SWEN90007 Part 2

Report – Group ‘tv_addicts’

Architecture Design Pattern record

Luyun Li 1586333
GitHub name: lucy-lulu
Email: luyunli@student.unimelb.edu.au

Haitian Wang 1467513
GitHub name: Morty931
Email: haitianw@student.unimelb.edu.au

Mingda Zheng 1382885
GitHub name: MnnDa
Email: mingdaz1@student.unimelb.edu.au

Sameer Sikka 1169800
GitHub name: SamSike
Email: ssikka@student.unimelb.edu.au

1. Project introduction

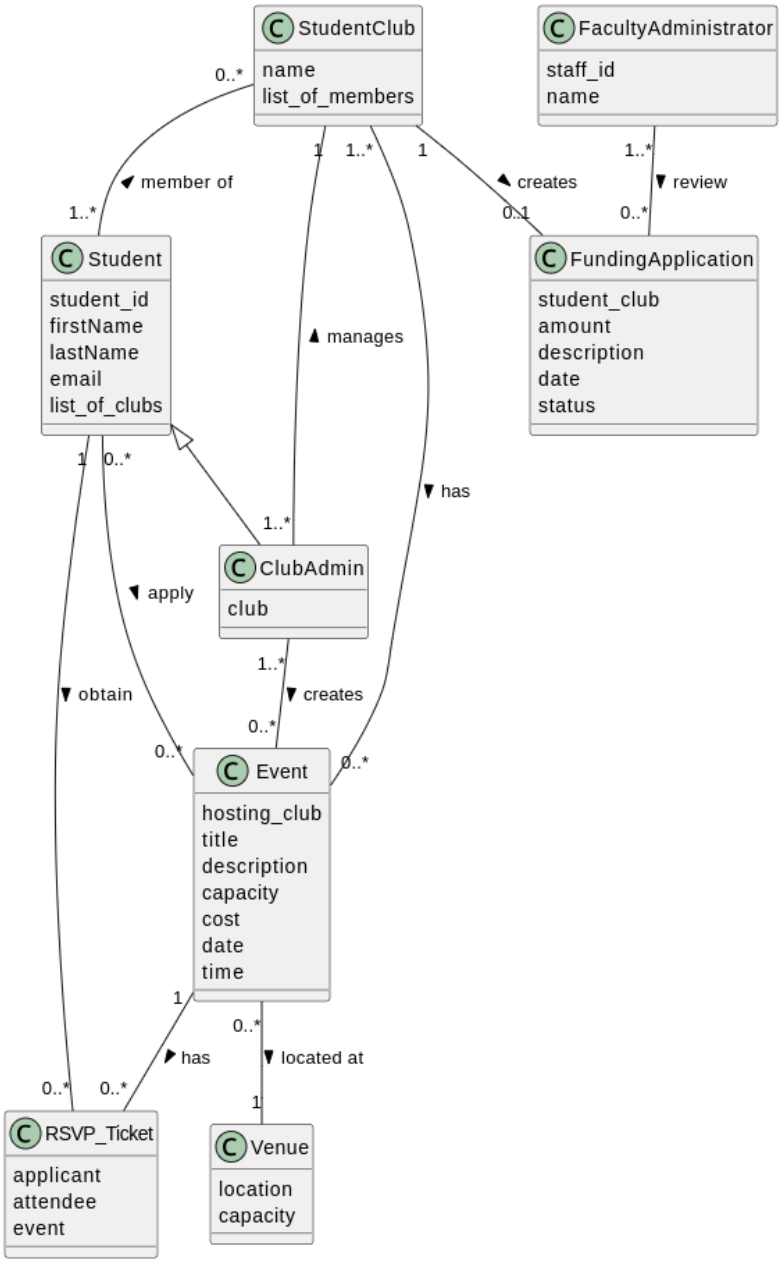
The main aim of the project is to create an application that be used by Students at The University of Melbourne to manage on-campus student clubs and events. Our system promotes student engagement by managing events hosted by individual student clubs. The App will allow students to create, search and RSVP for events while enabling club administrators to manage events, club members and funding requests. The system will serve as a central platform for students to organise and participate in extracurricular activities, making event management simple.

The system now enables multiple user stories: students search for upcoming events, RSVP events, create and manage events, and handle club administration tasks. In addition, the system supports student clubs in applying for and managing funding for their events.

This document records the architecture, object relation structure and behavioral design, using domain model and class diagram, entity relationship diagram, sequence diagram and use case diagram to represent them respectively. This report also serve as a Architectural Design Record(ADR) document that record all the design patterns we implemented, including inheritance pattern, identity field, foreign key mapping, association table mapping, data mapper, lazy load, unit of work, and authentication and authorization, each of them is documented as an ADR that can be tracked below. ADR manages to be a useful communication tool between team members, and also improves the overall control of the development process. It is a pathway to resolving disputes issues and record implementation ideas and steps of each design pattern we choose to use comprehensively .

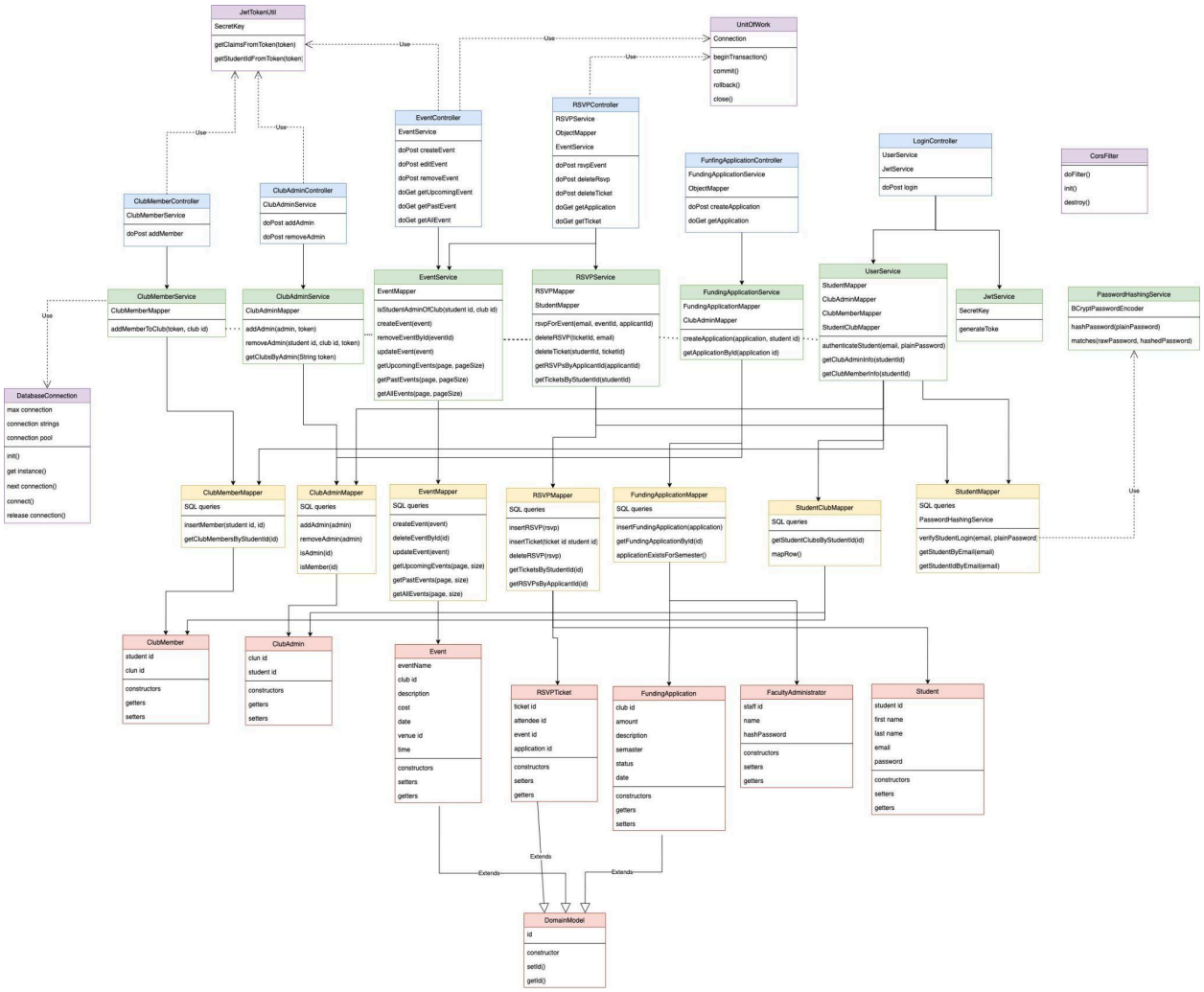
2. Architecture patterns

2.1 Domain Model



The detailed explanation about domain model design can be seen in part 1A in our github repository, no need to repeat it here .

2.2 Class Diagram



Class diagram is a traditional method to show the software structure design, it describes system structure by presenting all the classes, attributes and methods, and using an arrow to represent the relationship between different classes. In this class diagram, blue is the controller layer, which is also the layer that is closest to the frontend. Green represents the service layer, which mainly used for connection database and link between controller and mapper. Yellow is the data mapper layer, which serves as a sending queries to database, the detail design of this layer is in 3.5. For the red one, they are the domain model layer, storing the attributes of each domain model. The purple package represents shared usage including database connection(used by all the class in serves layer, but to simplify the figure only link with one of them), unit of work, JwtTokenUtil, and for CorsFilter, it is not directly used by any class, but serves as tool to receive correct request from front-end.

The original version of class diagram is stored in our github doc/figure, it is not clear enough here since the diagram is too large.

2.3 Inheritance pattern

Inheritance pattern provides a way to handle shared behavior and attributes across related entities in a relational database. By allowing common attributes and methods to be defined in a parent class, child classes can inherit and extend this behavior. This promotes code reuse and enhances the maintainability of application, especially when dealing with complex entities like events, RSVPs, clubs, and students.

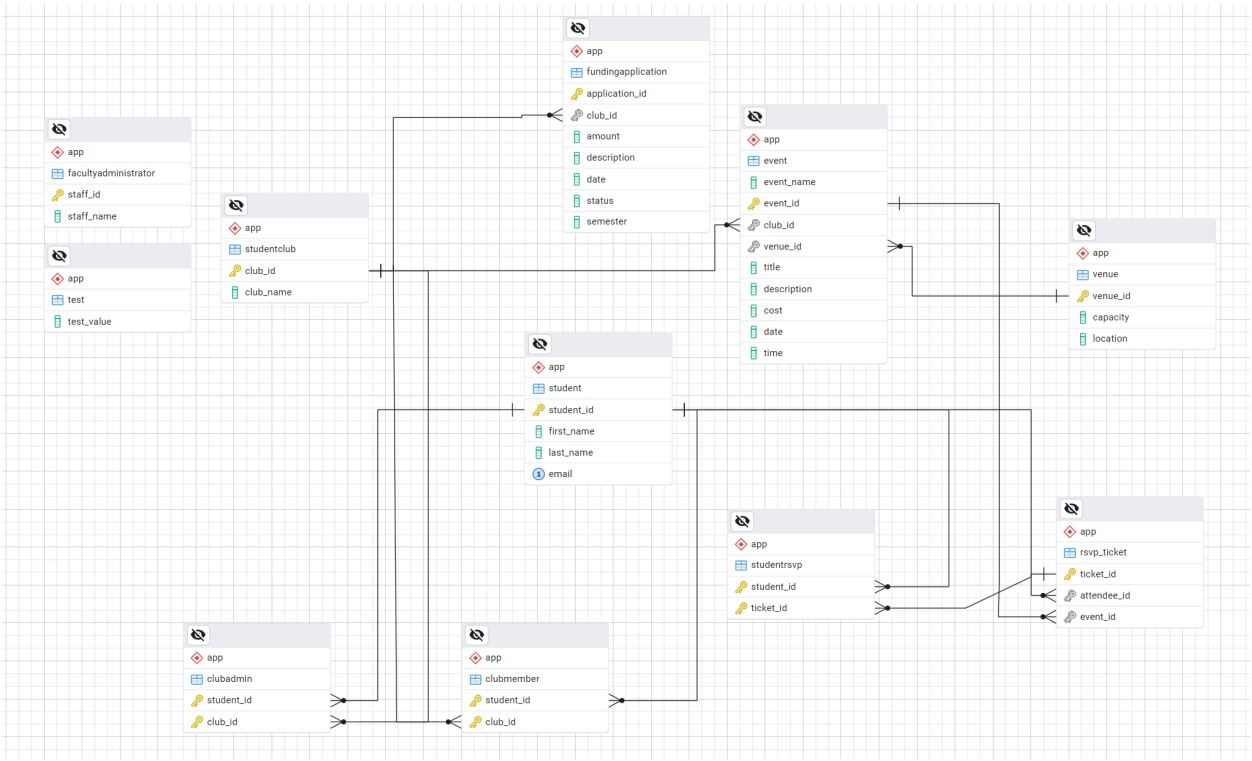
Title	Implementing the Inheritance Pattern in the Club management system' s Domain Model
Status	Accepted
Context	The Club management system handles complex business logic for managing entities like students, events, clubs, and RSVP tickets in a university's event management system. As the application evolves, many of these entities share common characteristics, such as having an `id`, and follow similar patterns for data access and manipulation. The goal is to improve maintainability, reduce redundancy, and promote code reuse. For instance, all entities such as `Event`, `RSVPTicket`, `Student`, and `StudentClub` need a unique identifier (`id`) and basic methods for managing this ID (getters/setters). If each class independently manages its identifier logic, it results in duplicate code across the system. To address this, we decided to implement the **Inheritance Pattern**, introducing a base class `DomainModel` that contains the shared logic, which is then inherited by individual entity classes. This approach aligns with the **DRY principle** (Don't Repeat Yourself), ensuring that common functionality is abstracted into a parent class and reused across child classes.
Decision	We will implement the **Inheritance Pattern** by introducing a `DomainModel` class that contains shared attributes (such as `id`) and methods for managing these attributes. Other domain models like `Event`, `RSVPTicket`, `Student`, and `StudentClub` will extend this `DomainModel` to inherit its functionality and reduce redundancy in the codebase.
Implementation Strategy	**DomainModel Class:** ● Create a base class called `DomainModel`, which will hold the common attribute `id` and its associated getter and setter methods. ● The `DomainModel` is the parent class for the other domain models. **Inheriting Classes:**

	• The domain models like `Event`, `RSVPTicket`, and `StudentClub` will inherit from `DomainModel` to reuse the common logic related to the `id`. **Codebase Impact:** • All models that require `id` and similar behavior will extend the `DomainModel` class, identify common functions and realize code reusing pattern. • Any future changes to shared functionality (e.g., updating how IDs are managed) can be applied at the `DomainModel` level, without requiring updates to each individual class.
Consequences	**Positive:** 1. **Reduced Redundancy:**Common code related to the management of `id` is abstracted into the `DomainModel`, reducing the repetition across different domain models. 2. **Code Reusability:**Reusing the common logic from the `DomainModel` encourages code reusability, adhering to the **DRY principle**, making the codebase cleaner and easier to manage. 3. **Alignment with Object-Oriented Programming:** Inheritance pattern aligns with Object-Oriented Programming principles, this make modular code cleaner, and more maintainable. **Negative:** **Couple tighter:** The use of inheritance introduces a tighter coupling between models and the base `DomainModel` class, meaning changes to the parent class could have unintended effects on child classes.
Alternatives Considered	**1. Composition over Inheritance:**Instead of using inheritance, we separate the use. Each domain model can contain instance of a separate `Identifier` class responsible for managing the `id` field. This could have provided more flexibility, as changes to the `id` logic wouldn't directly affect other models. **Positive:** Composition can add flexibility. Each domain model can independently manage its `id` field through a separate `Identifier` class. This decouples the `id` logic from the domain model, allowing for easier modifications or enhancements to the identifier logic without impacting other models. **Negative:** This approach might introduce some additional complexity in terms of managing interactions between the domain models and the identifier logic. **2. Interfaces or Mixins:** Another alternative is to use interfaces or mixins to define the shared behavior for managing identifiers. This would allow domain

	models to implement only the specific behaviors they require, while avoiding the tighter coupling that inheritance introduces. **Positive:** This approach would offer more flexibility in terms of which models need certain behaviors. And also allows greater control, since each model can include only the features it needs, avoiding the inheritance of unnecessary functionality. **Negative:** But it might increase the amount of boilerplate code and reduce the simplicity of a unified inheritance structure. This approach may also make the codebase become more fragmented, as different models could manage shared behavior in varied ways. This could potentially make the system harder to maintain and reduce the overall consistency.
Consequences of Not Implementing	Without this inheritance pattern, we would have higher redundancy in our domain model layer. This would make the code harder to maintain, as any changes to this shared logic would require updates across all models individually. With more modification, increased the risk of bugs and inconsistencies in our system. Moreover, not implementing this pattern would go against best practices in **Object-Oriented Programming (OOP)**, resulting in a less modular and reusable codebase.

3. Object relation structure design patterns

3.1 Entity Relationship Diagram



This ER diagram describes the database structure of a student club event management system. The main entities include students, clubs, activities, venues, and funding applications. Students can be members or administrators of multiple clubs, which are managed through the clubMember and clubadmin association tables. Clubs can create events, which are associated with venues, and record student registration information through the RSVP system (rsvp_ticket and studentRSVP). At the same time, clubs can submit funding applications, recording information such as the application amount, description, date, and status. This design ensures the connection between core functions such as students, clubs, activities, venues, and funding applications.

3.2 Identity Field

The Identity Field Pattern provides a reliable way to ensure that each entity in our club management application is uniquely identifiable. This guarantees data integrity by enabling consistent references between entities like clubs, events, and RSVPs through foreign keys. By automating the generation of unique identifiers, we simplify the management of relationships, reduce manual effort, and ensure accurate processing of operations. Below is our ADR for the Identity Field design pattern.

Title	Implementing the Identity Field Pattern in the Club Management System
Status	Accepted
Context	In the Club management system, multiple entities must be uniquely identifiable. The identity field pattern ensures that each entity in the system is unique. This is necessary when implementing their relationships, such as events that require a club ID as a foreign key. Club management systems do not need to update multiple domain objects at the same time. We implement the identity field schema by automatically generating meaningless, system-assigned keys in each entity's respective table. Then, these generated keys can be used as foreign keys when referencing related entities.
Decision	When creating each domain object, a system-generated unique identifier is added to distinguish and uniquely identify it. The references between entities are also realised through this unique identifier.
Implementation Strategy	1. Each entity in the system adds an automatically generated and non-repeating id identifier: such as StudentClub, Event, RSVPTicket, and FundingApplication : we have ClubID, EventID, FundingID. 2. The foreign key will refer to the entity's primary key to implement the entity-to-entity relationship. When using foreign keys, data integrity is ensured by **CASCADE**. When parent data is deleted, child data is also deleted, for example in the case of events, when a student is deleted from the database, the corresponding event is updated. In this way, data integrity is further ensured through unique identification and accurate business processing.
Consequences	By using unique identifiers, entities can be maintained with foreign keys to each other, ensuring data integrity. By using the identity pattern, entities can be created and managed in a simpler, more predictable way. It also allows for more automation of the system and less manual administration.
Alternatives Considered	The system can use natural keys to maintain uniqueness, e.g. club names, event names, student names etc. However, if natural keys are used, there is a risk of naming conflicts. There is also the option of assigning ids manually, but this complicates the system's processing.
Consequences of Not Implementing	Without the identity field schema, complexity would increase and managing relationships between domain objects would become cumbersome. Using natural keys requires additional logic to maintain data integrity. Because each entity needs to be uniquely identifiable, without the use of unique identity fields, the system schema becomes redundant and may result in not being able to identify a specific instance, for example, when booking events, if

	there are multiple events, the system will not be able to process them according to the correct logic.

3.3 Foreign Key Mapping

Foreign Key Mapping offers ways to establish relationships between tables in our database by creating links from one table to another through foreign key constraints. This ensures the data integrity by enforcing- a field in one table corresponds to a primary key in another. It allows the database to efficiently manage relationships between entities like students, events, RSVPs, and clubs. Below is our ADR for Foreign Key Mapper design pattern.

Title	Design the Foreign Key Mapper Pattern for Club manage system
Status	Accepted
Context	In our database structure design, we needed to ensure enough data consistency and relationships between different entities, such as students, clubs, events, and tickets. Foreign key constraints are crucial to maintain that integrity between tables. These keys provide links between records(data rows) in one table to the primary keys of another, ensuring that data remains consistent and no invalid records stored in our database.
Decision	We decided to use **Foreign Key Mapping** in our database schema to enforce relationships between the tables. This decision allows us to link tables such as `RSVP_Ticket`, `Event`, `ClubAdmin`, and `ClubMember`, ensuring referential integrity across the database.
Implementation Strategy	**RSVP_Ticket Table:** • Foreign Key: `attendee_id` → `Student.student_id` • Foreign Key: `event_id` → `Event.event_id` **StudentRSVP Table:** • Foreign Key: `student_id` → `Student.student_id` • Foreign Key: `ticket_id` → `RSVP_Ticket.ticket_id` **Event Table:** • Foreign Key: `club_id` → `StudentClub.club_id` • Foreign Key: `venue_id` → `Venue.venue_id` **ClubAdmin Table:** • Foreign Key: `student_id` → `Student.student_id`

	• Foreign Key: `club_id` → `StudentClub.club_id` **ClubMember Table:** • Foreign Key: `student_id` → `Student.student_id` • Foreign Key: `club_id` → `StudentClub.club_id` **FundingApplication Table:** • Foreign Key: `club_id` → `StudentClub.club_id`
Consequences	The database will ensure that records in related tables (e.g., students, clubs, events) are consistent and prevent all the invalid records. Foreign Key Mapping also includes **Cascading updates and deletions** to maintain relationships when changes occur to primary records (e.g., when a club is deleted, associated events or RSVPs may also be deleted, align with unit of work pattern).
Alternatives Considered	We can ensure consistent and valid relationships manually between tables without using foreign keys. Efficient Joins can be used in SQL queries to replace foreign keys mapping design.
Consequences of Not Implementing	Without foreign keys, there's higher risk of inserting wrong or inconsistent records into related tables. For example, it would be possible to create an RSVP ticket referencing an event or student that does not exist.

3.4 Association Table Mapping

Association Table Mapping is used to represent many-to-many relationships between entities in a relational database. By introducing association tables that hold foreign keys referencing the entities it links, this pattern breaks down many-to-many relationships into two one-to-many relationships. This mapping simplifies database structure and allows efficient management of complex relationships, such as students being members of multiple clubs or RSVP tickets linked to multiple events. Below is our ADR for Foreign Key Mapper design pattern.

Title	Design the Association Key Mapper Pattern for Club manage system
Status	Accepted
Context	In database structure design, many-to-many relationships frequently arise when multiple entities relate to multiple other entities. For example, students can RSVP to multiple events, or a club can have multiple student members. Directly storing these relationships in a database without another associate table is quite difficult. To address the issue, we introduce association tables(studentRsvp, ClubMember, and ClubAdmin) that break many-to-many relationships into manageable one-to-many relationships. Additionally, This also avoids storing a

	list of data in the table. Storing list in postgres is easy, but when fetching the data and using the data as parameters, it becomes complicated if we use list to store, since the list will be stored as a string in database.
Decision	Implement association tables to efficiently manage many-to-many relationships between entities like students, events, tickets, and clubs.

Implementation Strategy	**StudentRSVP Table:**
	• **Purpose:** Links the `Student` and `RSVP_Ticket` tables to track which students RSVP to which tickets.
	• **Foreign Keys:**
	  ○ `student_id` → `Student.student_id`
	  ○ `ticket_id` → `RSVP_Ticket.ticket_id`
	
	ClubMember Table:
	• **Purpose:** Links the `Student` and `StudentClub` tables, representing the membership of students in clubs.
	• **Foreign Keys:**
	  ○ `student_id` → `Student.student_id`
	  ○ `club_id` → `StudentClub.club_id`
	
	ClubAdmin Table:
	• **Purpose:** Links the `Student` and `StudentClub` tables, representing the administrative roles students have in clubs.
	• **Foreign Keys:**
	  ○ `student_id` → `Student.student_id`
	  ○ `club_id` → `StudentClub.club_id`

Consequences	**Normalization:** Reduces redundancy and ensures that complex many-to-many relationships are represented efficiently and easily to use. **Optimize Queries:** By turning relationships into association tables, queries become more efficient, especially for data in association tables.
Alternatives Considered	We can also directly represent many-to-many relationships without designing associate tables. A direct many-to-many relationship could have been considered for simplicity in the early stages of design. However, This may lead to data redundancy. Data management will be harder, querying also faces challenges, especially as our system's scope grows. It also violates principles of database normalization, increasing risk of anomalies.
Consequences of Not Implementing	Without association tables, it would be harder to maintain consistent relationships between students, clubs, events, RSVPs, and also tickets. Data integrity also may not be easy to maintain as many-to-many relationships will be hard to represent.

3.5 Data Mapper

Data Mapper pattern provides the best trade-off between flexibility, scalability, and maintainability. As our project continues to grow, introducing the separation between data access and other layers is important, since it can improve overall code quality. Below is our ADR for Data Mapper design pattern.

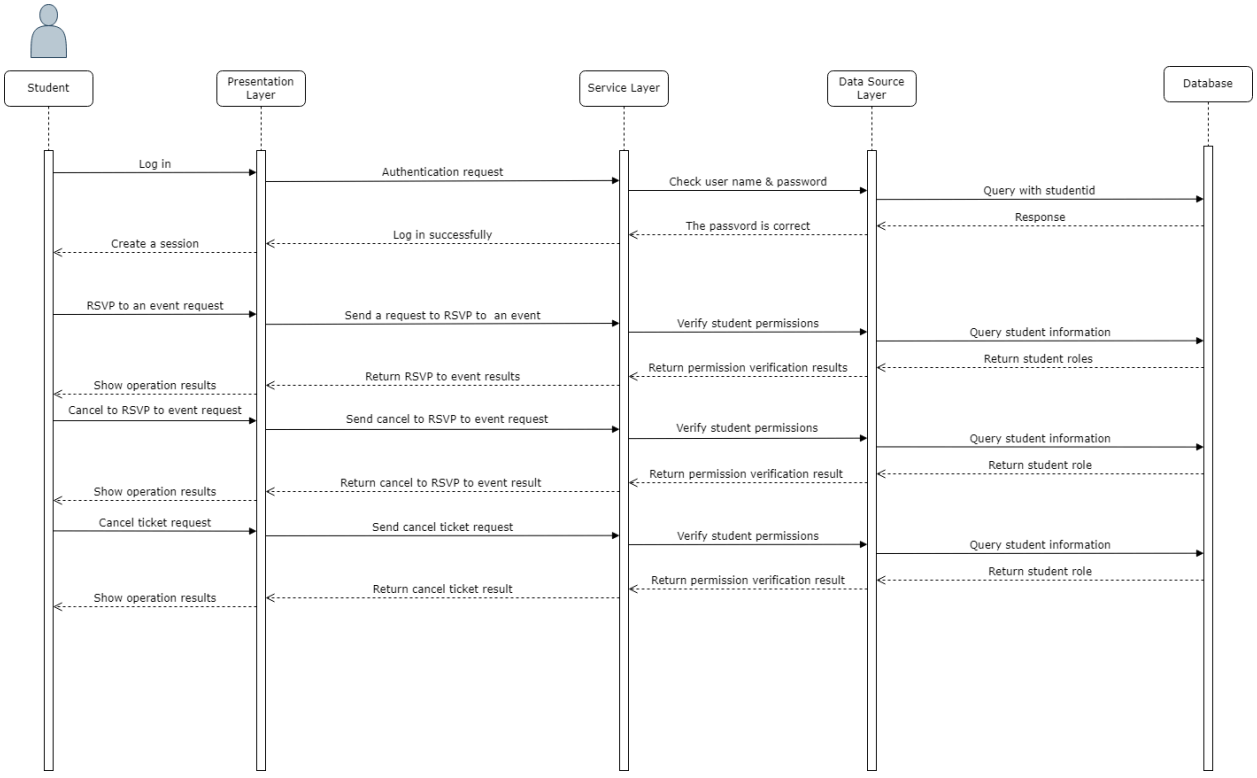
Title	Implementing the Data Mapper Design Pattern for Club manage system
Status	Accepted
Context	In our Club manage system, the business logic complexity and database interactions increased following the development phase. Currently, the data access layer is coupled with domain logic(like the test resource.java code example from workshop), where SQL queries and database interactions are embedded within domain model objects. This approach leads to higher costs, low scalability, and in future may lead to difficulties in testing and clear separation between logic and data access. As our system code scale expands, we notice the need for cleaner separation between domain models and data persistence code. Therefore, we need a design pattern that allows us to decouple business logic from database interactions, and therefore improve the testability of our code. To achieve the separation, we learned the idea about the MVC architecture. This widely used model showed that we need to adopt the Data Mapper pattern, to ensure the Model focuses solely on logic, while the Data Access Layer (DAL) is responsible for managing database operations.
Decision	We will implement the **Data Mapper** design pattern for our system's data access layer to ensure a clean separation of database query and other logic.
Implementation Strategy	**Implementation Strategy:** 1. **Data Access Layer(DAL):**Introduce Data Access Layer that includes mapper classes such as `EventMapper`, `RSVPMapper`, `StudentMapper`, etc. These mapper classes will be used for managing the interactions between the domain model objects (e.g., `Event`, `Student`, `Club`) and the database. 2. **Model Layer:**The Model layer, in accordance with the MVC pattern, will contain the domain logic. By adopting the Data Mapper pattern, the domain models (e.g., `Event`, `Student`, `Club`) will no longer handle database operations, which will be delegated to the DAL. This separation ensures that the Model solely represents the business logic of the application. 3. **Controller Layer:**Controllers will now interact with the **Service Layer**, which in turn uses the mapper classes from the DAL. This ensures that controllers focus on handling HTTP requests, delegating business logic to the service layer, and responding appropriately, while the service layer manages all interactions with the database through the mappers.

	4. **Service Layer:**The service layer (e.g., `EventService`, `RSVPService`) will contain the business logic and communicate with the mappers to retrieve or store data in the database. This ensures that the **Controller** and **View** components do not need to concern themselves with the details of data persistence. 5. **Lazy Loading and Unit of Work Integration:**The Data Mapper pattern will work in conjunction with other design patterns such as **Lazy Loading** and **Unit of Work** to ensure efficient database access and transaction management, without tightly coupling these concerns with the business logic.
Consequences	**Positive:** 1. **Aglin with MVC:** This implementation brings our project more in line with the **MVC** pattern by keeping the **Model** focused on business logic and delegating data access concerns to the DAL, thereby promoting clear separation of concerns. 2. **Separation of Operation:** The Data Mapper pattern ensures that business logic is not intertwined with data access code, making the application more modular, maintainable, and easier to test. 3. **Testability:** By decoupling business logic from data access logic, it becomes easier to mock the data layer for unit tests, improving overall test coverage and reliability. **Negative:** 1. **Complexity:** The Data Mapper pattern introduces more layers to the application, which increases the overall code complexity. Developers need to manage separate mapper classes for each entity and ensure that the service layer communicates correctly with the mappers.
Alternatives Considered	**Active Record Pattern:** Active Record pattern allows each domain object to directly manage its own database interactions. This pattern is simpler and faster to implement for small applications but lacks scalability as the application grows. **Positive:** ● **Lower Complexity:** With the Active Record pattern, the code is simpler and requires fewer classes to implement, leading to faster development in the short term. ● **Direct Interaction:** Each domain model directly handles its database operations, reducing the need for separate layers.

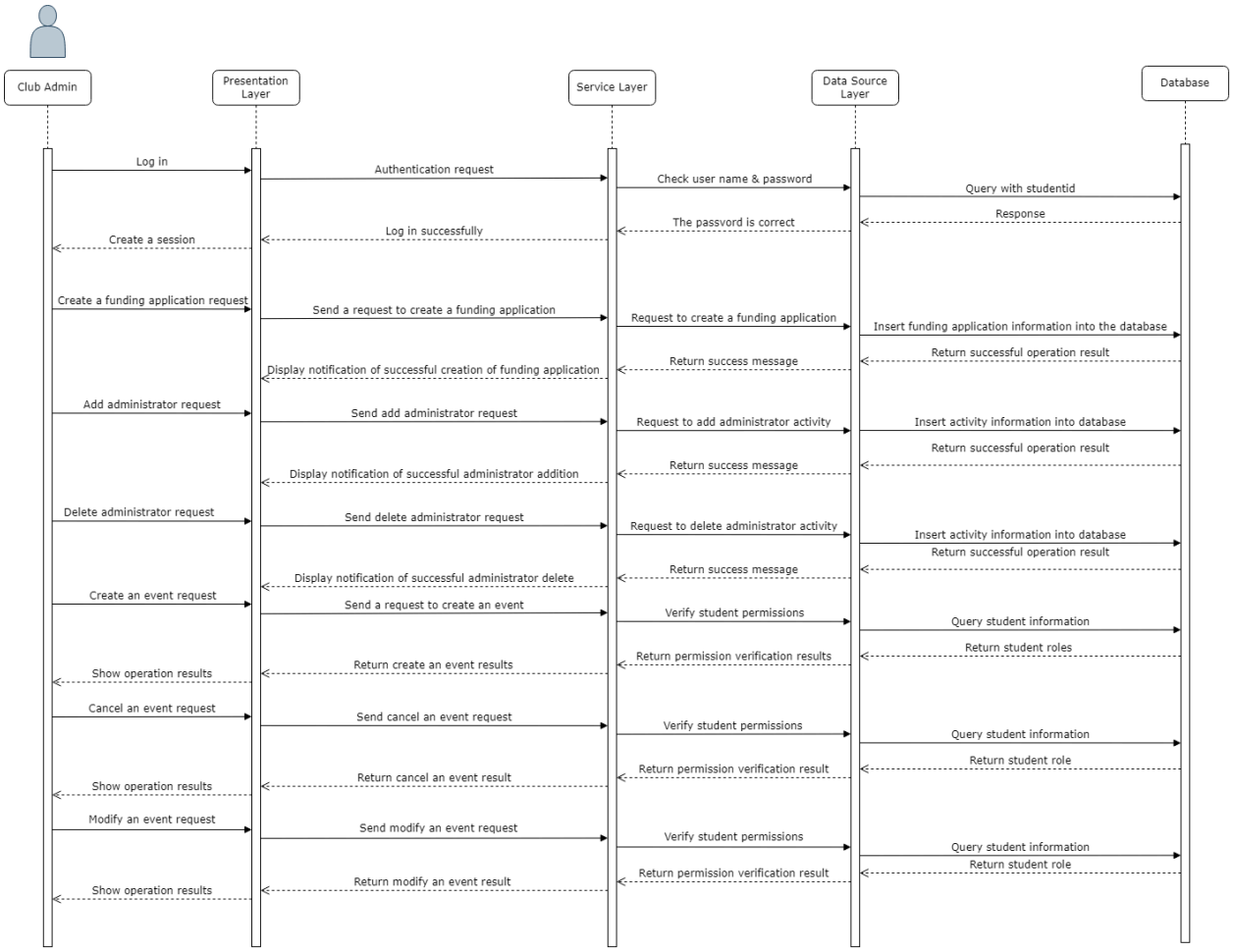
	Negative: ● **Tightly Coupled Logic:** The business logic is tightly coupled with database queries, making it difficult to scale, test, or maintain in the long run. ● **Reduced Flexibility:** As the application grows, updating database schema or adding new business logic becomes more cumbersome since business rules and data access are mixed. ● **Scalability Concerns:** It's not well-suited for large, scalable systems because business logic becomes embedded in data access code, making the application harder to extend.
Consequences of Not Implementing	Without the Data Mapper pattern, the system will continue to have tightly coupled business logic and data access, making future scalability and maintainability a challenge. As the system grows, the absence of a clear separation will lead to higher technical debt, reduced flexibility, and difficulty in testing and modifying the codebase.

4. behavioral design patterns

4.1 Sequence Diagram



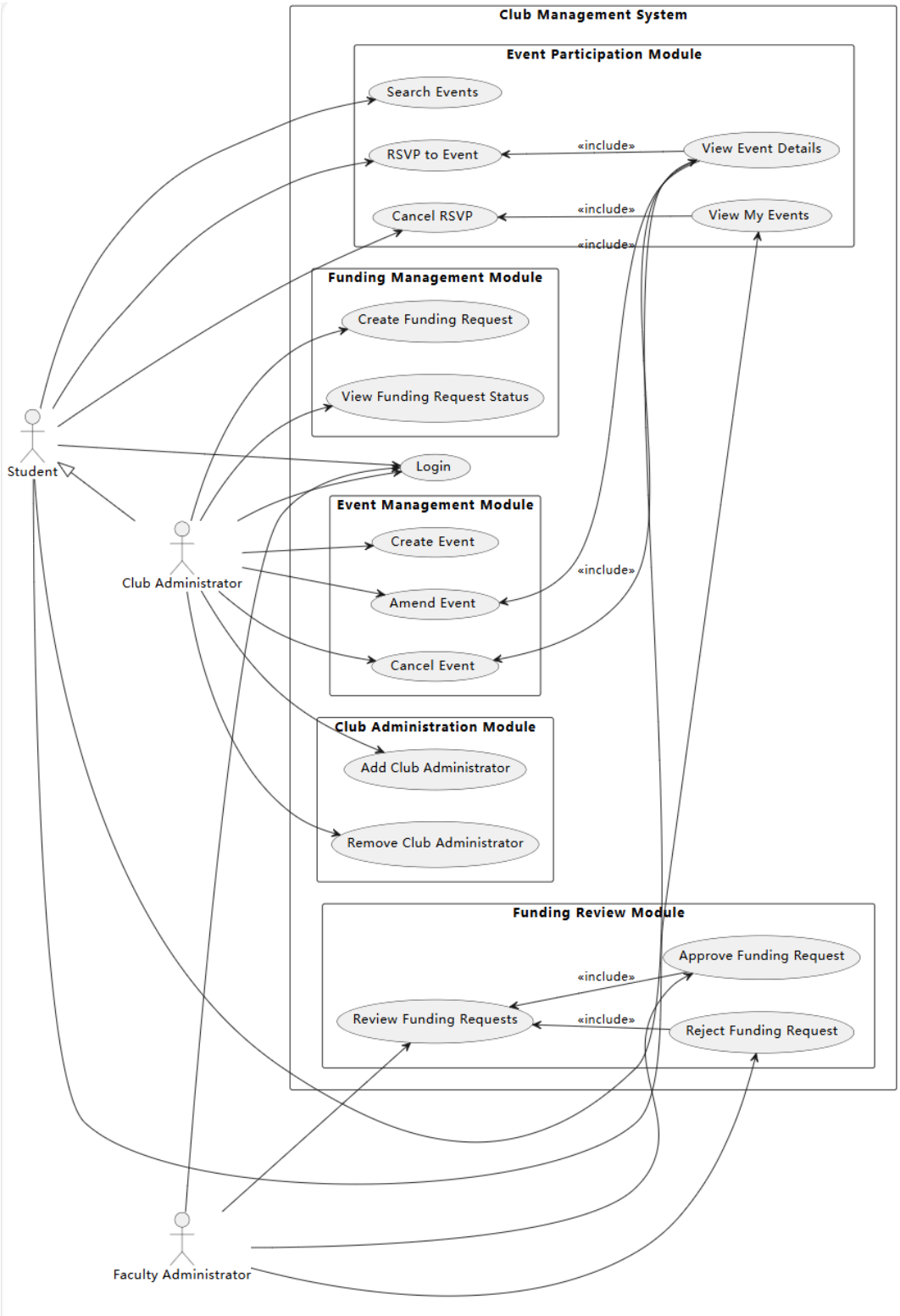
This sequence diagram shows the operation process of **students** in the student club management system. First, students enter the system through a login request. The service layer verifies the username and password, queries the student information in the database through the data layer, and confirms the login. After successful login, students can initiate requests to rsvp for events, cancel rsvp and cancel ticket operations. Each request will first verify the student's permissions at the service layer. The service layer obtains the student's role information from the data layer, confirms the student's authentication status, and then performs the corresponding operation and returns the operation result. The entire process ensures that only students with successful authentication can operate this system.



This sequence diagram describes the operation process of student club **administrators** in the club management system. First, the administrator logs in to the system, and the service layer verifies the username and password and creates a session. Administrators can perform operations such as creating funding applications, adding or deleting administrators, creating,

canceling, and modifying events. Each request is processed by the service layer, which passes the relevant request to the data source layer, and the database completes the data insertion or update operation and finally returns the operation result. The system verifies the administrator's permissions to ensure that only users with appropriate permissions can perform the corresponding operations, and promptly feedbacks the operation results to the administrator.

4.2 Use Case Diagram



This use case diagram shows the core functional modules and user interaction methods of the student club management system. Students can search for activities, RSVP to activities, cancel

RSVP and view event details through the event participation module. In addition to performing student functions, club administrators can also create, modify and cancel activities in the event management module, add or remove other administrators through the club management module, and submit and view the status of funding applications through the funding management module. The college administrator is responsible for reviewing funding applications and approving or rejecting funding applications through the funding review module. All users must first enter the system through the login function, which ensures user permission control and operation security.

4.3 Lazy load

The Lazy Load pattern provides an efficient way to optimize performance for large datasets by deferring data fetching until necessary. It ensures that the application responds to user operation soon as the data grows, especially in terms of managing complex relationships between entities like events and RSVPs. Below is our ADR for Lazy Load design pattern.

Title	Implementing the Lazy Load Design Pattern for Club manage system
Status	Accepted
Context	The Club manage system manage growing amount of data, and some entities in our application (such as events, RSVPs, and associated details) are becoming increasingly large. When loading data from the database, we are often retrieving more information than necessary, leading to performance bottlenecks and excessive memory consumption. For example, retrieving all associated RSVPs when displaying a list of events, even though the RSVP data might not be needed immediately, has led to inefficient use of resources. To address this, we need a design pattern that delays the loading of large datasets until they are actually needed, optimizing performance and reducing unnecessary memory overhead.
Decision	We will implement the **Lazy Load** design pattern for our project' s entities that involve large datasets. Lazy loading will be applied to entities such as events and their associated RSVPs, ensuring that data is only fetched when it is explicitly required.
Implementation Strategy	**Lazy Loading with Pagination:** ● Introduce lazy loading with pagination in the mappers (e.g., `EventMapper`, `RSVPMapper`). This ensures that the application only retrieves a specific number of rows when required, instead of fetching entire datasets. ● Lazy load RSVPs or related data only when requested by the client.

	Proxy Objects: ● For entities like events and RSVPs, implement proxy objects that delay fetching related entities until explicitly accessed. For example, when accessing an event, the associated RSVPs will not be retrieved immediately. The data will only be loaded when the RSVPs are explicitly requested in the business logic. **Modification to Data Access Layer (DAL):** ● We modified the Data Access Layer to support lazy load. Mappers will return proxy objects instead of fully populated entities, and when a specific method is invoked (e.g., `getRSVPs()` for an event), the actual data will be fetched from the database at that point. **Controller Support for Lazy Loading:** ● In the controller layer (e.g., `EventController`), expose lazy-loaded endpoints that can fetch additional data (e.g., RSVPs) on-demand rather than as part of the initial response. Pagination parameters (e.g., `page`, `pageSize`) will be used to control how much data is retrieved per request.
Consequences	**Positive:** 1. **Improved Performance:** By only loading necessary data when needed, the Lazy Load pattern reduces the memory footprint and improves the application's performance, especially for large datasets like events and RSVPs. 2. **Optimized Resource Usage:** Reduces database load by not fetching unnecessary data, ensuring that queries only retrieve what the user is currently interacting with. 3. **Scalability:** The application will scale better as datasets grow because lazy loading ensures that only the data required at a given moment is retrieved. **Negative:** 1. **Increased Code Complexity:** Implementing Lazy Load introduces additional complexity, especially around proxy objects and deferred data fetching. Developers will need to ensure that data is fetched correctly when required. 2. **Potential Performance Overhead in Some Cases:** Although lazy loading optimizes initial loading, there may be scenarios where multiple database trips are required, which could affect performance in high-frequency operations if not managed carefully.

Alternatives Considered	**1. Eager Loading:** Eager loading would involve fetching all related data upfront. While this reduces the complexity of managing deferred loading, it can result in performance bottlenecks and wasted resources for large datasets when only a subset of the data is actually needed. **Positive:** Eager load is simpler to implement, no need to manage proxy objects or deferred data fetching. It also reduces the number of database queries by fetching all data in one go. **Negative:** Performance may be impacted, as unnecessary data is loaded even when it is not needed. It may cause inefficient memory and resource usage, especially for large datasets like events and RSVPs. **2. Batch Loading:** Batch loading involves loading related data in fixed-size batches. This approach can optimize performance in certain cases but lacks the on-demand flexibility provided by lazy loading. **Positive:** Batch load may reduce performance impact by fetching related data in chunks, rather than all at once. **Negative:** Batch load seems to be a resource saving but actually may still fetch more data than necessary in many cases. Not as flexible as lazy loading, as it forces the application to load predefined data chunks regardless of whether they are needed.
Consequences of Not Implementing	Without the Lazy Load design pattern, the application will continue to suffer from performance issues related to loading large datasets unnecessarily. As the data grows, the inefficiency of eager loading will become more pronounced, leading to slower response times, higher memory usage, and a poor user experience.

4.4 Unit of Work

The Unit of Work pattern ensures that all data changes are made within a transactional boundary. This approach allows us to manage complex data operations involving multiple entities in a reliable and maintainable way. The scalability and consistency benefits far outweigh the complexity it adds to the codebase. Below is our ADR for Unit of Work design pattern.

Title	Implementing the Unit of Work Design Pattern for Club manage system
Status	Accepted
Context	The Club management system requires a mechanism to ensure consistent data handling operations, particularly during the execution of multiple database operations as part of a single business transaction. Our current approach lacks

	transaction management, which often results in inconsistencies, especially when there are failures in the middle of a sequence of operations. For example, in scenarios such as updating multiple related records (e.g., during event creation, RSVP, and related ticket operations), if one operation fails, the preceding successful operations are not rolled back. This leads to data integrity issues and complex error-handling logic in our code. To address this, we need a pattern that allows us to manage and commit multiple operations as a single unit, ensuring either all operations succeed or none are applied.
Decision	We will implement the **Unit of Work** design pattern to manage database transactions and ensure data consistency in our system. The Unit of Work will track changes to objects during a transaction and handle committing or rolling back operations based on success or failure.
Implementation Strategy	**Unit of Work Layer:** ● Introduce a `UnitOfWork` class responsible for managing the transaction lifecycle, including starting, committing, and rolling back transactions. ● The `UnitOfWork` class will be responsible for coordinating database operations involving multiple repositories (e.g., EventMapper, RSVPMapper) and ensuring data consistency. **Data Access Layer (DAL):** ● All data access classes (e.g., `EventMapper`, `RSVPMapper`) will interact with the `UnitOfWork` to handle operations such as inserts, updates, and deletes. ● Mappers will perform CRUD operations within a transactional scope provided by the `UnitOfWork`. **Error Handling:** ● In case of an error during any operation in the transaction, the `UnitOfWork` will ensure all preceding operations are rolled back to maintain data integrity. ● The `UnitOfWork` will provide methods to track multiple changes and ensure a clean rollback upon failure.
Consequences	**Positive:** 1. **Consistency:** Ensures data consistency by grouping multiple operations into a single unit of work that either completely succeeds or fails. 2. **Simplified Transaction Management:** By using a centralized

	`UnitOfWork` class, we simplify the management of complex transactions that span multiple mappers and database tables. 3. **Scalability:** As the system grows, the Unit of Work pattern will handle increasing complexity with ease, allowing us to manage complex transaction workflows across several related entities. **Negative:** 1. **Complexity:** Implementing the Unit of Work pattern adds an extra layer of abstraction, which could increase the complexity of the codebase. Developers need to understand and maintain the lifecycle of transactions through the `UnitOfWork`. 2. **Performance Overhead:** There may be slight performance overhead, as the pattern tracks changes to multiple objects and their states.
Alternatives Considered	**Single Transaction Per Operation (Without UoW):** Each operation could be managed independently with its own transaction. However, this approach would fail in situations where operations depend on one another (e.g., inserting event data and related RSVP records). **Positive:** This function can simplify implementation without introducing an extra layer like Unit of Work. **Negative:** This approach may be inadequate for some scenarios that require multiple operations to succeed as a unit. It also does not address the issue of maintaining consistency across multiple interdependent operations.
Consequences of Not Implementing	If we do not implement the Unit of Work pattern, our system will likely suffer from data consistency issues during complex operations. As the system scales, handling these issues manually would become increasingly difficult and error-prone, leading to potential data corruption and longer development time for future features.

4.5 Authentication and Authorization

The Authentication and Authorization process secures the Club management system by ensuring that users access only permitted resources based on their roles. We use JWT for stateless authentication and Role-Based Access Control for managing permissions. This approach provides a secure and scalable solution, allowing only authorized users to perform privileged actions, with the added complexity justified by improved security and flexibility.

Title	Implementing the Authentication and Authorization Design Pattern for Club manage system
Status	Accepted

Context	In order to ensure the security and functional integrity of the student club management system, we need to implement the authentication and authorization design pattern. The system needs to ensure that only authenticated users can access the relevant functions of the system and the scope of operations that users can perform through the authentication and authorization design pattern. This system involves the following types of users: ordinary students, club administrators, and college administrators. All these users must be authenticated and assigned corresponding permissions according to their roles.
Decision	We chose **JWT**-based stateless authentication and role-based access control. The system authenticates users through JWT tokens and assigns permissions based on user roles. When a user logs in, the system generates a JWT token that contains the user's identity information and their role in the community (such as ordinary member, administrator, or college administrator). All subsequent API requests need to carry this token for authentication and authorization.
Implementation Strategy	**User Authentication:** ● Users log in by submitting the staffId or email and password through LoginController. The system uses UserService to verify the user's identity. If an email is provided, the system treats it as a student and verifies the email and password; if a staffId is provided, the system verifies it as a faculty administrator. **Password Processing:** ● The system uses BCryptPasswordEncoder to hash and store user passwords. When logging in, the system hashes the password entered by the user and compares it with the hashed password in the database. **Token Generation:** ● After verification, the system uses JwtService to generate a JWT token. The token contains the user's studentId and role information (such as ClubAdmin, ClubMember), and is used for subsequent authentication and authorization. **Token Verification:** ● Each time a request is made, the client needs to provide a JWT token in the request header. The system verifies the validity of the token through JwtTokenUtil and parses the studentId to obtain the user's permission information. **Authorization control:** ● The system obtains the user's role in the club (such as ClubAdmin or ClubMember) by parsing the studentId in the JWT token, and restricts the user's operation permissions based on the role. **Token expiration and refresh mechanism:** ● The JWT token has an expiration date, and after expiration, the user needs to log in again to obtain a new token. If the token is invalid or expired, the system will return an HTTP 401 Unauthorized response.
Consequences	**Positive:** 1. Security: Authentication through JWT ensures that only authenticated users can access the system.

	2. Flexibility: Use role-based access control to allow different users to perform specific operations based on their roles. 3. Scalability: JWT tokens can carry rich user information to support future system expansion requirements. 4. Statelessness: JWT is stateless and suitable for distributed system architecture. There is no need to maintain user session status on the server side. **Negative:** 1. Token management: It is necessary to handle the expiration and refresh issues of JWT to ensure the validity of the token when the user uses it for a long time. 2. Security risks: If the token is not properly protected, it may lead to security issues, such as tokens being stolen and abused.
Alternatives Considered	**API Key-based Authentication:** We considered using an authentication method based on API Key, where users provide a static API Key for authentication each time they make a request. While this approach is simple to implement and widely used in many systems, it has certain limitations, particularly in terms of security and flexibility. **Positive:** ● Simplicity: API Key-based authentication is easy to implement and does not require complex server-side session management. **Negative:** ● Security Risks: API Keys can be easily stolen if they are not properly protected. For instance, if the key is exposed in client-side code or intercepted over insecure connections, attackers can gain unauthorized access to the system. ● No Expiration or Revocation: Typically, API Keys do not have built-in expiration or revocation mechanisms, meaning they can be valid indefinitely unless explicitly managed, leading to potential long-term security vulnerabilities.
Consequences of Not Implementing	If a JWT-based authentication and authorization system is not implemented, the entire system will lack the necessary security controls, allowing unverified users to access sensitive data. In addition, it will not be possible to effectively distinguish between ordinary users and administrators, and the system will lack a flexible permission management mechanism, which may cause security risks and affect the functionality of the system.