SWEN90007-Part2 Report

1. Introduction

Purpose of the report

This report presents the design and implementation details of an application aimed at managing student events and student club activities. Each implemented pattern has its own ADR in the report. We also provide diagrams such as sequence diagrams and entity relationship diagrams to illustrate the architecture and functionality of each design pattern implemented in the application.

The application allows students to browse upcoming events hosted by student clubs, RSVP to those events, and manage their tickets. Furthermore, students with administrative roles within their clubs are given additional functionalities, such as the ability to create, modify, and cancel events, manage club administrators, and handle funding applications. **Specifically, we do not provide any functionality related to faculty administrators (as required in Part3).**

According to the requirements, the following user stories were implemented in our current application:

- As a Student, I want to search for upcoming events, so that I can find events hosted by Student Clubs and RSVP to them.
- As a Student, I want to RSVP to an event, so that I can attend events hosted by Student Clubs.

- As a Student, I want to create an event for a Student Club I administer, so that I can organise activities and gatherings.
- As a Student, I want to cancel an event for a Student Club I administer, so that I can remove events as needed.
- As an administrator of a Student Club, I want to add other Students as administrators, so that they can create, amend or cancel events as needed.
- As an administrator of a Student Club, I want to remove other Students as administrators, so that they can no longer create, amend or cancel events as needed.
- As an administrator of a Student Club, I want to create an application for funding, so that the Student Club can hold event/s.
- As a Student, I want to cancel one or more RSVPs to an event, so that it shows the correct number of students attending.
- As an administrative member of a Student Club, I want to modify an event within my Student Club, so that events contain accurate information.

2. Class Diagram of the Application

Key Relationships:

1. Inheritance Relationships:
    a. Person ← Student
    b. Person ← FacultyAdministrator
    c. DomainObject ← Club, Event, RSVP, Ticket, FundingApplication, Venue: All of these entities inherit from the abstract `DomainObject` class, which provides a shared `id` field to uniquely identify each object in the system.
2. Association Relationships:
    a. Club - Student (Many-to-Many): students can join multiple clubs, and each club can have multiple students.
    b. Student - RSVP (Many-to-Many): a student can submit multiple RSVPs and a RSVP can have information of multiple participants.
    c. Event → Venue(One-to-One): Each event is hosted at a specific venue.
    d. FacultyAdministrator → FundingApplication(One-to-Many): A faculty administrator reviews multiple funding applications, but each application is reviewed by only one administrator.
3. Composition Relationships:
    a. Club → FundingApplication (One-to-Many): a funding application cannot exist without the club it is related to.
    b. Club → Event (One-to-Many): each club can host multiple events. Each event is linked to one specific club.
    c. Student → Ticket(One-to-One): each ticket is tied to a specific student and cannot exist independently of the student.
4. Dependency Relationships:

a. RSVP → Event: an RSVP depends on an event. If an event is deleted, its RSVPs are also removed.
b. Ticket → RSVP (One-to-Many): a ticket depends on an RSVP as tickets are issued after students RSVP for events.

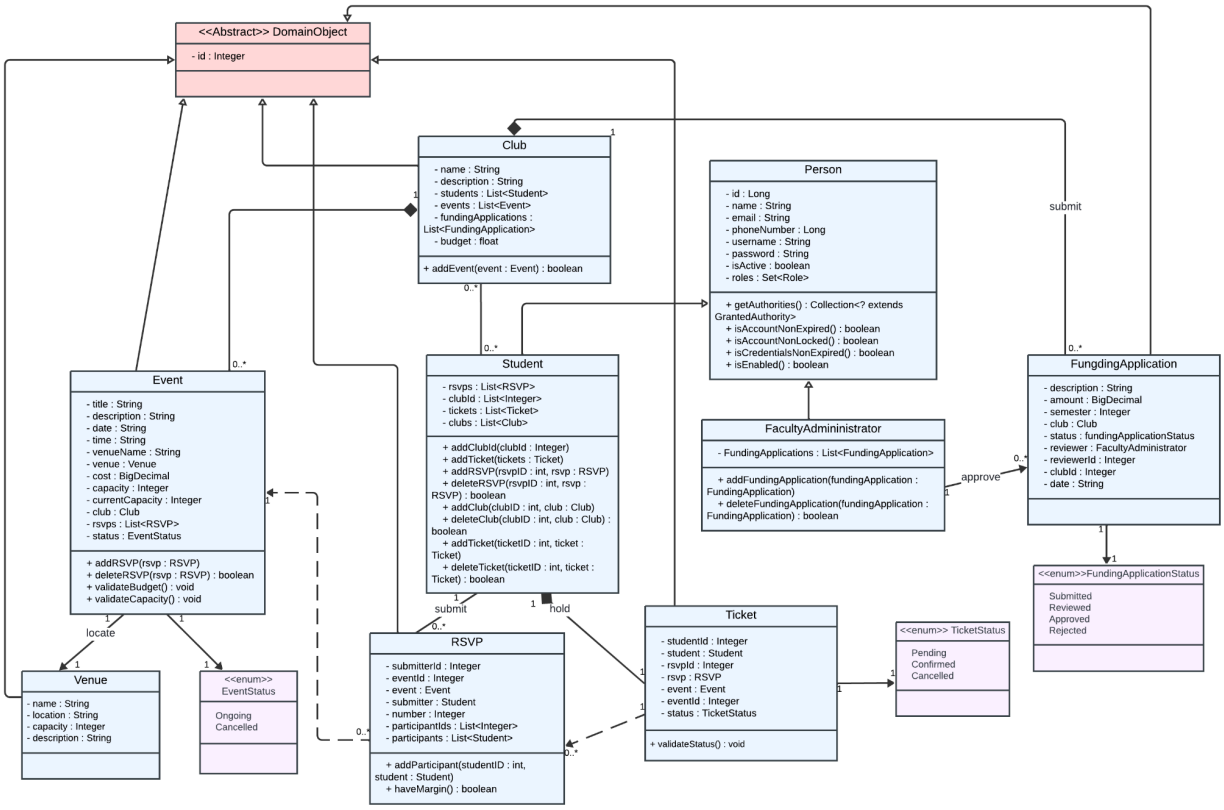


Figure 1. Class Diagram

The entities in class diagram are in the model layer of the Figure 2:

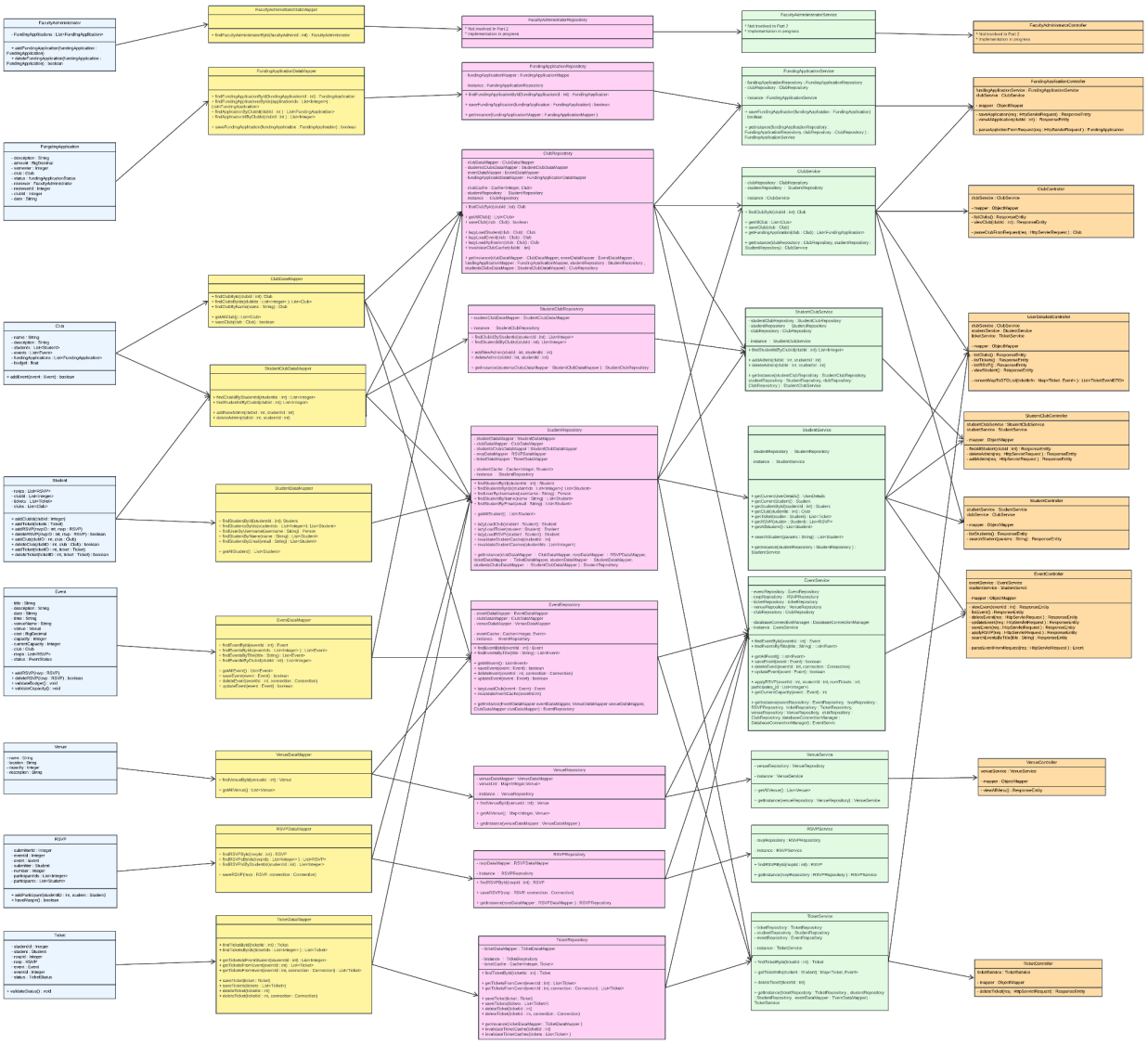


Figure 2. Architecture Diagram

The Figure 2 can also be accessed via the link:
https://lucid.app/publicSegments/view/39313f04-8f13-47ef-bdc0-c6ff3e4095ba/image.png

1. Domain Model (Model Layer) : represents the business entities and logic (state and behaviour). The data mapper interacts with the domain model to access business data.
2. Data Mapper Layer: transforms data between objects and database schemas. The repository uses it to map data from the domain model to the database entities.
3. Repository Layer: provides an abstraction over the persistence mechanism, interacting directly with the data mapper to retrieve or persist data.
4. Service Layer: handles business logic and rules. It coordinates the flow between the controller, repository, and other services.The service layer interacts with the repository layer to perform data operations.

5. Controller Layer: acts as the entry point for handling user requests and responses.The controller interacts with the service layer to delegate business logic processing.

3. Implemented Patterns

3.1 Domain Model

3.1.1 Context

Our system includes creating and managing clubs and events, RSVP (reservation) functionality, handling funding applications, and managing event capacity, among other complex business logic. It needs to support different types of user roles (students, club administrators, faculty administrators) performing various operations, where each user role's behavior and interactions involve multiple layers of logic and rules.

The domain model pattern encapsulates complex business logic directly within domain objects, simplifying the management of business processes while supporting state management, such as the creation, modification, and capacity validation of club events. It also enhances the system's maintainability and scalability, allowing business rules to be modified or extended within domain objects without impacting other parts of the system, thereby improving development efficiency and system flexibility.

3.1.2 Decision

Based on the context, we decided to adopt the Domain Model Pattern. Particularly, we chose an **anaemic domain model** instead of POJO[1], which was mentioned by Martin Fowler in *Patterns of enterprise application architecture*[2]

That is, our domain objects contain the definition of object properties and getter/setter methods for manipulating objects and **simple domain validation logic**. And, we put complex business logic and database persistence-dependent logic in the service layer.

3.1.3 Implementation Strategy

We implemented the domain object and service layers, respectively, according to the domain model pattern. The overview architecture diagram as shown in Figure 3:

---

[1] Medium. (2020, August 27). *Creating coding excellence with domain-driven design*. The Startup. https://medium.com/swlh/creating-coding-excellence-with-domain-driven-design-88f73d2232c3#:~:text=The%20POJO%2C%20or%20blood%20loss,manager%20(the%20father's%20conscience).

[2] *M. Fowler. Patterns of enterprise application architecture. Addison-Wesley Professional, 2003.*

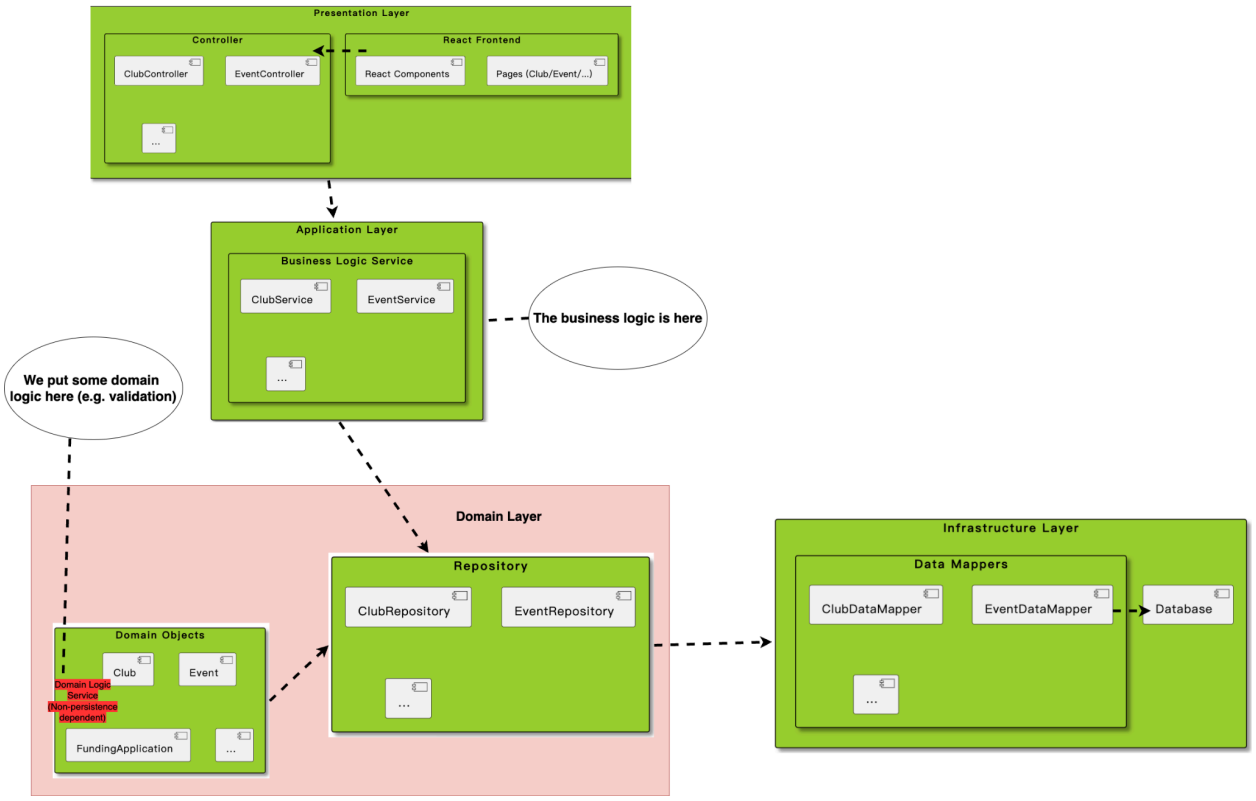


Figure 3. The schematic implementation diagram in our domain model pattern

3.1.3.1 Domain Object

A domain object contains only logic and validation related to its own state. These objects(e.g.,Club, Event, etc.) manage their properties primarily through getter and setter methods and implement some simple validation related to business rules. The achieved domain objects(except the Enum class) are shown below in Figure 4:

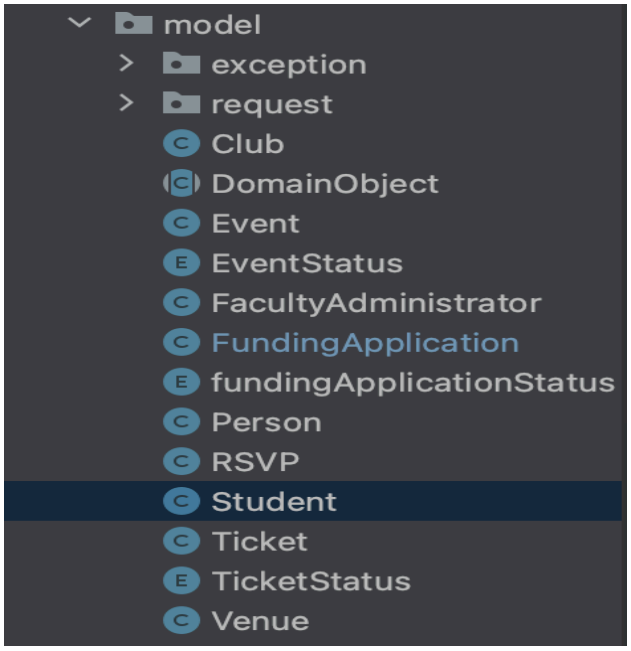


Figure4. The achieved domain object

```
public class Event extends DomainObject {
    5 个用法
    private String title;
    5 个用法
    private String description;
    6 个用法
    private String date;
    6 个用法
    private String time;
    4 个用法
    private String venueName;
    3 个用法
    private Integer venueId;
    6 个用法
    private BigDecimal cost;
    5 个用法
    private Integer clubId;
    3 个用法
    private Club club;
```

Figure5. The attributes in Club

```
Crystallen1
public EventStatus getStatus() { return status; }

Crystallen1
public void setStatus(EventStatus status) { this.status = status; }

8 个用法  Crystallen1
public String getTitle() { return title; }

0 个用法  Crystallen1
public void setTitle(String title) { this.title = title; }

Crystallen1
public String getDescription() { return description; }

Crystallen1
public void setDescription(String description) { this.description = description; }
```

Figure 6. The getter and setter method in Club

For instance, the Event class has constructor methods, getter and setter methods, which has shown in Figure 5 and Figure 6. Similarly, other domain objects have the same things.

However, we mentioned before that we have achieved an anaemic domain model. Let us explain it further. Think of it like a person who has certain attributes, such as name, gender, and age, as well as some abilities, like walking, eating, and falling in love. That is, how much behaviour do we include, in fact, means whether our domain model is

anaemic or rich, and if it doesn't include anything, in some materials, it's called a blood loss model. In our implementation, we define these abilities as some simple business validation logic, and we do not include business logic that depends on the persistence layer. For example, as shown in the below figure 7, these checks are performed directly within the domain object. Meanwhile, in the service layer, these actions are simply encapsulated to ensure that transaction management and persistence of these business logics are handled. In addition, you might find some unused methods in our domain objects. These are for features that are not currently implemented in the system but are reserved for future use.

By doing this, we have implemented an anaemic model. In fact, if you're familiar with the anaemic model, our domain model isn't the simplest version (which only contains attributes and getter/setter methods). Instead, it's closer to the rich model and aligns more with Fowler's true definition of a domain model.

```
public void validateBudget() {
    if (this.cost.compareTo(BigDecimal.valueOf(club.getBudget())) > 0) {
        throw new RuntimeException("Budget not enough");
    }
}

1 个用法  Crystallen1
public void validateCapacity() {
    if (this.capacity > venue.getCapacity()) {
        throw new RuntimeException("Venue capacity not enough");
    }
}
```

Figure 7. The validation logic we achieved in domain object

3.1.3.2 Service Layer

In our implementation, the service layer handles complex business logic and persistence operations. For instance, the ClubService class manages club persistence and interacts with the repository layer, coordinating multiple domain objects when

needed(as shown in Figure 8).

```java
public void deleteEvent(List<Integer> eventsId)throws Exception{
    EventDeleteUoW eventDeleteUoW = new EventDeleteUoW(eventRepository,ticketRepository);
    for (Integer eventId : eventsId){
        eventDeleteUoW.addDeleteEvents(eventId);
    }
    System.out.println(eventDeleteUoW.toString());
    eventDeleteUoW.commit();
}

4 个用法  Crystallen1
public Integer getCurrentCapacity(Event event)throws Exception{
    List<Ticket> tickets = ticketRepository.getTicketsFromEvent(event.getId());
    int count=0;
    for(Ticket ticket:tickets){
        if(ticket.getStatus().equals(TicketStatus.Issued))count++;
    }
    return event.getCapacity()-count;
}
```

Figure 8. The business logic we achieved in service layer

3.1.4 Consequences

Here, we talk about the consequences of our implementation, not the general consequences about the domain model pattern.

Positive

- **Clear separation of responsibilities**: Domain objects handle simple validation and state management, while the service layer handles complex business logic and persistence operations, ensuring clean separation of concerns.
- **Scalability**: Domain objects are easy to extend through self-contained validation, and the service layer handles complex business logic and interactions between domain objects.
- **Testability**: Domain objects can be unit tested independently of the database, while the service layer can be tested using mock objects for data access.

Negative

- It does not handle very complex logic and scenarios well.
- The Service layer is still a little bit thick.

3.2 Data Mapper

3.2.1 Context

As mentioned in Section 3.1, our system involves several complex domain entities, such as Students, Clubs, Events, and RSVPs, which require frequent data exchange with the persistence layer (database). Since the business logic of these domain objects needs to be separated from the persistence logic, we have chosen to implement the Data Mapper pattern to solve this problem.

This pattern allows domain objects and database operations to exist independently, ensuring that business logic and persistence logic remain separated, thus increasing the flexibility and maintainability of the system.

3.2.2 Decision

We have chosen to create a separate Data Mapper class for each domain object (e.g., Student, Club, Event). These mapper classes are responsible for translating data between the in-memory objects and the database. The mappers will query the database and convert the results into domain objects or convert domain objects into the format needed for database storage.

3.2.3 Implementation Strategy

There are 9 Mapper classes in our system as shown in Figure 9:

- ClubDataMapper
- EventDataMapper
- FacultyAdministratorMapper
- FundingApplicationMapper
- RSVPDataMapper
- StudentClubDataMapper
- StudentDataMapper
- TicketDataMapper
- VenueDataMapper

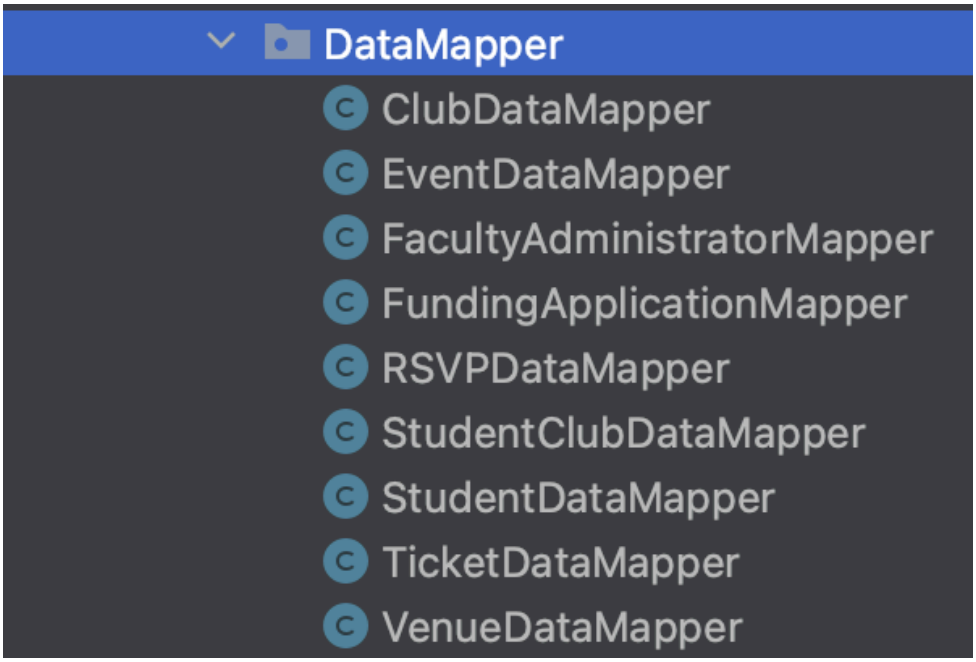


Figure 9. The nine achieved mapper classes in our application

In our system, each DataMapper class acts as an intermediary between domain objects and the database, ensuring that domain objects remain independent of the database schema. For each domain entity (e.g., Club, Event, RSVP, Student, Ticket, Venue), it remains unaware of the database schema or persistence mechanisms. Specifically, we created a corresponding DataMapper class that is responsible for translating database rows into domain objects and vice versa.

For example, the ClubDataMapper class handles converting records from the clubs table into Club objects and persists Club objects back to the database as rows. Figure 10 shows a simple class diagram that uses a data mapper pattern for Club in our implementation.

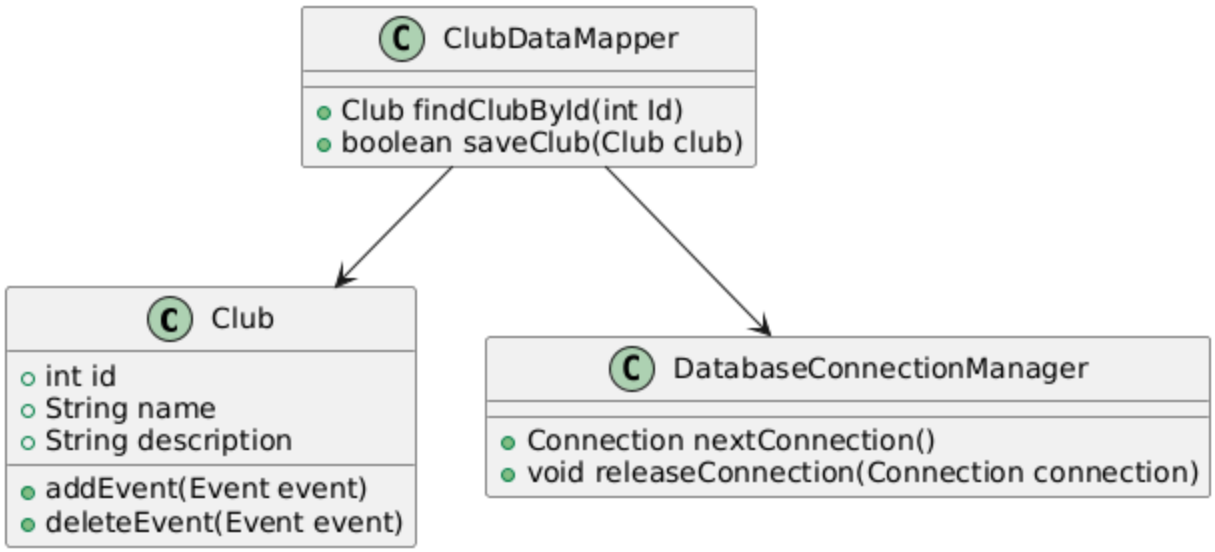


Figure 10. A class diagram that use data mapper pattern for Club in our implementation

Figure 11 shows an interaction diagram outlining how this would happen to an update.

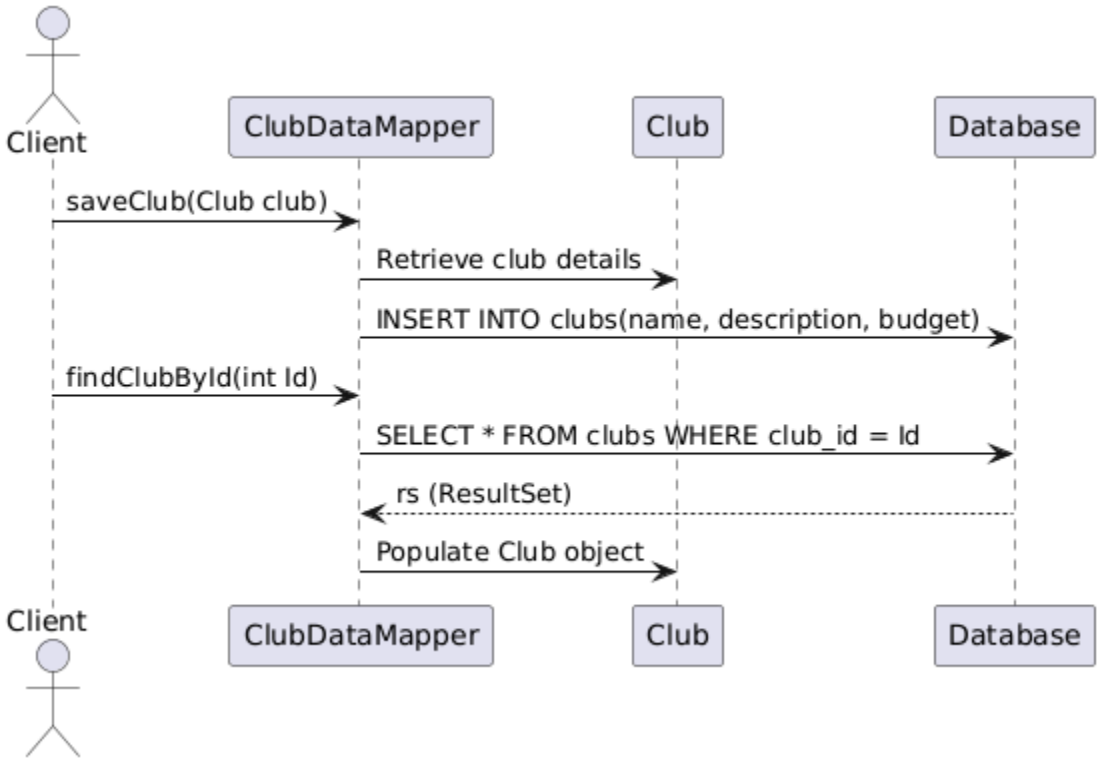


Figure 11. An interaction diagram that uses the data mapper pattern for Club.

Interaction Diagram Explanation:

Interactions as shown in Figure 11, note that we only consider datamapper layer here, i.e, ignore the repository/service layer:

1. Saving a Club (saveClub)

- Client -> ClubDataMapper: The client initiates a request to save a new club (saveClub(Club club)).
- ClubDataMapper -> Club: The ClubDataMapper accesses the Club object to retrieve the necessary club details (such as name and description).
- ClubDataMapper -> Database: The ClubDataMapper then interacts with the database to insert these club details into the clubs table via an INSERT query.

2. Finding a Club by ID (findClubById)

- Client -> ClubDataMapper: The client sends a request to find a club by its ID (findClubById(int Id)).
- ClubDataMapper -> Database: The ClubDataMapper sends a query to the database to retrieve the club's details using the provided ID (SELECT * FROM clubs WHERE club_id = ?).
- ClubDataMapper -> Club: After getting the data, the ClubDataMapper populates the Club object with the retrieved information (such as the club's name and description).

The implementation details are as follows:

A . **Retrieving data from the database:** Each DataMapper class contains methods such as findById, findByName, or findByClubId, which execute SQL queries and map the result set to domain objects. Examples include:

- ClubDataMapper.findClubById(int id) retrieves a Club object based on club_id (Refer to the below Figure).

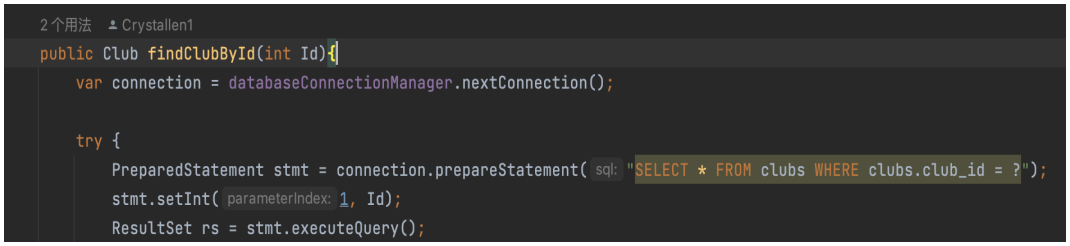


Figure 12. The screenshot of the FindClubById(int ID) method in ClubDataMapper

B. **CRUD Operations Encapsulated:** Each DataMapper class implements save(), update() and/or delete() methods, which take domain objects as input, extract their properties, and execute SQL INSERT or UPDATE statements to persist the object in the database.

C. **Handling related entities:** For domain objects that are related to other entities (e.g., Event linked to Club, or RSVP linked to Event), Data Mappers handle these relationships. For instance:

- RSVPDataMapper uses PostgreSQL arrays to store participant IDs.
- StudentClubDataMapper manages the many-to-many relationship between students and clubs, providing methods like findClubIdByStudentId and findStudentIdByClubId to retrieve the clubs a student belongs to and vice versa.

3.2.4 Consequences

Positive

- **Loose Coupling**: This pattern decouples the database access from the domain objects that represent the information in the database. This further allows the shape of the domain model to differ from the underlying database, as well as allowing the design of the domain layer to proceed independently of the design of the database layer.
- **Compatibility with domain model**: This pattern is highly compatible with the domain model pattern in section 3.1. The data mapper objects create domain model objects from the database. This allows high re-use of both the underlying database and the domain layer.

Negative

- **Complexity**: The Data Mapper pattern introduces an extra layer, which adds complexity. Every entity requires a corresponding Mapper class, leading to more code and higher maintenance effort.

3.3 Unit of Work

3.3.1 Context

When handling a single transaction that includes multiple database operations, we need to ensure the atomicity of the entire transaction. If a failure occurs during the process, these operations may leave the database in an inconsistent state. In our application, when deleting an event, all tickets associated with the event should also be deleted. If the database operation to delete the event succeeds but the operation to delete the tickets fails, this could result in “orphaned data” in the ticket table (data that will never be accessed and never deleted).

3.3.2 Decision

In our application, there are two instances where Unit of Work (UoW) is used. Both business logics involve modifying the contents of two different database tables. The first case is event delete: when a Club administrator deletes an event, the tickets previously obtained by users for this event also need to be deleted. The second case is RSVP submission: after a user submits an RSVP form for an event, the corresponding ticket is also saved along with the RSVP form.

3.3.3 Implementation Strategy (Use Case in the Application)

We will use **event deletion** as an example to introduce our UoW implementation. Since our UoW classes all implement a UnitOfWork interface, the RSVP submission follows a very similar process, with the only difference being the specific database operations involved.

Figure 13. The screenshot of UnitOfWork interface

We created a UoW class for the event deletion business logic. This class contains a list that holds the event IDs to be deleted, enabling our application to perform batch deletions. The core method in this class is commit(), where we first obtain a connection object from the connection pool (the details of the connection pool will be described in sections 3.10.1) and disable the auto-commit feature of this connection. Then, we iterate through the list, executing the DAO layer's delete event operation (modifying the event's status field) and finding the associated tickets, followed by executing the DAO layer's delete ticket operation (modifying the ticket's status field). Finally, we commit the transaction. If any database operation fails at any step, we catch the error and perform a transaction rollback.

Figure 14. The screenshot of EventDeleteUoW which implements UnitofWork interface

```java
public void commit() {
    Connection connection = null;
    try {
        connection = connectionManager.nextConnection();
        connection.setAutoCommit(false);
        for (Integer eventId:eventsId){
            eventRepository.deleteEvent(connection,eventId);
            List<Ticket> tickets = ticketRepository.getTicketsFromEvent(connection,eventId);
            for (Ticket ticket : tickets) {
                ticketRepository.deleteTicket(connection,ticket.getId());
            }
        }
        connection.commit();
    } catch (Exception e) {
        if (connection != null) {
            try {
                connection.rollback();
            } catch (SQLException rollbackEx) {
                rollbackEx.printStackTrace();
            }
        }
        throw new RuntimeException("Error committing UoW: " + e.getMessage());        }
    finally {
        if (connection != null) {
            try {
                connection.setAutoCommit(true);
                connectionManager.releaseConnection(connection);
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        clear();
    }
```

Figure 15. The screenshot of commit function in EventDeleteUoW

3.3.4 Diagram

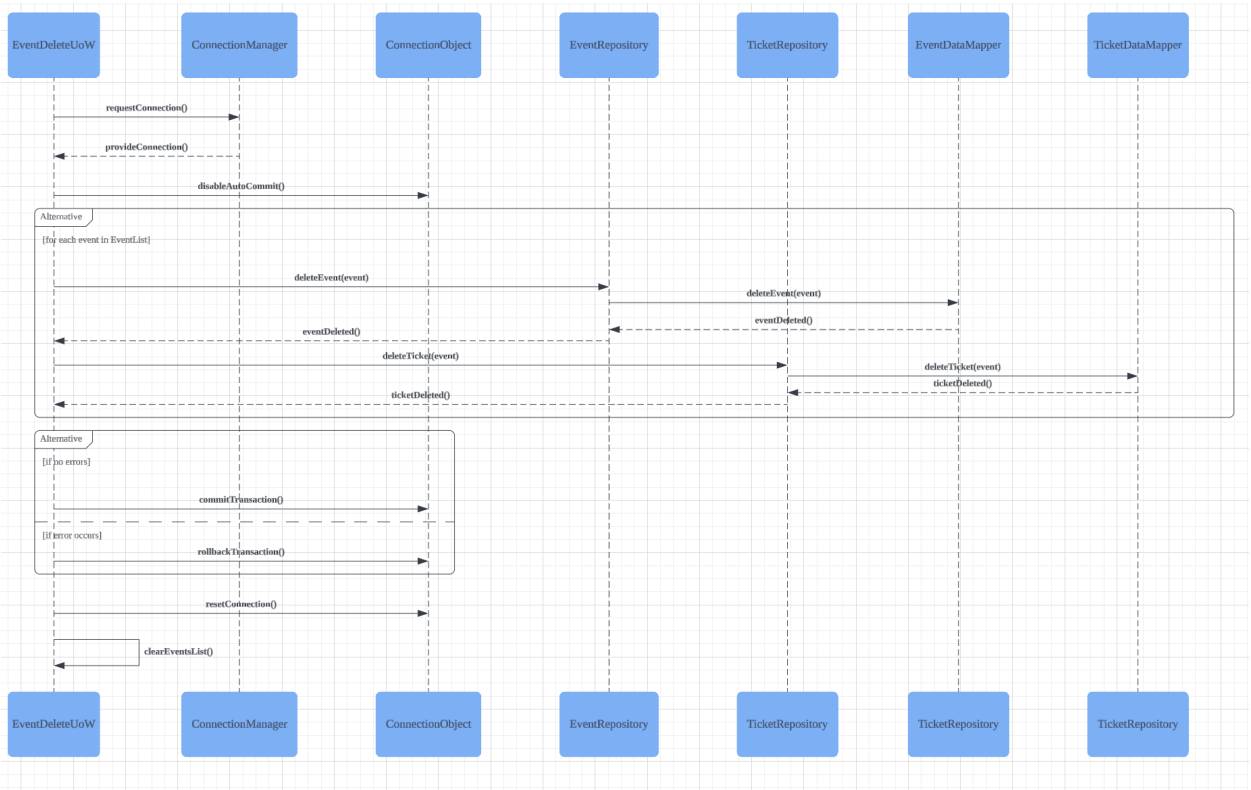


Figure 16. A sequence diagram that Delete a event in our implementation using UoW

3.3.5 Consequences

Positive

- **Simplicity:** The unit of work pattern is an elegant and simple way to keep track of objects. Even for the most trivial applications, using a unit of work is likely to be worthwhile.
- **Efficiency:** This is quite an efficient approach. Given a large set of objects with only minimal changes, committing to the database will be much faster, while the registration of objects would not take a considerable amount of resources.
- **High cohesion:** All information regarding what has been changed, added, or deleted is contained in a single place for each thread

Negative

- **Forgetfulness:** Whether using caller or object registration, if a developer forgets to register an object, the resulting fault may prove difficult to trace

3.4 Lazy Load

3.4.1 Context

In our application, many entities/domains are interrelated and associated with each other. In our application, the Club class has a Student field to represent the club administrator, and the Event field represents the activities organised by the club. However, if we instantiate the Student and Event objects every time creating a new Club object, it would lead to inefficiency and a waste of resources, as the Student and Event fields are only used in specific scenarios.

3.4.2 Decision

To address this issue, we have adopted the Lazy Load pattern, which delays the initialization of objects until they are actually needed.

3.4.3 Implementation Strategy

a. ***Create Proxy Field:*** *In each class, we have set a proxy field for each entity field, which corresponds to the entity's ID, as shown in Figure 17. When we read data from the database and initially instantiate the object, only the ID field of the corresponding entity is initialised.*

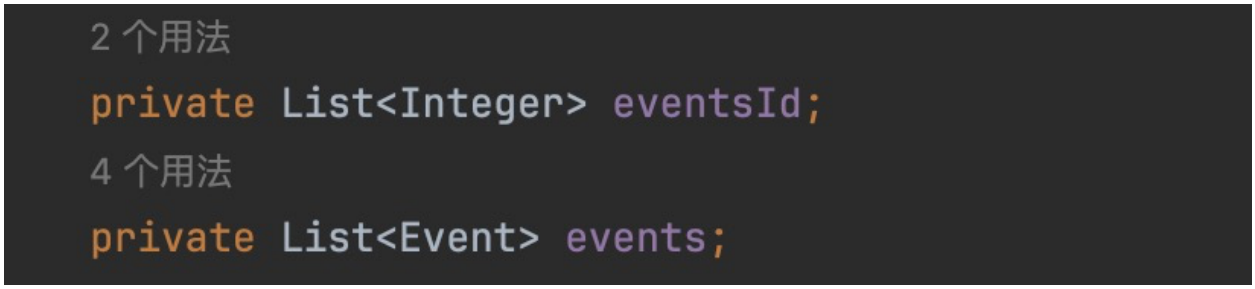


Figure 17. The screenshot of Proxy Filed in Event

b. ***Lazy Initialization***：*In the Repository, we have implemented the lazy load logic. For a specific entity object, we use the proxy field in the object to load the necessary entity field from the database, and then use a setter to attach this entity field to the object. Since we check the in-memory object in the service layer to determine if the entity field is null, the lazy load method is only called from the repository layer if the field is null. This allows us to read the field directly from memory next time, without needing to access the database again, as shown in Figure 18,19.*

```
public Club lazyLoadApplication(Club club){
    try {
        List<FundingApplication> fundingApplications =
                fundingApplicationMapper.findFundingApplicationsByIds(club.getFundingApplicationsId());
        club.setFundingApplications(fundingApplications);

        clubCache.put(club.getId(),club);
    } catch (SQLException e) {
        throw new RuntimeException("Error loading Application for club", e);
    }
    return club;
}
```

Figure 18. The screenshot of lazy load strategy in ClubRepository

```
public List<FundingApplication> getFundingApplication(Club club){
    if (club.getFundingApplications()==null||club.getFundingApplications().isEmpty()){
        club=clubRepository.lazyLoadApplication(club);
    }
    return club.getFundingApplications();
}
```

Figure 19. The screenshot of lazy load using in ClubController

3.4.4 Alternatives Considered

For loading entity class fields, we primarily employ two strategies and one special approach. For fields of types like Integer, other numeric types, String, or custom enum classes, we load them directly from the database when instantiating the object because these fields are small in size and initialise quickly. However, for certain entity fields, especially those involving lists of other entity classes (e.g., the student field in the club class, which is a list of Student objects), loading these fields directly during the instantiation of a club object would also instantiate multiple Student objects. This process is obviously inefficient and would consume a large amount of memory. These entity fields are usually only used in specific business logic contexts. Therefore, we adopt a lazy load strategy for these fields, which speeds up the initial instantiation and conserves memory when the field's information is not needed. The field is only loaded into memory the first time it's required.

In addition to these two strategies, we have a special handling for the Venue class. Considering that the total number of Venue objects is relatively small (as a school doesn't have too many venues), and updates are infrequent (new venues are not created often), we instantiate all Venue objects at the start of the program and store them in the Venue class's repository. Subsequently, other classes' Venue fields can be directly retrieved from memory.

3.4.5 Consequences

Positive

- **Efficiency:** The clear advantage of using lazy loading is the efficiency gains it can potentially deliver.

Negative

- **Complexity:** Clearly, implementing a system using lazy load is going to increase complexity over not using lazy load, so it should only be used in cases in which it is truly needed.
- **Inheritance:** If the domain hierarchy contains inheritance, it can cause confusion over what type of object to create. For example, the fields contained in the database may affect the type of object to create. If we do not load these, then we cannot know what type of object needs to be created if we are using lazy initialisation or ghost (or using generics in the value holder case).
- **3. Ripple loading:** A potential problem is with ripple loading. This is where many more database accesses are performed than is required. For example, our calling code may iterate over a collection of objects, which are each created when they are accessed. It may be much faster to query all of these at one time, and create the entire collection. Design trade-offs need to be made to decide whether to load the entire collection at once, or only load an object in the collection if it is accessed

3.5 Identity Field

3.5.1 Context

In our application, there is a need to maintain consistency between in-memory objects and their corresponding rows in the database. The system handles various entities like Students, Clubs, Events, and RSVPs, all of which need unique identifiers for reliable data persistence and retrieval. Without a clear mechanism to link these objects to their database rows, maintaining data integrity could become a challenge as the system grows. To address this, we require a pattern that ensures each in-memory object holds a unique identifier (ID) corresponding to its database entry, simplifying database operations like updates and deletions.

3.5.2 Decision

We will implement the Identity Field Pattern to associate each domain object in the application with a unique identifier (ID). This ID will be used to maintain a direct link between the objects and their corresponding database rows, ensuring data integrity and facilitating easier database operations.

3.5.3 Implementation Strategy

a. ***Create DomainObject Class:*** *We created an abstract DomainObject class with an 'id' field as shown in Figure 20. The 'id' field is used to identify each object in the database, which is a typical implementation of the identity field pattern, used to associate an object with a unique identifier (primary key) in the database during persistence.*

```
package org.teamy.backend.model;

6 个用法  6 个继承者  lvyl9909
public abstract class DomainObject {
    2 个用法
    private Integer id;

    lvyl9909
    public Integer getId() { return id; }

    lvyl9909
    public void setId(Integer id) { this.id = id; }
}
```

Figure20. The created DomainObject class

b. ***Inherit from DomainObject:*** *In our implementation, the entity classes(e.g., Ticket, Club, Event, RSVP, Venue, Funding Application) inherited from this class and will automatically have an 'id' for identification. Let we take RSVP for example, the RSVP class inherits the 'id' field from DomainObject, and this ID will correspond to the unique identifier of each RSVP record in the database. We call*

setId() inherited from DomainObject to assign the id field.

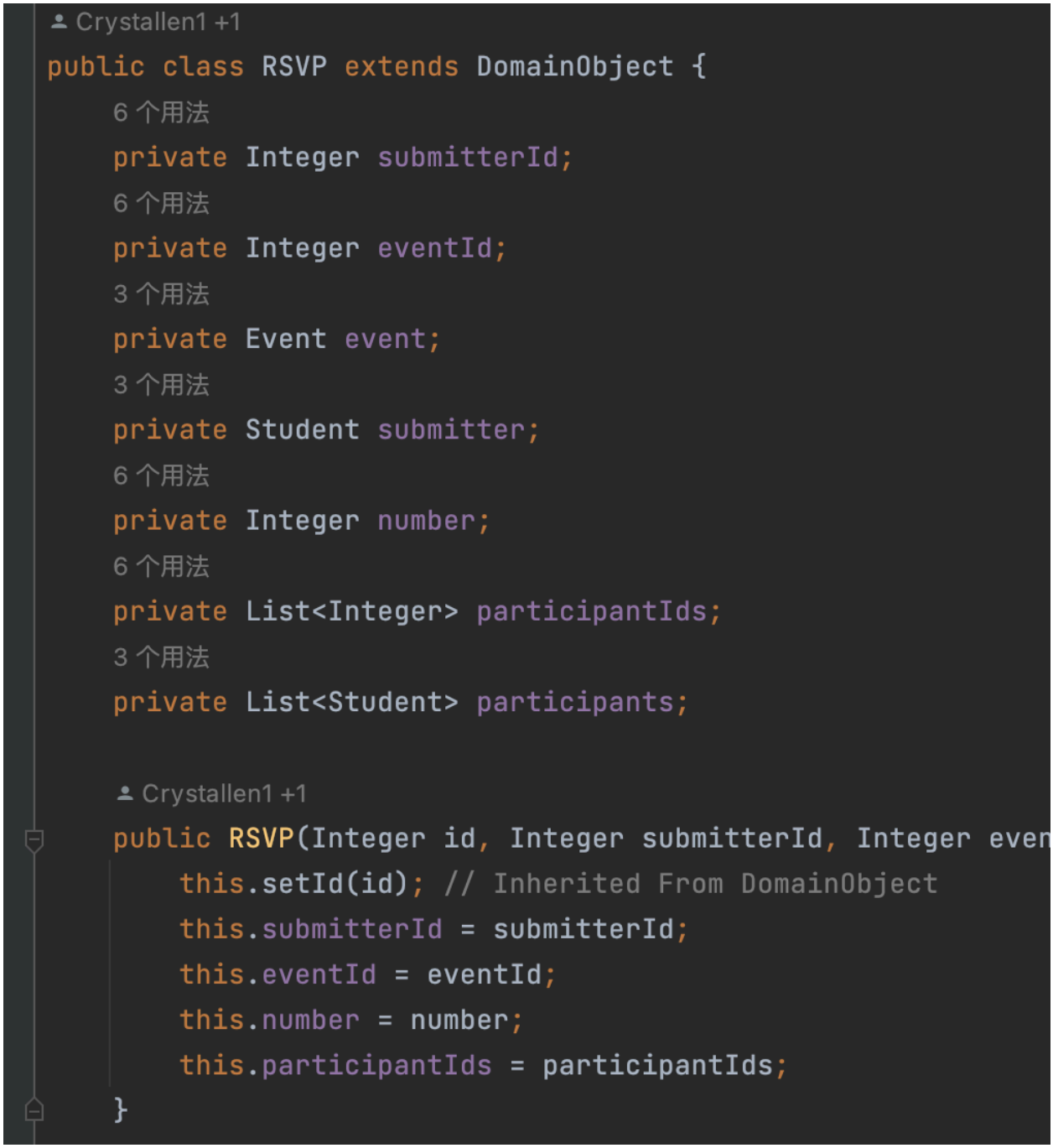


Figure21. RSVP inherit from DomainObject

3.5.4 Consequences

Positive

- **Simplicity:** The approach is simple and straightforward, which reduces the complexity of managing unique identifiers manually.
- **Easier Maintenance:** By centralizing the handling of ‘id’ fields in the DomainObject class, changes to how ids are managed only need to be made in one place.

Negative

- **Coupling:** Linking a domain object to its table key implies a higher coupling between the domain layer and the database.

3.6 Foreign Key Mapping

3.6.1 Context

The system uses object-relational mapping (ORM) to manage relationships between domain objects and database tables. The Foreign Key Mapping pattern is applied to maintain associations between these objects, specifically handling how references are persisted in the database through foreign keys. The Identity Field and Data Mapper patterns complement this approach by ensuring unique identifiers for entities and separating domain logic from persistence mechanisms. The system includes entities like RSVP, Event, and Club, which are linked to each other through foreign key relationships, ensuring data integrity and consistency.

3.6.2 Decision

To implement the Foreign Key Mapping pattern, each entity in the system must have a unique identifier (via the Identity Field Pattern) that links to a corresponding row in the database. Foreign keys are used to represent the relationships between these entities, such as RSVP linked to Event or Ticket linked to Student and RSVP. The Data Mapper pattern is chosen to manage the translation of these domain objects to database entries, allowing for efficient retrieval and manipulation of related objects through foreign keys.

3.6.3 Implementation Strategy

Figure22. Management System Database

- Identity Field Pattern

Each table (e.g., rsvps, tickets, events) will have a unique identifier (e.g., rsvp_id, ticket_id, event_id) as its primary key. These keys are represented in the domain model as fields inherited from a common DomainObject class, ensuring that each domain object has a corresponding unique identifier.

- Foreign Key Relationships

Foreign keys will be used in tables like rsvps and tickets to link the entities. For example, the rsvps table will include student_id and event_id as foreign keys to establish the relationship between RSVPs, students, and events.

- Data Mappers

Custom mappers like RSVPDataMapper will handle foreign key relationships, mapping domain objects (like RSVP) to their corresponding database rows. These mappers will extract and store the relevant foreign key IDs (e.g., event_id), ensuring correct associations between entities when inserting or updating data.

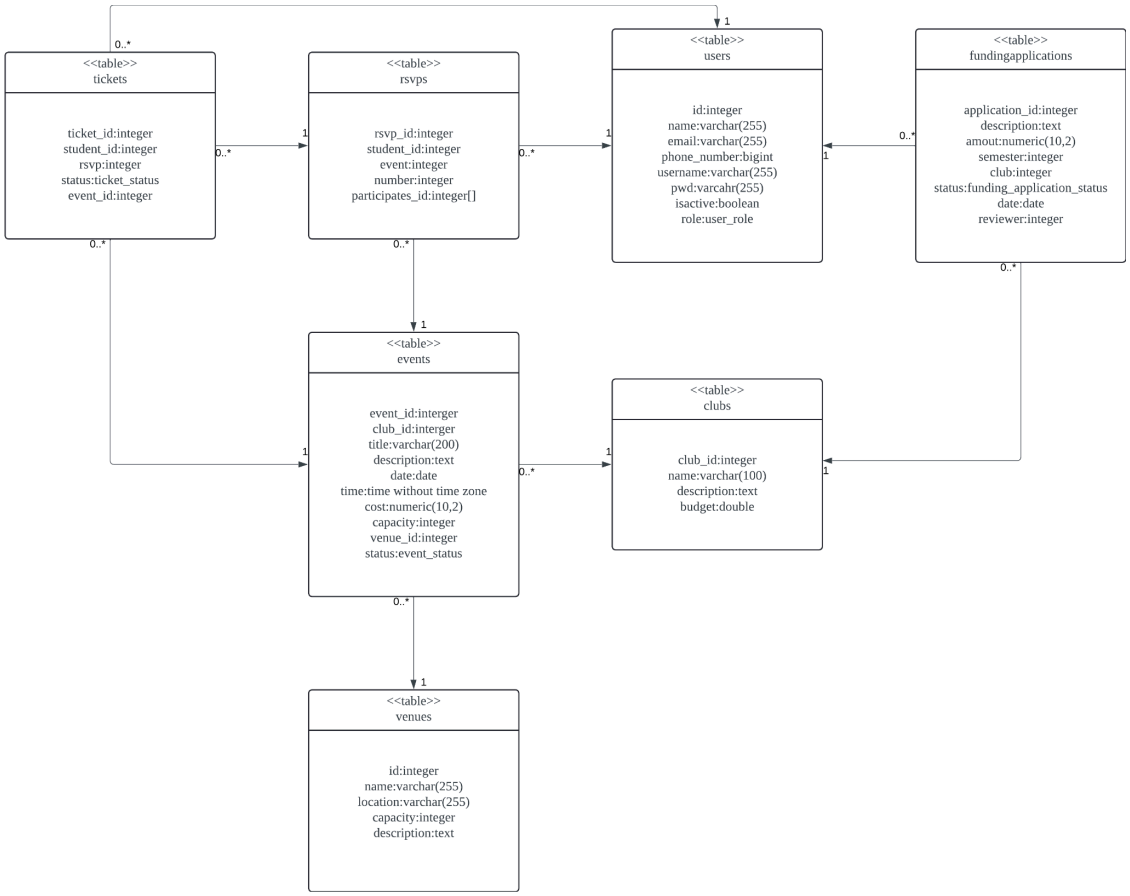


Figure23. Management System Database(Shows the foreign key)

3.6.4 Consequences

Positive:

- **Data Integrity**: By using foreign keys, relationships between entities such as RSVP and Event are enforced at the database level, ensuring referential integrity.
- **Separation of Concerns**: The Data Mapper Pattern decouples domain logic from persistence, promoting a cleaner and more maintainable architecture.
- **Cascading Operations**: Foreign key constraints enable operations like cascading deletes, ensuring that when a parent entity (like an Event) is removed, its associated entities (like RSVPs) are automatically managed.

Negative:

- **Complexity**: Managing foreign key relationships through Data Mappers adds an extra layer of complexity, as the system needs to account for loading and saving related entities.

- **Potential Tight Coupling**: While the domain logic is decoupled from persistence, there may still be a tight coupling between database schema and object design, particularly if schema changes impact the domain model significantly.

3.7 Association Table Mapping

3.7.1 Context

The Association Table Mapping pattern is used to manage many-to-many relationships between entities in a relational database, such as the relationship between Students and Clubs, where multiple students can belong to multiple clubs. Since relational databases do not natively support many-to-many relationships, an association table is required to maintain these relationships. This table stores references (foreign keys) to both entities, enabling efficient query performance while keeping the entities independent and manageable.

3.7.2 Decision

To handle many-to-many relationships in the system, the Association Table Mapping pattern is applied. The decision to use this pattern ensures clean decoupling of entities while allowing for efficient querying and maintenance.

3.7.3 Implementation Strategy

This pattern creates an intermediary association table that breaks down the many-to-many relationships into two one-to-many relationships. The association table will contain two foreign keys—one referencing each of the related entities (e.g., Students and Clubs). Our implementation as follows:

Association Table for Students and Clubs: Create a student_clubs table with two foreign keys:

student_id: references the students table.
club_id: references the clubs table.

This table manages the many-to-many relationship between Students and Clubs, ensuring that multiple students can join multiple clubs and vice versa.

3.7.4 Consequences

Positive

- **Data Normalisation**: The use of an association table ensures that relationships between entities are managed efficiently without data duplication, maintaining normalised data.
- **Decoupled Entities**: The pattern allows Students, Clubs, Funding Applications, and Events to remain independent of each other, reducing direct dependencies between entities. Changes in one entity do not affect the other.
- **Efficient Queries**: Queries involving relationships between entities (e.g., Students and Clubs) can be performed more efficiently through the association table. This method ensures scalable performance even with complex queries.
- **Flexibility**: The pattern enables easy management of relationships. For example, new students can be added to multiple clubs without altering the structure of either the students or clubs table.

Negative

- **Increased Complexity**: While the pattern simplifies entity relationships, it adds complexity by introducing additional association tables, which can increase the complexity of database schema design and query writing.
- **Potential Performance Overhead**: In certain scenarios, joining multiple tables (e.g., Students, Clubs, and the association table) could introduce performance overhead, especially when the number of related entities grows.

3.7.5 Diagrams: ERD or database schema

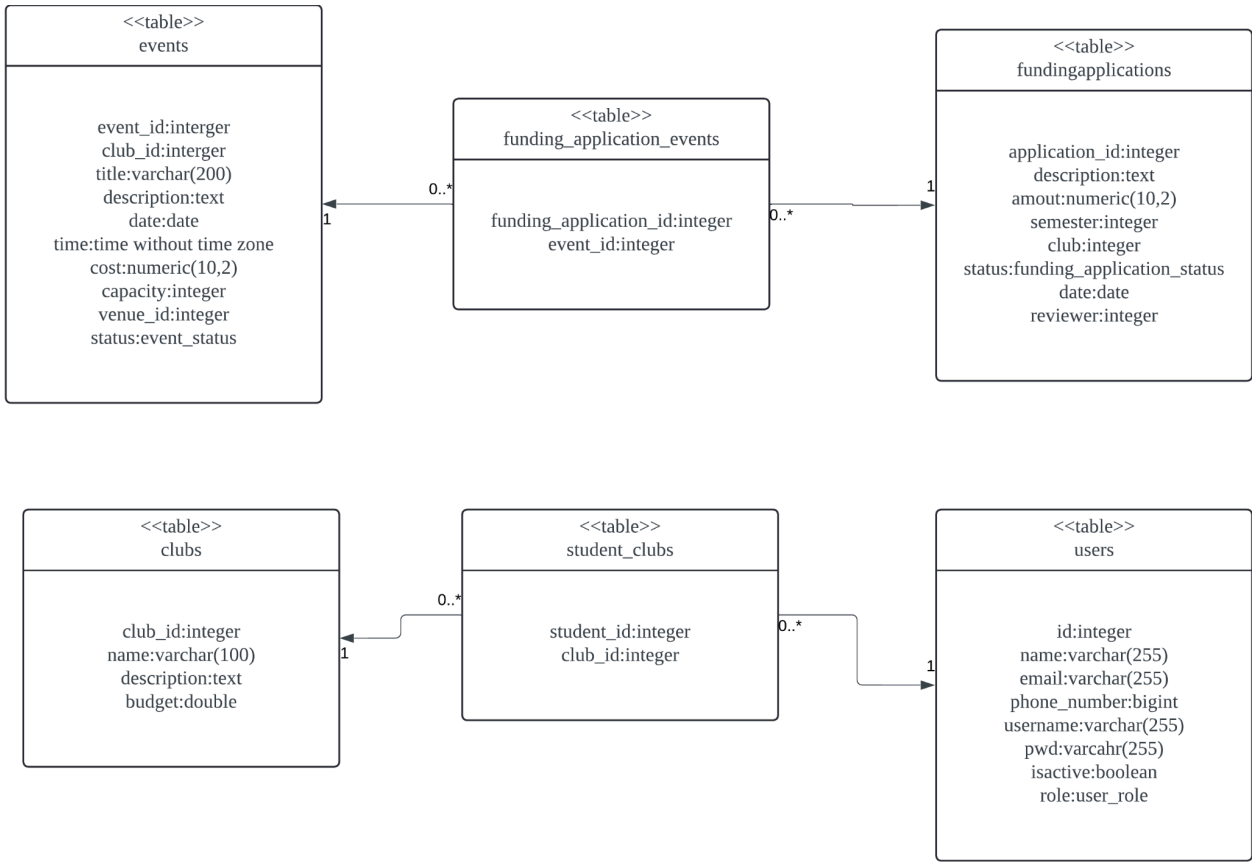


Figure24. Management System Database(Shows the association)

3.8 Inheritance Patterns

3.8.1 Context

The inheritance strategies include: Single Table Inheritance, Class Table Inheritance, and Concrete Table Inheritance. These reflect the relationships between parent and child classes in the database.

a. **Single Table Inheritance**: Each row in the database table represents an instance of one of the subclasses, and a special column is used to identify which subclass the row belongs to.
b. **Class Table Inheritance**: Each class (including the base class and its subclasses) has its own table. The subclass tables store only the attributes specific to that class and are linked to the base class table via a foreign key.

c. **Concrete Table Inheritance**: Each subclass has its own table, which contains all the fields for that subclass (including fields inherited from the base class). There is no shared base table, and each subclass table is independent.

3.8.2 Decision

In our application, we use inheritance patterns to manage common functionality across multiple classes, while allowing subclasses to implement their own unique features. Inheritance is a fundamental object-oriented programming principle that promotes code reuse, reduces redundancy, and simplifies the maintenance of related classes.

3.8.3 Implementation Strategy

In our application, two inheritance patterns are applied.

The first instance is in the Identity Field pattern (Jump to Section 3.5 for details), where we define an abstract class to represent the mapping between entity classes and the database primary key. Each entity class inherits from this abstract class. For more details, refer to section 3.5.

The second instance is in the distinction of user Roles. We first define a base class Person, which implements the UserDetails interface from Spring Security. This class contains basic user information such as username, password, email, and so on. To differentiate between Student and Faculty Admin users, we define two classes for these entities, both of which inherit from the Person class. The Student class stores information related to students, such as Club and Ticket, while the Faculty Admin class stores information related to FundingApplication and other faculty admin-related data.

```
public class Person implements UserDetails {
    5 个用法
    private Long id;
    6 个用法
    private String username;
    9 个用法
    private String name;
    9 个用法
    private String email;
    8 个用法
    private Long phoneNumber;
    7 个用法
    private String password;
    8 个用法
    private boolean isActive;
    8 个用法
    private Set<Role> roles = new HashSet<>();
```

```
public class Student extends Person{
    6 个用法
    private List<Integer>rsvpsId;
    6 个用法
    private List<RSVP>rsvps;
    10 个用法
    private List<Integer>clubId;
    6 个用法
    private List<Integer>ticketsId;

    7 个用法
    private List<Ticket>tickets;
    6 个用法
    private List<Club>clubs;
```

```
public class FacultyAdministrator extends Person{
    7 个用法
    private List<FundingApplication> FundingApplications;
```

Figure25,26,27. Student and Faculty Administrator inherit from Person

3.8.4 Alternatives Considered

For the two inheritance patterns we implemented, different database mapping strategies were chosen.

For the first Identity Field inheritance, we applied the Concrete Table Strategy, where each subclass is managed by its own table, and all tables contain the parent class attribute id. Since all subclasses must include the id field from the parent class, which serves as the primary key in the database and the unique identifier for entity objects, using the Concrete Table Strategy simplifies CRUD operations for each subclass. It avoids the need to query other related tables for information.

If the Single Table Strategy were used, there would be a significant amount of redundancy due to the numerous distinct fields in each subclass, which would increase the complexity of queries and degrade database performance.

Using the Class Table Strategy would involve creating a separate table for the parent class, but since the parent class only contains an id field, this would unnecessarily complicate the system and increase query complexity without offering any advantages. The parent class is merely an abstract class in the application, and it doesn't carry any meaningful information on its own.

In summary, for the id inheritance, the Concrete Table Strategy can simplify CRUD operations by giving each subclass its own table with the parent class's id field, avoiding redundant queries. In contrast, the Single Table and Class Table strategies would introduce unnecessary complexity and redundancy due to the distinct fields in each subclass and abstract nature of the parent class.

For the second inheritance of the Person class, we used the Single Table Strategy, where both the parent and child classes are stored in a single table. The main reason for this design is to simplify the process during authentication and authorization in the

system. In our custom UserDetailsService for Spring Security, user information is loaded in the loadUserByUsername method. Based on the username passed from the frontend, we need to determine whether the user is a student or a faculty admin and assign different permissions accordingly.

If we were to use the Concrete Table Strategy, we would need to ensure that there are no duplicate usernames in both the student and admin tables. Additionally, during login, we would need to query both tables separately, which could lead to a situation where both a student and an admin object are retrieved simultaneously, causing ambiguity in permission assignment.

The Class Table Strategy could meet our requirements, but compared to the Single Table Strategy, it requires managing two additional database tables and setting up appropriate foreign keys. During login, after reading the parent class table, we would need to query the corresponding child class table again. This added complexity does not provide any clear advantages.

While the Single Table Strategy might result in some empty columns, this is considered acceptable since our two child classes don’t contain many additional fields (around 2 per subclass). Moreover, we have implemented lazy loading for the unique fields of student and admin objects, ensuring the simplicity of queries.

In summary, the Single Table Strategy was chosen for the Person class to streamline authentication and authorization, avoiding separate queries for student and admin. While it may introduce some empty columns, it simplifies permission assignment and reduces query complexity compared to the Concrete or Class Table strategies.

3.9 Authentication and Authorization

3.9.1 Context

In our application, authentication and authorization are critical parts to ensure that users can securely access the system and are granted appropriate permissions based on their roles. Our system involves multiple user roles, including administrators, students, and club managers, each having different levels of access to features such as accessing the interface and managing clubs.

3.9.2 Decision

a. Authentication Strategy

There are two mainstream methods for authentication. One is to use a session ID stored in a cookie to track the user's identity in the browser, and the other is to use a token stored in local storage to determine the user's identity. The first method has a major flaw—it cannot prevent CSRF (Cross-Site Request Forgery) attacks. This is because when we use the session ID stored in a cookie for authentication, every request made by the browser will automatically include this session ID. As a result, if a user clicks on a malicious link, the server will recognize the erroneous request as coming from the user. In contrast, a token stored in local storage perfectly addresses this issue, as only requests written in the client's JavaScript code will include the token, while a standalone request link will not carry the token.

However, tokens also have their own issues. First, since tokens are not stored in cookies, they cannot be set as httponly, making them vulnerable to XSS (Cross-Site Scripting) attacks. Second, once a token is issued, it is difficult to revoke its permissions before its expiration time. Both of these issues can be mitigated by setting a very short token expiration time. However, this would lead to users having to log in frequently, which negatively impacts the user experience.

So, In our project we use an access/refresh token strategy. When a user login, the backend generates both an access token and a refresh token. The refresh token is stored in the database, and the access token is sent to the frontend, while the refresh token's ID is set in a cookie. The frontend stores the access token in local storage, and all requests will include the access token. When the access token expires, the backend uses the refresh token ID from the cookie to retrieve the refresh token from the database. It then checks whether the access token ID in the refresh token matches the expired access token's ID. If they match, a new access token is generated.

b. Authorization Strategy

In our application, the RBAC (Role-Based Access Control) model is used as the authorization management system. The RBAC model constructs a "User-Role-Permission" authorization model. In our application, each user object has a Role field. When a user logs in, the role information is retrieved from the Student table and the Student_Club table in the database and set in the Role field. Then, in the controller layer, during URL path parsing, permission checks are performed. Only certain roles are allowed to access specific URLs, while others will receive a 403 Forbidden error.

3.9.3 Implementation Strategy

In our specific implementation, we use the Spring Security framework for management.

a. Authentication Strategy

For the authentication part, we need to provide two methods for Spring Security. The first method uses Spring Security's built-in UsernamePasswordAuthentication. We simply need to have our user class implement the UserDetails interface and create a service class that implements UserDetailsService. This allows us to authenticate by customising the loadUserByUsername method to retrieve the user password from the database and compare it with the password from the frontend form.

Figure 28,29,30. Three key parts to implement UsernamePasswordAuthentication in Spring Security

The second method is our custom TokenAuthentication. We implemented a custom TokenAuthenticationFilter, which parses the JWT sent from the frontend to extract the username and roles. Then, we use Spring Security's built-in authentication object methods to validate these roles.

```java
@Override
public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) throws AuthenticationException{
    var authorizationHeader = Optional.ofNullable(request.getHeader(HEADER_AUTHORIZATION))
            .orElseThrow(() -> new BadCredentialsException(String.format("%s header is required", HEADER_AUTHORIZATION)));
    var matcher = PATTERN_TOKEN.matcher(authorizationHeader);
    if (matcher.find()) {
        var token = matcher.group(1);
        var authentication = jwtTokenService.readToken(token);
        try {
            var result = getAuthenticationManager().authenticate(authentication); // 验证对象
            return result;
        } catch (AuthenticationException e) {
            System.out.println("Authentication failed: " + e.getMessage());
            throw e;
        }
    }
    throw new BadCredentialsException(String.format("invalid %s header value", HEADER_AUTHORIZATION));
}
```

Figure 31. Custom TokenAuthenticationFilter *in our application*

b. Authorization Strategy

In Spring Security, permissions are generally retrieved during the loadUserByUsername process. In our application, we first obtain the user's basic roles: student or club administrator (in this stage, we only need to implement the student functionality, and in the next stage, the roles will be retrieved from the database). After that, we query the Student_Club table in the database to determine which club administration permissions the student has.

```java
@Override
public Person loadUserByUsername(String username) throws UsernameNotFoundException {
    Person user=null;
    try {
        user = studentRepository.findUserByUsername(username);
        if (user == null) {
            System.out.println("username not found");
            throw new UsernameNotFoundException("User not found");
        }else {
            Set<Role> roles = new HashSet<>();
            if (user instanceof Student) {
                roles.add(new Role( roleName: "USER"));
            }else{

            }
            List<Integer> clubIds = studentClubRepository.findClubIdByStudentId(Math.toIntExact(user.getId()));
            for (Integer clubId : clubIds) {
                roles.add(new Role( roleName: "CLUB_" + clubId));
            }
            user.setRoles(roles);
            System.out.println("yong hu quan xian :"+user.getAuthorities());
            System.out.println(user.getPassword());
        }
        return user;
    } catch (SQLException e) {
        throw new UsernameNotFoundException("Database error", e);
    }
}
```

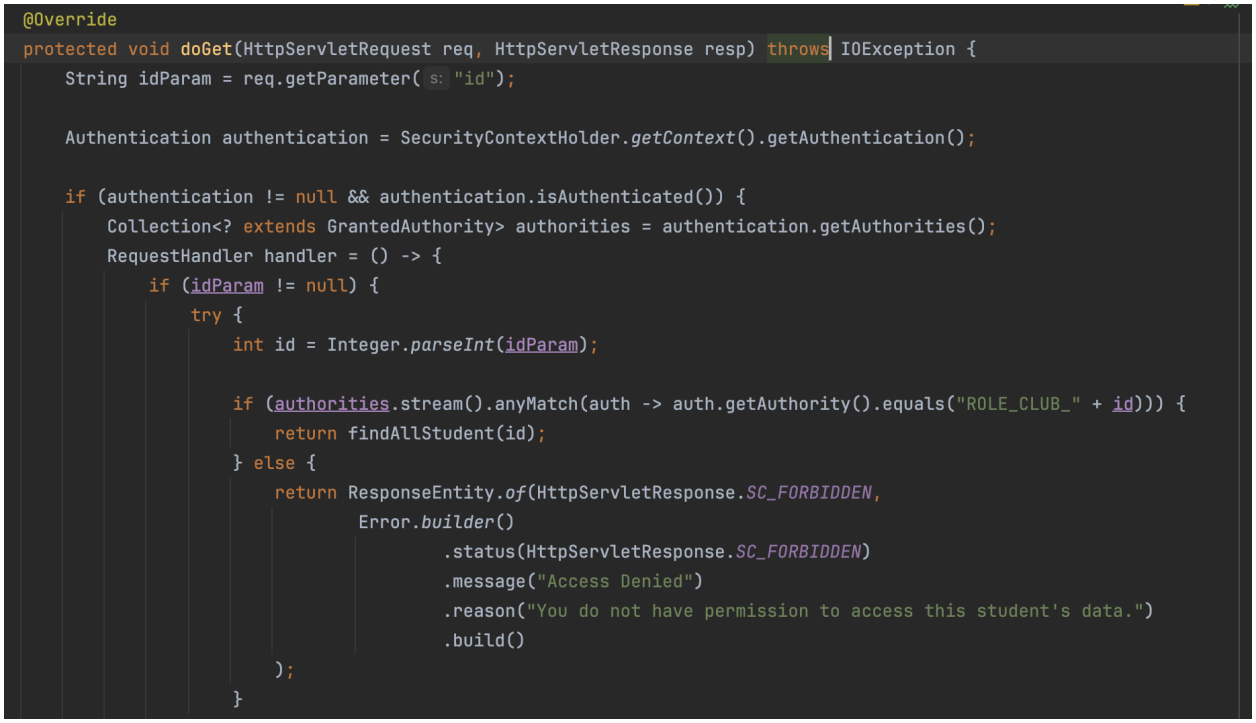
Figure 32. Override loadUserByUsername to set Authority in Role in CustomUserDetailsService

Spring Security allows for two different types of permission management. The first method is configured directly in the WebSecurityConfig file. For example, in our application, users with the ROLE_USER permission can only access resources under the /student/ path, while ROLE_ADMIN users can only access resources under the /admin/ path.

```java
    .authorizeHttpRequests(authorize -> authorize
            .requestMatchers("/auth/token").permitAll() AuthorizationManagerRequestMat...
            .requestMatchers(ADMIN_PROTECTED_URLS) AuthorizedUrl
            .hasAuthority("ROLE_ADMIN") AuthorizationManagerRequestMat...
            .requestMatchers(STUDENT_PROTECTED_URLS) AuthorizedUrl
            .hasAuthority("ROLE_USER") AuthorizationManagerRequestMat...
            .anyRequest() AuthorizedUrl
            .permitAll())
```

Figure 33. Spring Security Filter Chain to set some Permissions to Role

The second method is configured within the controller, offering finer-grained control. After logging in, Spring Security stores the instantiated object that implements UserDetails in the security context. When performing permission checks in the controller, we simply retrieve this object from the security context and check its permissions.



```java
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
    String idParam = req.getParameter( s: "id");

    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

    if (authentication != null && authentication.isAuthenticated()) {
        Collection<? extends GrantedAuthority> authorities = authentication.getAuthorities();
        RequestHandler handler = () -> {
            if (idParam != null) {
                try {
                    int id = Integer.parseInt(idParam);

                    if (authorities.stream().anyMatch(auth -> auth.getAuthority().equals("ROLE_CLUB_" + id))) {
                        return findAllStudent(id);
                    } else {
                        return ResponseEntity.of(HttpServletResponse.SC_FORBIDDEN,
                                Error.builder()
                                        .status(HttpServletResponse.SC_FORBIDDEN)
                                        .message("Access Denied")
                                        .reason("You do not have permission to access this student's data.")
                                        .build()
                        );
                    }
```

Figure 34. Set Role Permissions in the Controller Layer in ClubController

3.9.4 Diagram

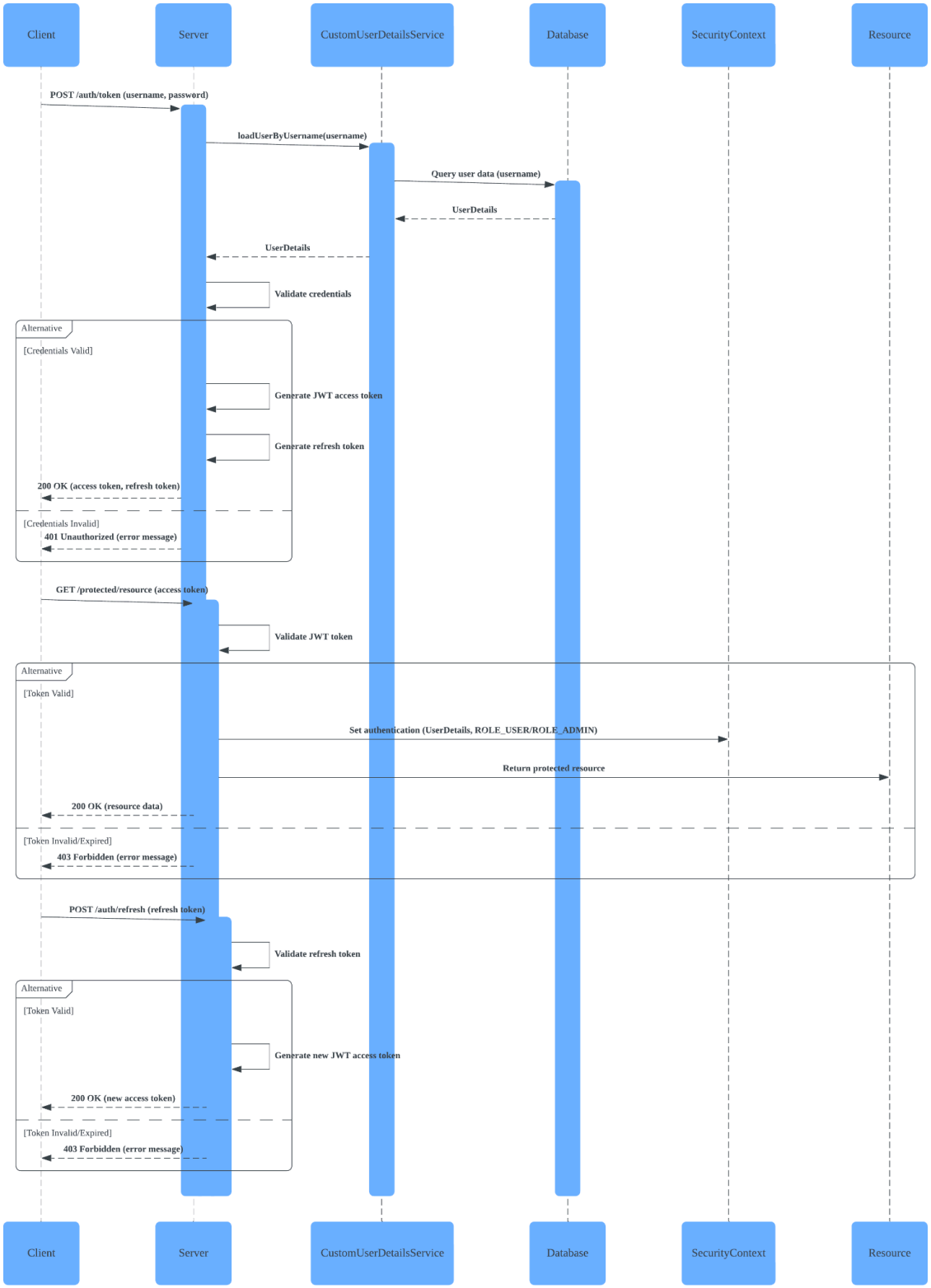


Figure 35. Sequence diagram of the Authentication process

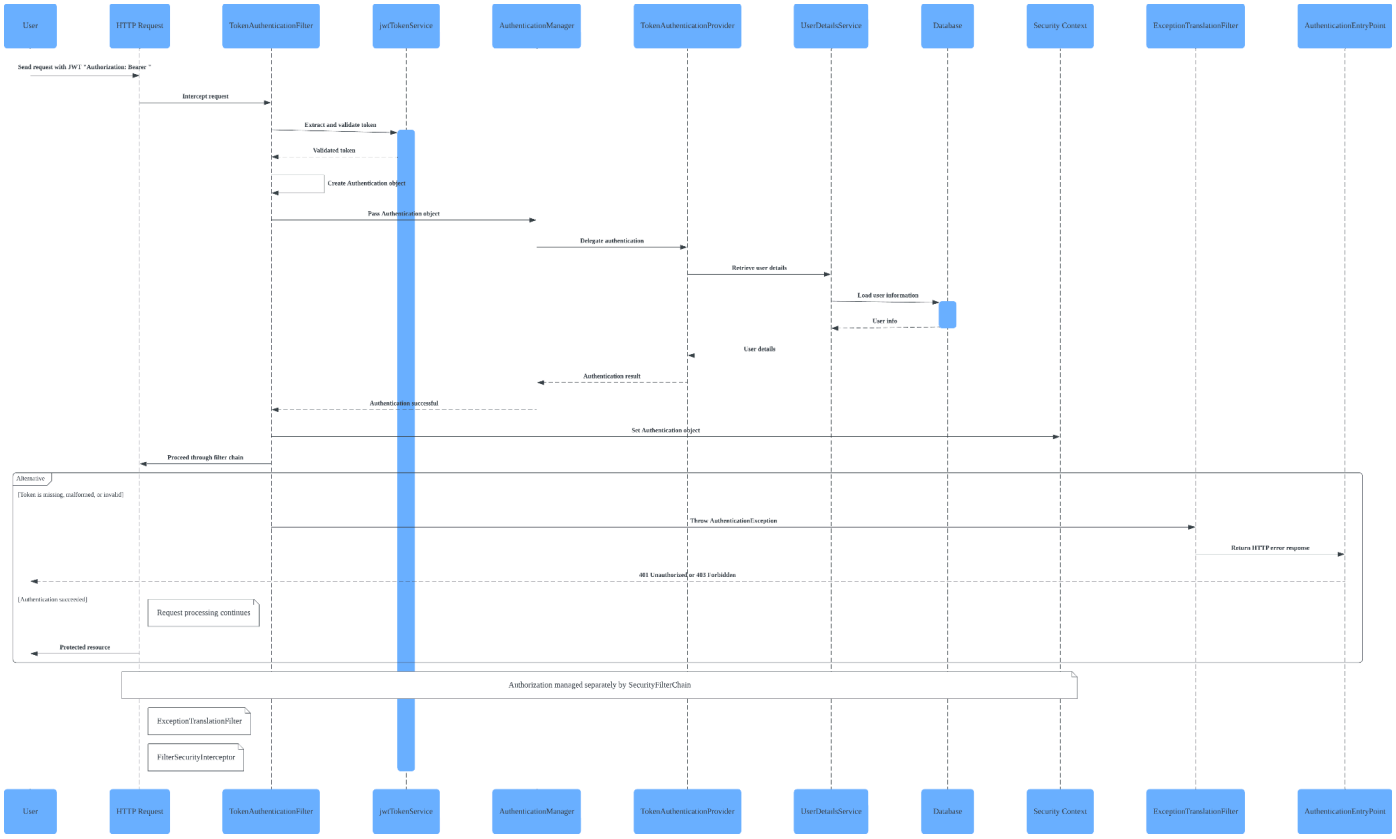


Figure 36. Sequence diagram of the Authorization process

3.9.5 Consequences

Positive

- **Improved security:** By centralising the authentication, this pattern reduces the number of places within the presentation layer where the mechanism is accessed, and therefore reduces the chance of security holes due to incorrect usage.
- **Maintainability:** Encapsulating the security policy in a centralised module supports migration to different types of authentication; e.g. if business requirements change.
- **Reuse:** Encapsulating the code consolidates authentication into a single place, which both reduces duplicate code within the system via reuse, and supports reuse in other systems.

Negative

- **Making it usable:** Experience suggests that if the authentication mechanism is difficult to use or does not provide the functionality as expected, developers will add their own authentication mechanisms directly into their applications. This increases the risk of vulnerabilities that can be exploited to gain access.

3.10 Additional Strategy

3.10.1 Database connection pool

In testing, we found that the first database connection operation after each startup is very slow. This is because the first operation after startup includes the initialization of the database connection, which is typically time-consuming.

To address this, we defined a DatabaseConnectionManager class to implement a connection pool. This pool maintains a fixed number of database connections using a blocking queue, establishing persistent connections to the database when the backend application starts. The initialization process involves loading the PostgreSQL JDBC driver and filling the connection pool with a fixed number of connections in the init() method. When a connection is needed, the nextConnection() method retrieves one from the pool, waiting up to 100 milliseconds if necessary. If no connection is available within this timeout, it may return null. After using the connection, the releaseConnection() method returns it to the pool, again waiting up to 100 milliseconds to re-add the connection to the pool.

This class ensures thread safety by using a blocking queue and handles potential interruptions by reasserting the thread's interrupt status. This design allows for efficient reuse of database connections in a multithreaded environment, reducing the overhead of establishing new connections for each database operation.

```java
public class DatabaseConnectionManager {
    1 个用法
    private static final int MAX_CONNECTIONS = 10;
    2 个用法
    private static final Duration ACQUIRE_CONNECTION_TIMEOUT = Duration.ofMillis(100);
    2 个用法
    private final String url;
    2 个用法
    private final String username;
    2 个用法
    private final String password;
    5 个用法
    private final BlockingDeque<Connection> connectionPool;
    // Private constructor to prevent instantiation
    Crystallen1
    public DatabaseConnectionManager(String url, String username, String password) {
        this.url = url;
        this.username = username;
        this.password = password;
        this.connectionPool = new LinkedBlockingDeque<>();
    }

    Crystallen1
    public void init() {
        try {
            Class.forName( className: "org.postgresql.Driver");
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
        while (connectionPool.size() < MAX_CONNECTIONS) {
            connectionPool.offer(connect());
        }
    }
```

```java
1 个用法  Crystallen1
private Connection connect() {
    try {
        return DriverManager.getConnection(url, username, password);
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
Crystallen1
public Connection nextConnection() {
    try {
        return connectionPool.poll(ACQUIRE_CONNECTION_TIMEOUT.toMillis(), TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        throw new RuntimeException(e);
    }
}

Crystallen1
public void releaseConnection(Connection connection) {
    try {
        connectionPool.offer(connection, ACQUIRE_CONNECTION_TIMEOUT.toMillis(), TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        throw new RuntimeException(e);
    }
}
```

Figure 37,38. The detailed implementation of database connection pool.

3.10.2 Cache

In the initial design, the majority of our GET requests directly accessed the database, leading to inefficiency and placing a heavy load on the database. To address this, we plan to use caching to alleviate the issue. We set up caching in the repositories of several commonly used entity classes. When a user queries for the first time, the data from the database is loaded into the cache. During subsequent update or lazy loading operations, the cache is updated to ensure the accuracy of the cached information.

```
private final Cache<Integer, Event> eventCache;


1 个用法  Crystallen1 *
private EventRepository(EventDataMapper eventDataMapper, VenueDataMapper venueDataMapper, ClubDataMapper clubDataMapper) {
    this.eventDataMapper = eventDataMapper;
    this.venueDataMapper = venueDataMapper;
    this.clubDataMapper = clubDataMapper;

    this.eventCache = CacheBuilder.newBuilder()
            .maximumSize(100)
            .expireAfterWrite( duration: 30, TimeUnit.MINUTES)
            .build();
}
```

Figure 39. Cache Initialization in the Repository Layer

```
public Event findEventById(int id) {
    Event event = eventCache.getIfPresent(id);
    if (event != null) {
        return event;
    }

    event = eventDataMapper.findEventById(id);
    event.setVenue(venueDataMapper.findVenueById(event.getVenueId()));
    event.setClub(clubDataMapper.findClubById(event.getVenueId()));
    if (event != null) {
        event.setVenueName(venueDataMapper.findVenueById(event.getVenueId()).getName());

        eventCache.put(id, event);
    }
    return event;
}
```

Figure 40. An Example of Applying Cache in the Repository Layer to Reduce Database Access in EventRepository

Since there is a degree of coupling between our entity classes, when a coupled class is updated, we also need to update the cache of the associated class. Currently, our strategy is to directly clear the cache of the affected entity class, requiring the data to be reloaded from the database on the next read.

```
public void invalidateClubCache(Integer clubId) { clubCache.invalidate(clubId); }
```

Figure 40. Invalidate Mechanism to Ensure Cache Information is Up-to-Date

At this stage, the task is limited to a single-threaded environment, so we have not yet considered caching update strategies in a multithreaded context. The current implementation only ensures usability in a single-threaded scenario. Further improvements will be made in the next phase.