings a lot of benefits such as high reusability and encapsulation, ease of maintenance and faster development.

However, in a game-based application such as ours, the subsystems are closely integrated and frequently interact for the game features to function. Subsystems will send and receive data from each other, which is not optimal in an independent component architecture where each abstraction component does not depend on other components and interaction between components require complicated protocols. Therefore such an architecture would not be suitable for our system's purposes.

## 4.2 Data Flow

In a data flow architecture, there are 2 main styles – batch sequential and pipe-and-filter.

Our system is geared towards the batch sequential style, and a typical workflow is illustrated below.

- A user in the game completes a quest and earns 100 points
- The game client sends a PUT request to our application server via the `/users` endpoint to update the user's points in the system
- The application server issues an UPDATE query against our database instance
- The transaction is committed, and the application server returns a `204 No Content` success response to the game client

This approach, however, suffers from a few drawbacks.

- High latency and low throughput - individual components (application server and database) serve as bottlenecks in this sequential method of data processing
- Lacks the possibility of concurrent data processing - the downstream game client will have to wait for a successful write to the database and a 204 success response before it can continue

Hence, the data-centred architecture was eventually not chosen due to the complexity involved in addressing these drawbacks for this project. Even if the pipe-and-filter architecture was adopted, using message queues as our intermediate “pipes”, the additional complexity involved in configuring them was too high (for instance, in Kafka, topics, message brokers, and the partitioning of topics must be configured via ZooKeeper).

## 4.3 Data-centred

In a data-centred architecture, there are 2 main styles – repository and blackboard.

Our system is geared towards the repository style, where our backend application server instance (the active client) will actively issue CRUD operations against a database (the passive data store). These operations are issued as a set of serializable database transactions.

The following are benefits of the repository style in a data-centred architecture.

- Improved data integrity, via weekly routine backups of our Postgres tables, allowing easy reverts to these backup points in case of data corruption, and restoration in case of data loss
- De-coupling of downstream clients as all clients go through a central API to perform CRUD operations on entities within the database

There are, however, drawbacks to this style.

- The data store acts as a single point of failure. However, this can be mitigated by setting up read replicas in a cluster, ensuring high availability in our system. Azure Database for PostgreSQL and PostgreSQL in Amazon RDS both support this.
- The data store becomes a bottleneck when there’re many downstream clients. However, horizontally scaling read-heavy workloads can easily be done via a master-slave architecture, where additional read replicas (slaves) are spun up to handle read workloads, with a single write master, usually at the cost of slower writes if synchronous

replication is used. Horizontally scaling write-heavy workloads is more complex, where horizontal partitioning (sharding) is applied at the expense of increased cost in joins involving these sharded tables

Hence, the data-centred architecture was eventually not chosen due to the complexity involved in addressing these drawbacks for this project.

# 5 Subsystems

We have identified several subsystems within the whole game, but they can be further categorised into subsystems within our Game and Admin Service. The decomposition of the project into different subsystems will allow developers to work concurrently and minimize task dependency, thus reducing the development time. The modularity of our systems makes it easier to detect errors because they can quickly be narrowed down to a particular subsystem and thereafter the specific function. Furthermore, the separated subsystems will also be easier to design and test, allowing for easier management too. The breakdown is as such:

**Game Service:**

- Verification
- Leaderboard
- Quiz
- Challenge

**Admin Service:**

- Question
- Report

**Game Subsystems:**

| Interface Name | **VerificationService** |
|---|---|
| Functionality | Allow students to sign up using an NTU email address via an authentication method of a 6-digit verification pin. Once successfully signed up, students will be transferred to the login page to enter the pin. |
| Rationale | An independent component which can be reused for both students and teachers for authentication purposes during sign ups. |
| Parameters Required | EmailAddress : String<br>Student : Student<br>Teacher : Teacher |
| Subsystem Interaction | Uses FirebaseService (Data Access Layer) to access database |

| Interface Name | **LeaderboardService** |
|---|---|
| Functionality | To allow students to view their individual scores and ranking. Students can view other students' scores and rankings. Comparison can then be done by students to see where they stand. |
| Rationale | An independent component which is separate from our QuizService to further break down our component design for easier coding due to the large group size. Easier scalability of functions too. |
| Parameters Required | Students : Array<Student><br>ActiveChallengeList : Array<Object> |
| Subsystem Interaction | Uses FirebaseService to access database |

| Interface Name | **QuizService** |
|---|---|
| Functionality | Retrieve all questions, options, and answers for every aspect of the game. Handles everything related to questions of quizzes, including difficulty levels. |
| Rationale | As explained in LeaderboardService, we want to shift our functionality of leaderboards and quizzes separate for easier scalability. |
| Parameters Required | Student : Student<br>Difficulty : Enum<br>QuestionId : String |
| Subsystem Interaction | Uses FirebaseService to access database |

| Interface Name | **ChallengeService** |
|---|---|
| Functionality | Allow for PvP within the game, whereby players are able to request a challenge with another player, and the other player is able to choose whether to accept the challenge. Players can win by amassing more points during the challenge. |
| Rationale | This functionality requires a separate component due to its complexity, hence the creation of a separate controller component. |
| Parameters Required | Name : String<br>Students : Array<Students><br>StudentToRemove : Student<br>Student : Student<br>Difficulty : Enum<br>PointsEarned : Int |
| Subsystem Interaction | Uses FirebaseService to access database |

**Admin Subsystems:**

| Interface Name | **QuestionController** |
|---|---|
| Functionality | Allows teachers to create, edit, delete and assign questions for the quizzes. |
| Rationale | Similar to Game Subsystems design, we want to separate the leaderboard and question components for easier scalability and coding within a large group. |
| Parameters Required | Student : Student<br>Difficulty : Enum<br>QuestionId : String |
| Subsystem Interaction | Uses FirebaseService to access database |

| Interface Name | **ReportController** |
|---|---|
| Functionality | Allows the teacher to view either an individual student's performance, or the whole cohort's overall performance via a report. |
| Rationale | This functionality requires an independent component from the QuestionController as it requires a query from our Firebase, and requires some calculation (logic) to get the overview of students' performance. |
| Parameters Required | Student : Student<br>Students : Array<Students> |
| Subsystem Interaction | Uses FirebaseService to access database |

**Below is our class diagram which links both our Game service and Admin service together, which was not elaborated on in our previous lab proposal.**