Software Architecture Design

SWEN90007

SWEN90007 SM2 2020 Project

Magic Pigeons

In charge of:
Simai Deng: simaid @student.unimelb.edu.au
Jiayu Li: jiayul3@student.unimelb.edu.au
Yiran Wei: yirwei@student.unimelb.edu.au

SCHOOL OF
COMPUTING &
INFORMATION
SYSTEMS

Revision History

Date	Version	Description	Author
19/09/2020	02.00-D01	Initial draft	Simai Deng, Jiayu Li, Yiran Wei
25/09/2020	02.00-D02	Added Section 4 and 5 details to the document	Simai Deng, Jiayu Li, Yiran Wei
03/10/2020	02.10	The first version of the document	Simai Deng, Jiayu Li, Yiran Wei
04/10/2020	02.20	Improved the document and made the final version	Simai Deng, Jiayu Li, Yiran Wei

Contents

1. Introduction

This document specifies the design architecture of an online examination system, the project of SWEN90007 Software Design and Architecture. This document records the design rationale behind the system and the design patterns used, explained with the aid of domain model, class diagrams, component diagram, and sequence diagrams.

1.1 Proposal

This document will be updated as the project progresses and will last for the whole software development lifecycle. It aims to present an overview of the architecture design of the system at a higher level and detailed views of the use of design patterns at the lower level.

1.2 Target Users

This document is aimed at the project team, with a consolidated reference to the research and evolution of the system with the main focus on technical solutions to be followed.

2. Domain Model

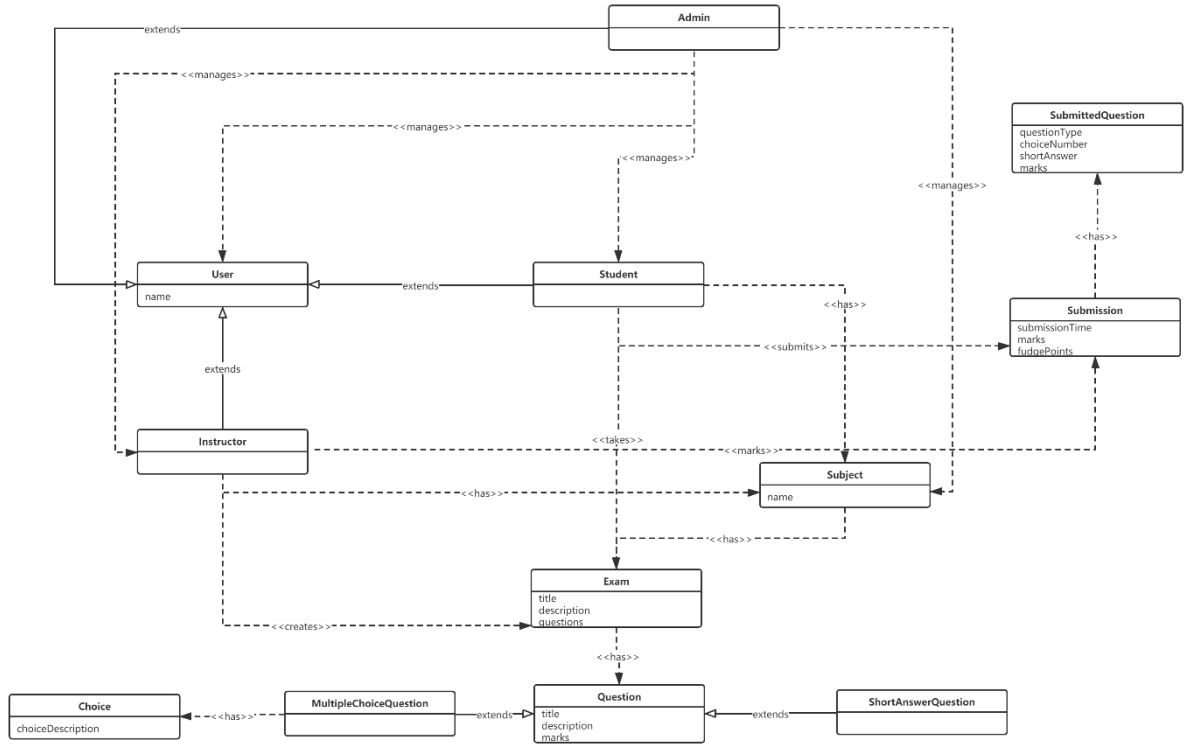

Figure 2.1 The domain level architecture design

3. Database Diagram

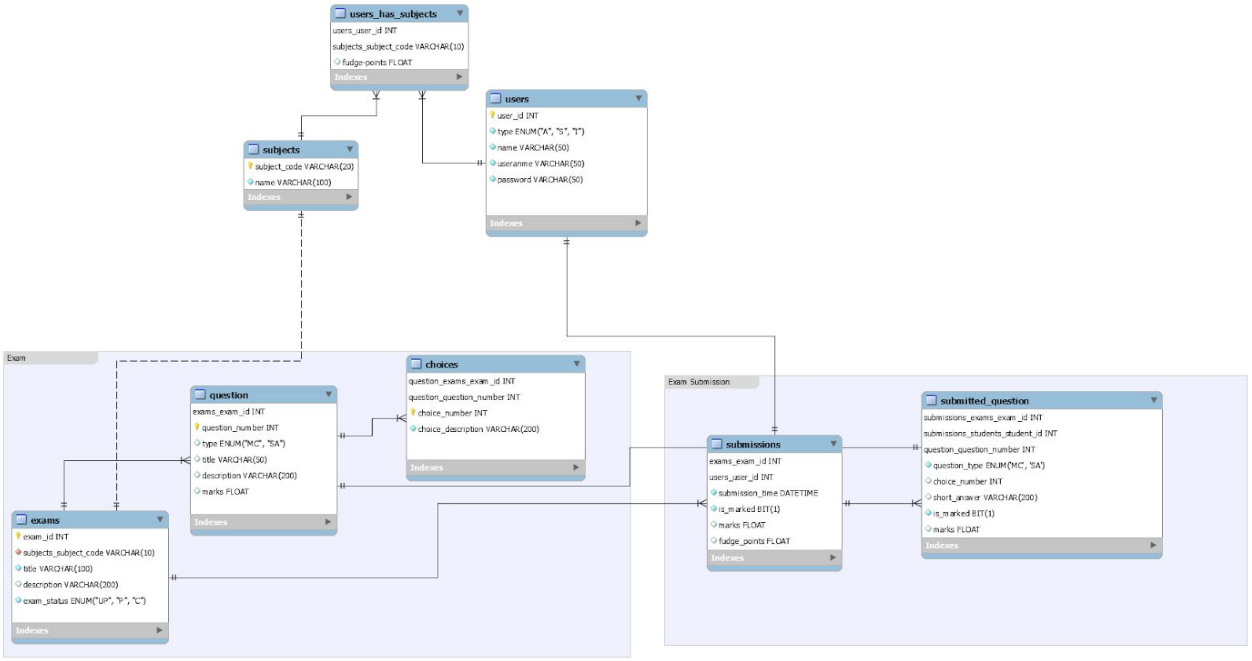

Figure 3.1 Database schema

The above diagram illustrates the database design of the system. Note this is modelled using MySQL workbench and the grammar differs slightly from PostgreSQL. Here is a table illustrating the difference:

MySQL	PostgreSQL
ENUM	TYPE
INT AUTO INCREMENT	SERIAL
BIT	BOOLEAN

The actual SQL script can be found in Appendix A.

4. Class Diagram

In this section, we will introduce the class design of our system. Since showing the whole system in a single class diagram is impossible due to its complexity, we will break the system down into components and introduce each component and relationship between them.

4.1 Domain Objects

Figure 4.1.1 shows all the domain objects used in the system. User is one of the abstract classes which will be extended by either Admin, Instructor, or Student. Question is another abstract class which will be extended by either ShortAnswerQuestion or MultipleChoiceQuestion.

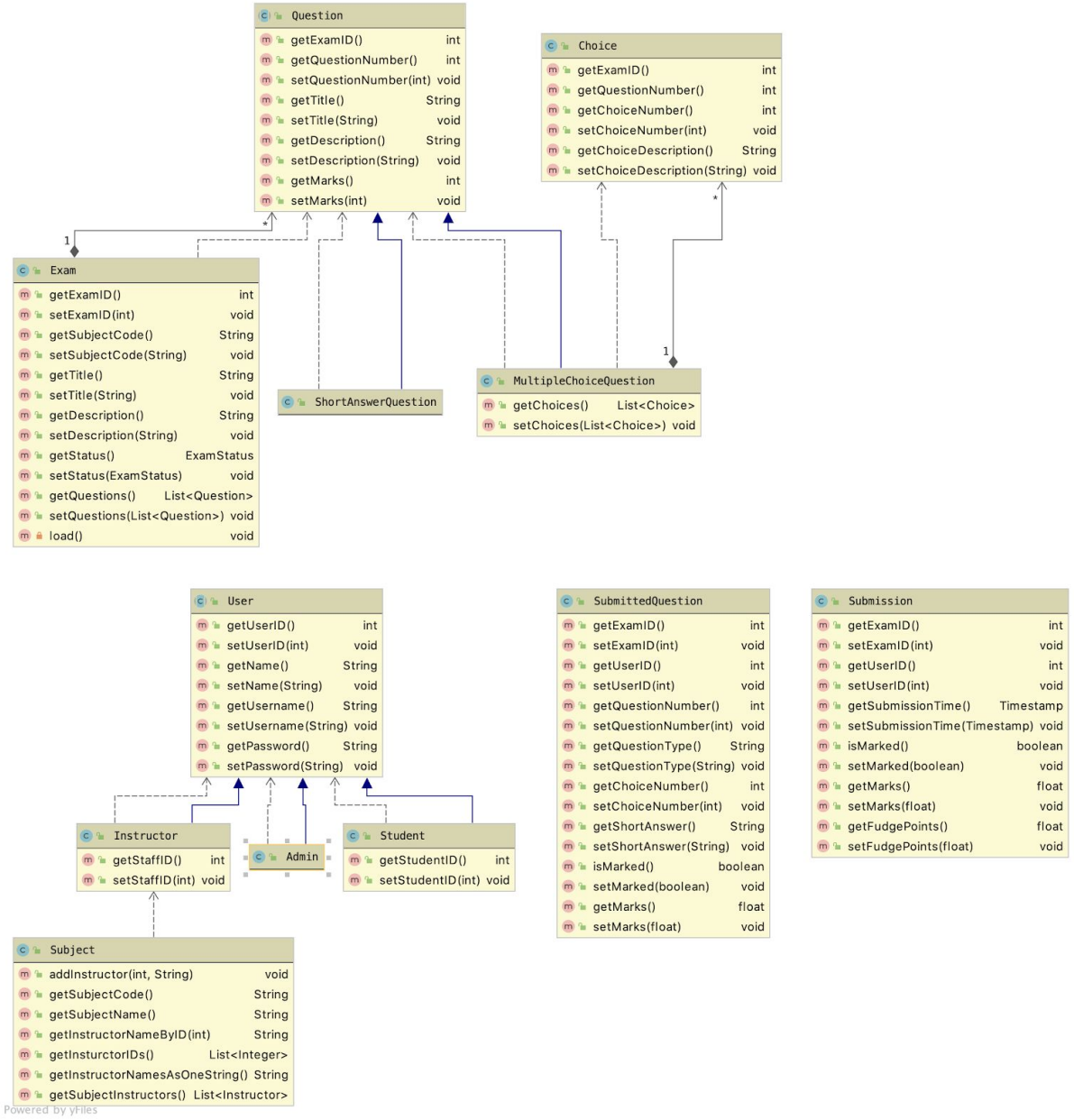

Figure 4.1.1 Class diagram of domain objects

4.2 Data Mappers

The data mappers are an essential layer in the software that separates the domain objects from the database. Figure 4.2.1 gives an overview of the dependencies between all data mapper classes. Among the data mappers, question mapper and choice mapper have weaker coupling with other data mappers and the front end logic because of the use of foreign key mapping. Choices are saved as a list of Choice objects in question objects, and questions are saved as a list of Question objects in exam objects. Therefore, the choice mapper will only be called by the question mapper when creating question objects. The question mapper will only be referred by Exam objects when lazy loading exam questions. The data mapper pattern will be discussed detailedly in section 6.2.

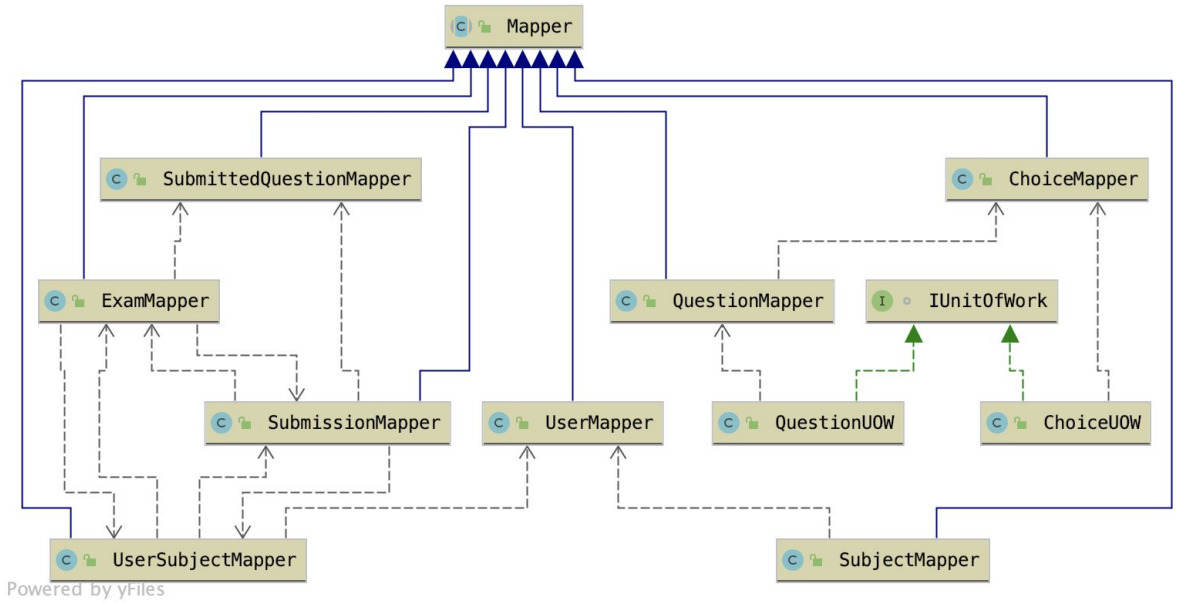

Figure 4.2.1 Dependencies between all data mappers

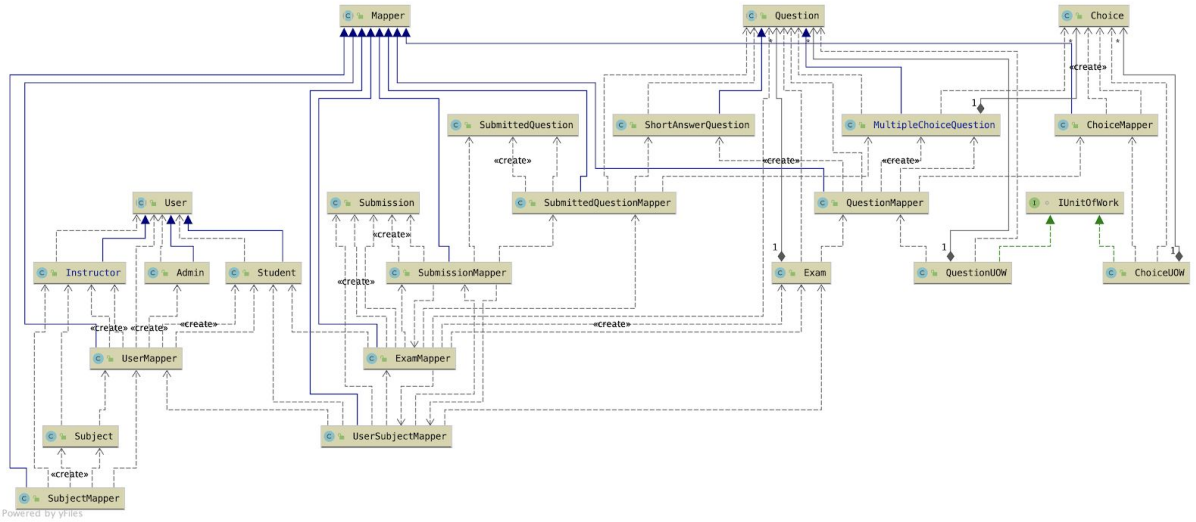

Figure 4.2.2 Dependencies between all data mappers and domain objects

For adding exam and editing exam, the question mapper and the choice mapper adopt the unit of work design pattern. The class diagram is shown in figure 4.2.3. More details about the use of the unit of work pattern will be introduced in Section 6.3.

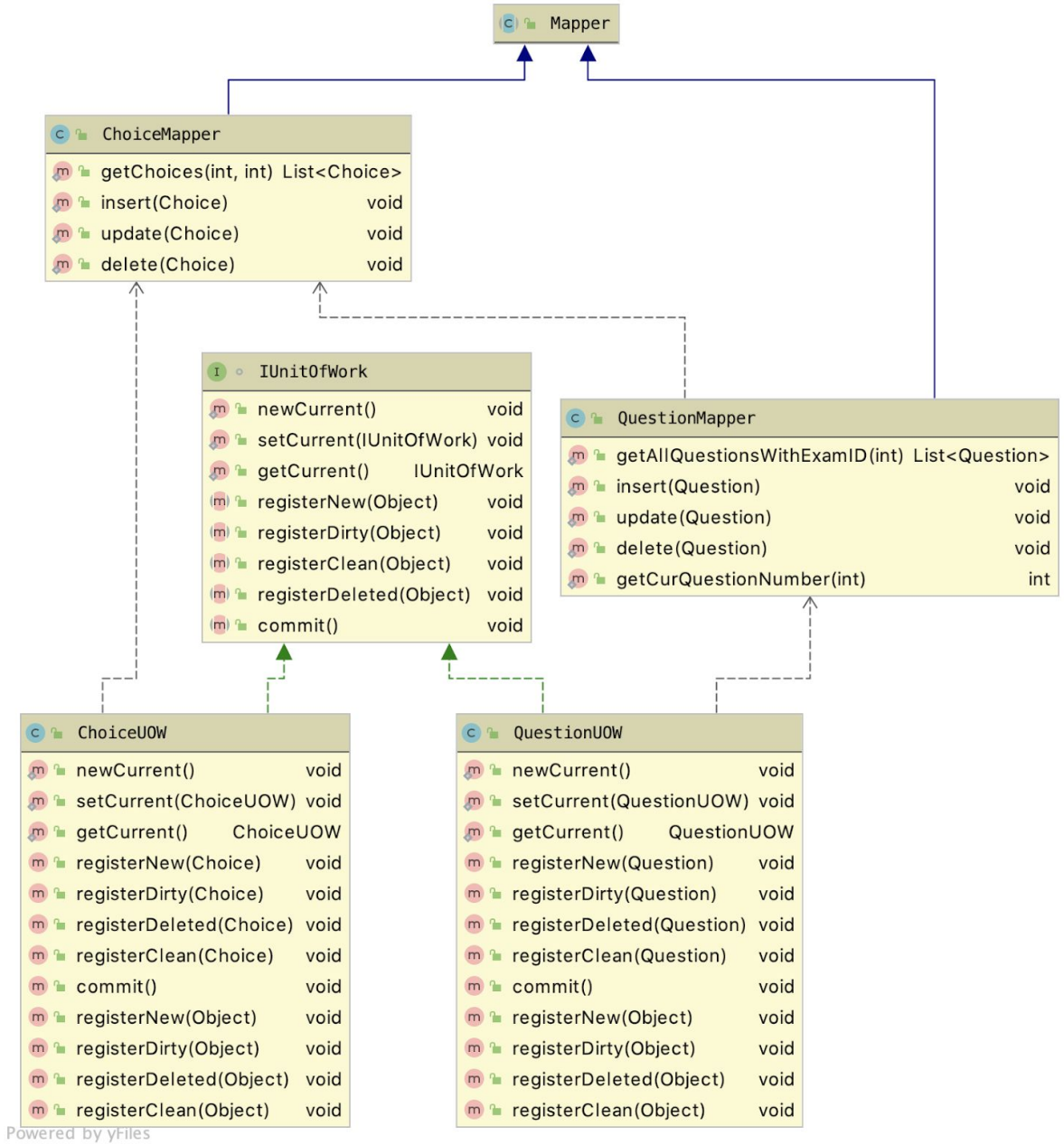

Figure 4.2.3 ChoiceMapper and QuestionMapper incorporating the unit of work pattern

Figure 4.2.4 shows the details of the other data mapper classes.

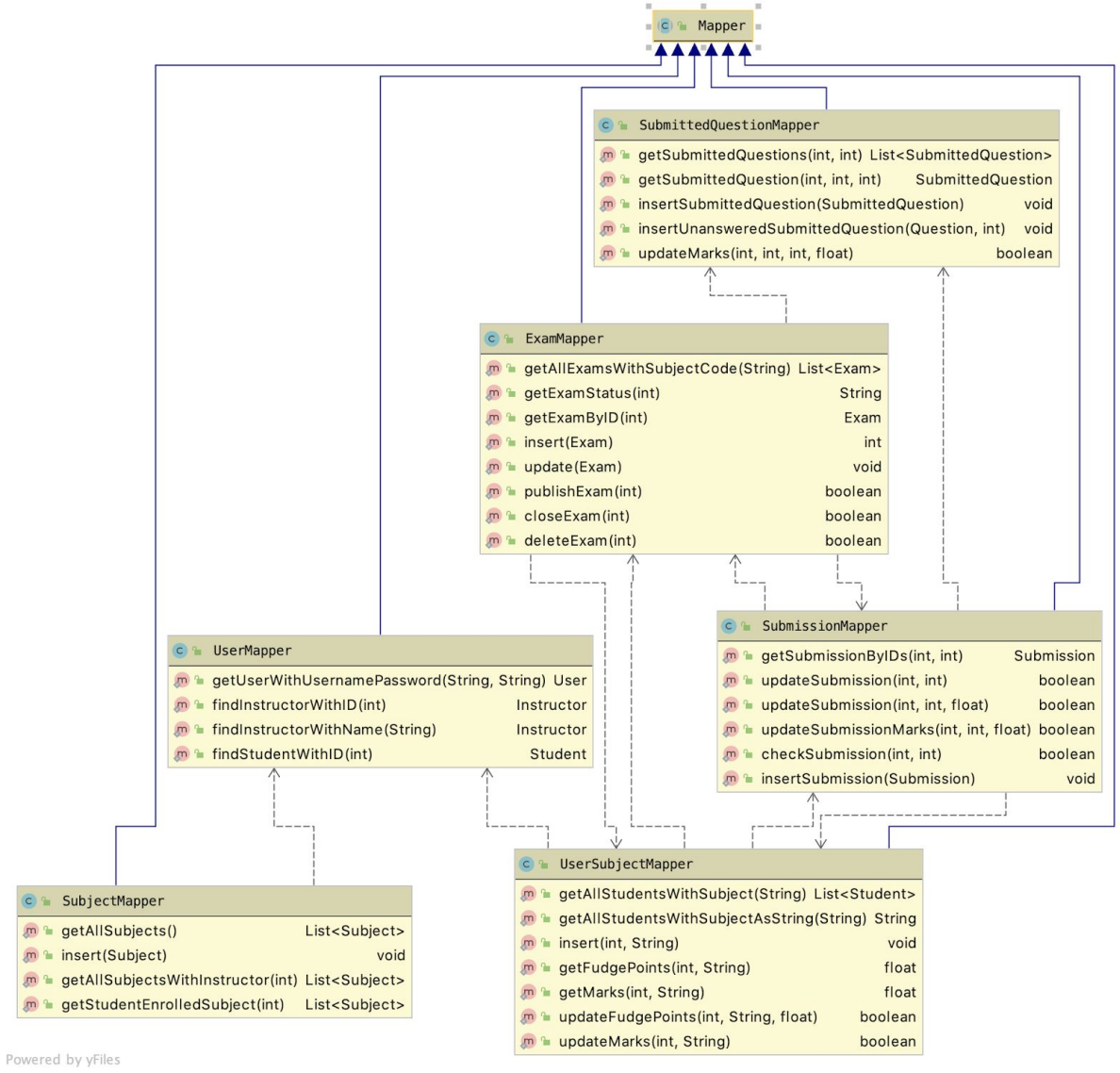

Figure 4.2.4 Class diagram of other data mappers

4.3 Database Connection

DBConnnection is the class that is used for getting connections with the PostgreSQL database and preparing query statements. Instances of DBConnection are created by data mappers when needed.

The following figure 4.3.1 will present the dependencies between DBConnection and all the mapper classes.

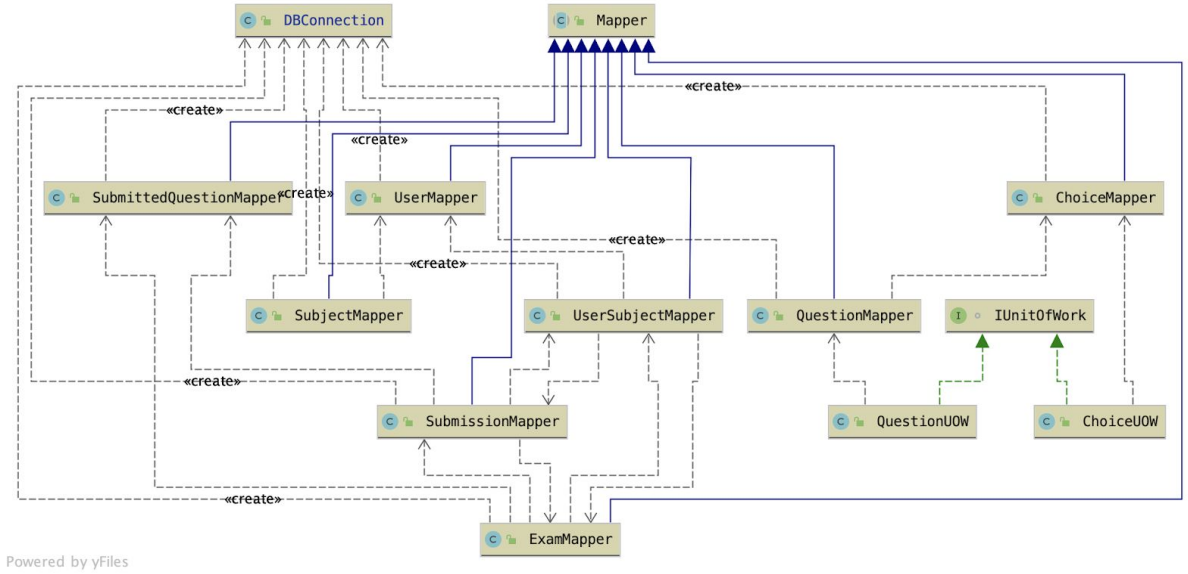

Figure 4.3.1 Dependencies between all data mappers and DBConnection

4.4 Controllers

Controllers are the classes that handle the HTTP GET and POST methods from the front end. They extend the HttpServlet class and handle GET and POST by overriding the methods doGet and doPost.

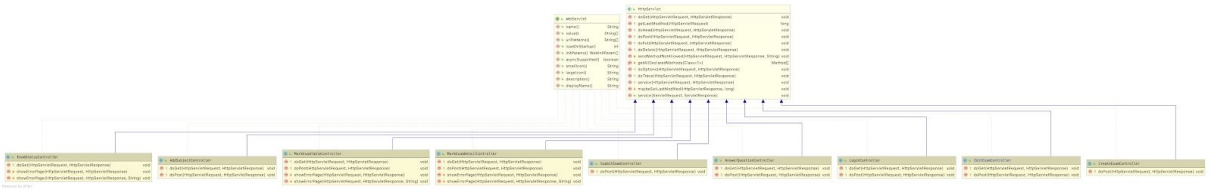

Figure 4.4.1 All controllers

Figure 4.4.2 shows the use of data mappers in controllers.

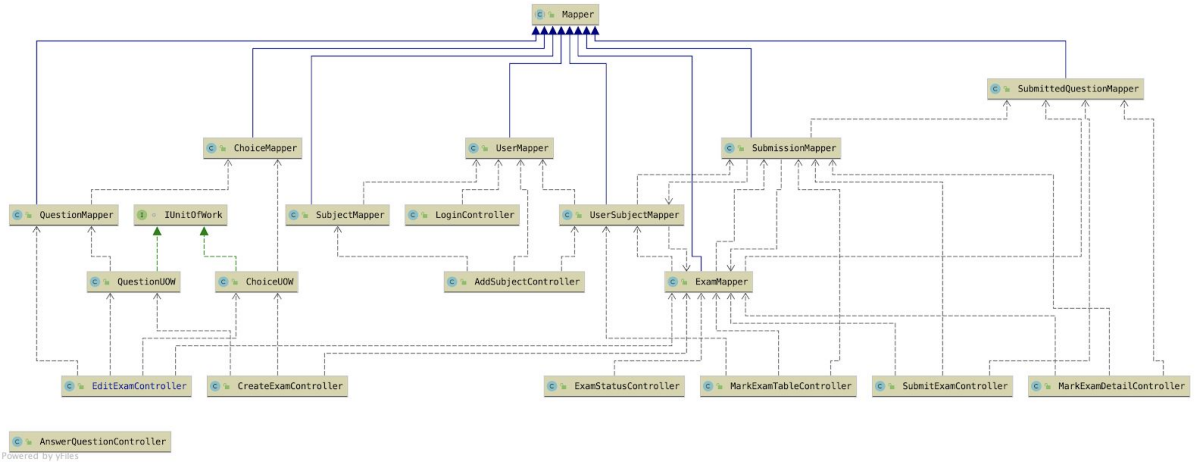

Figure 4.4.2 Dependencies between all controllers and all data mappers

5. System Architecture and Component Diagram

System Architecture

Figure 5.1 illustrates the architecture design of this system, which contains three layers, each of them has its corresponding role.

The presentation layer in charge of handling the interaction between users and the system. In this system, a web browser is used as the user interface, it displays the information and gets input data then forwards the requests to the rest of the system by calling business logic.

The domain layer is responsible for implementing the business logic of the system. Domain model and some behavioural design patterns like unit of work and lazy load are used in this layer.

The data source layer handles the data in the system and communicates with the database. Patterns like data mapper and some structural design patterns are adopted in this layer.

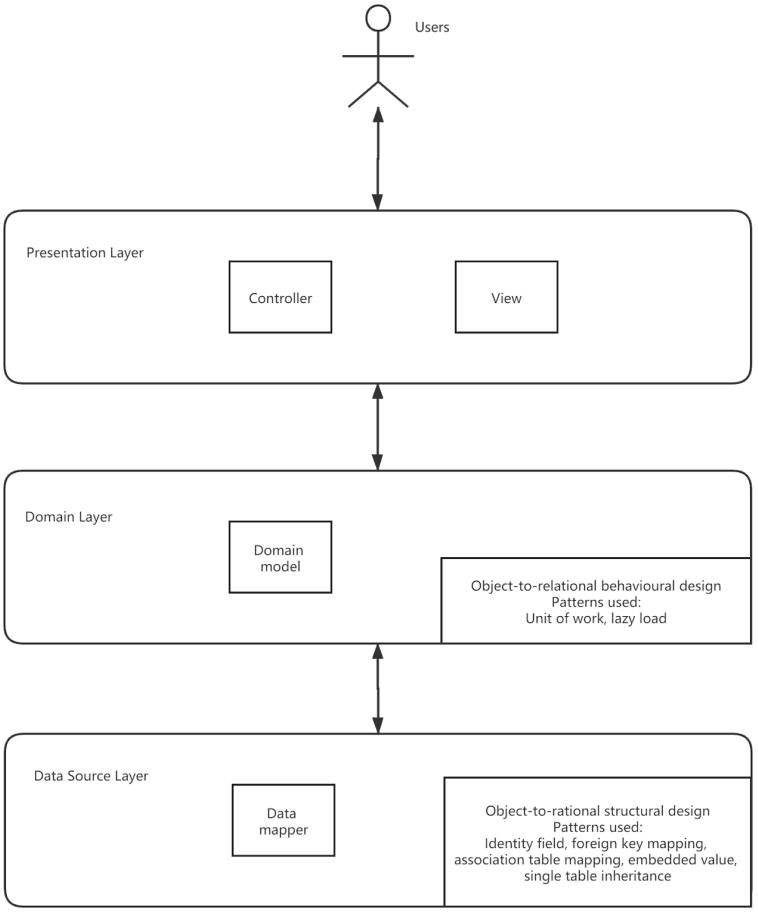

Figure 5.1 Architecture design of the system

Component Diagram

Figure 5.2 shows the component diagram of the online examination system in a high-level view. The system is deployed on Heroku.

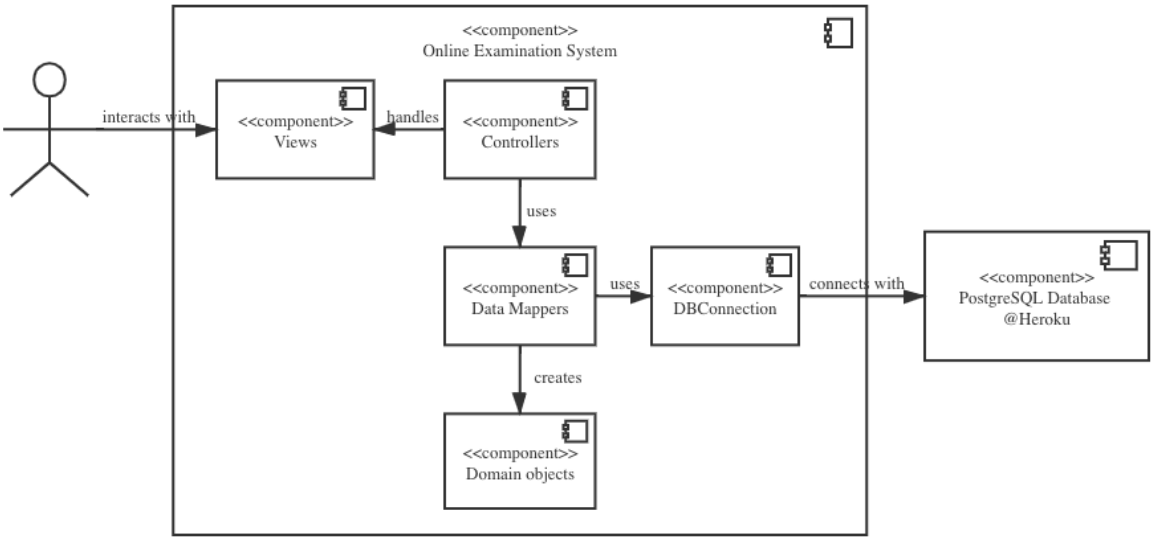

Figure 5.2 Component diagram

6. Design Patterns

6.1 Domain Model

Since the system is a large application and meets the requirements to be an enterprise application, to make it extensible and manageable, the domain model pattern is used to handle the domain logic. The domain logic reflects the business domain, which means the business rules of business objects are the reflections of the methods in the domain objects.

As the figure 4.1.1 in section 4.1 presents, the implemented domain model incorporates both data and behaviour, and each object takes its part of related logic.

6.2 Data Mapper

Data mapper is used to separate the domain objects from the underlying database. It enables objects and the database to keep independent from each other, as well as the mapper itself. The mappers play a role as a mediator, in this case, the domain layer is separated from the data source layer. A mapper class is used when called by the domain class to execute the query in the database as well as executes queries and use the results to create corresponding domain objects.

The pattern makes the domain object more flexible and the system more efficient for it decouples data from the database access then the domain layer does not need to worry about the database schema, and it allows the objects in the domain to be different from the database.

The data mapper outlines the other two patterns in the data source layer for it is more compatible with the domain model. In our design, the data mapper pattern is used consistently with the domain model pattern to achieve high re-use of both the database and the domain layer. Each class which is the root

of a domain hierarchy has a corresponding mapper class (Instructor class and Student class are child classes that extend the abstract User class, so they do not need mappers) as a mediator so that the domain classes do not need to handle the structured query.

In section 4, the usage of the data mapper pattern is presented in the form of class diagrams. Figure 4.2.2 in section 4.2 and Figure 4.3.1 in section 4.3 shows the class diagram of other data mappers and dependencies between all data mappers and DBConnection respectively.

In this section, the User table in the database will be used as a representative example as it is an abstract parent class in the domain, the other designing ideas are similar. The following diagrams will illustrate the class design and interactions between domain class and mapper class when implementing the User table in the database.

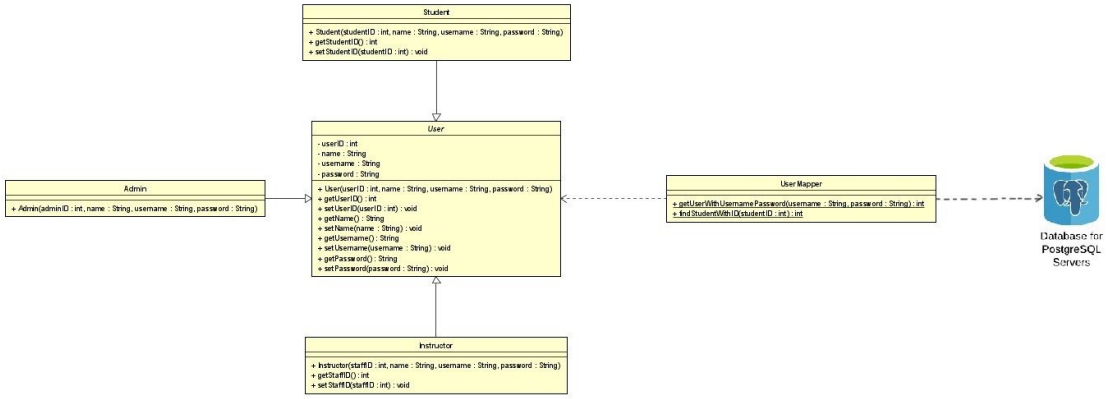

Figure 6.2.1 Class design that uses the data mapper pattern for the *User* table

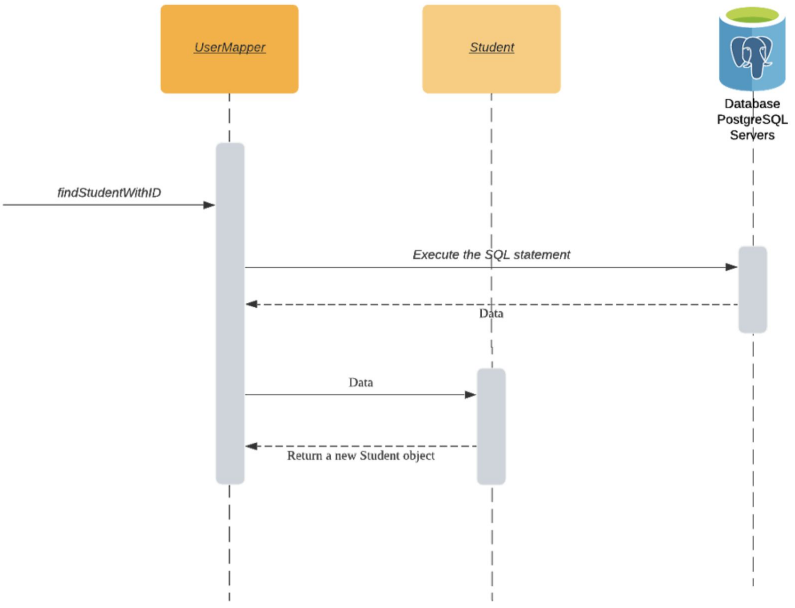

Figure 6.2.2 Interactions between domain class, mapper class and database

6.3 Unit of Work

The unit of work patterns keeps track of which domain objects are changed (or created). In our implementation, instead of using having a single UnitOfWork class and using factory or adapter patterns to allow the UnitOfWork class to work on different classes of object, I implemented two class specific UnitOfWork classes which implements the IUnitOfWork interface. Although the former offers a more elegant solution and better scalability, this implementation is much simpler and more straightforward.

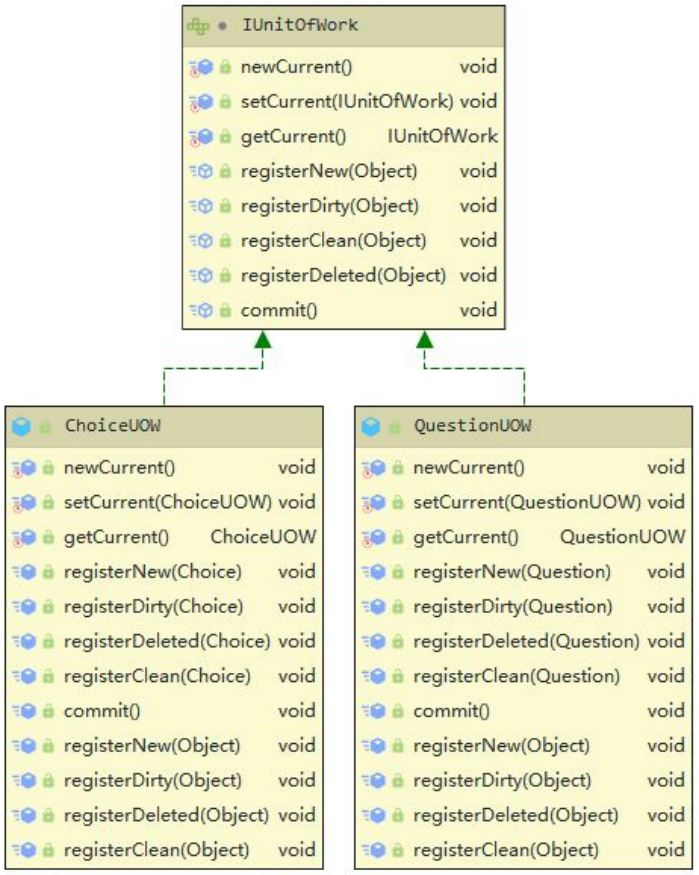

Figure 6.3.1 Unit of work design diagram

There are mainly two ways to register objects when they are changed: Caller registration and Object registration.

In our system, the caller registration is implemented. Although object registration suffers from the forgetfulness problem far less than caller registration, there are a few shortcomings:

1. When objects are read from the database, constructors are called to instantiate them. In this case, there is no need to register them as new. Although this can be resolved by having

different constructors for the object by using Polymorphism, it will further add to the complexity of the system.

2. When there is only a single operation, like a single update or delete, which happens very frequently in our system, using the unit of work pattern is a bit cumbersome.
3. There are other team members working on the same project and changing the domain object may result in unexpected behaviour in their part of the system.

Therefore, our system uses C*aller Registration*, that is, the calling code is responsible for registering the object.

Design Rationale

The UnitOfWork pattern is used in two parts of our system: when creating the exam, and when editing the exam. They are uses in these two parts for a couple of reasons:

1. These two parts involve inserting a set of newly added questions and choices into the database.

2. The editing question part involves updating the questions that are changed.

The UnitOfWork pattern provides the following benefits:

1. **Simplicity**: It simplifies the logic for inserting into and updating the database.

2. **Efficiency**: It reduces the number of updates made to the database. For example, if both the title, description and marks of a question are changed, instead of updating the database three times, now it is only done once.

3. **High cohesion**: all information regarding the change of objects is contained in one place.

4. **Atomicity**: It improves the atomicity of the transaction. All the changes to the database are made once at the time of the commit, or not at all. This will be extremely helpful when concurrency is factored into the system in Part 3.

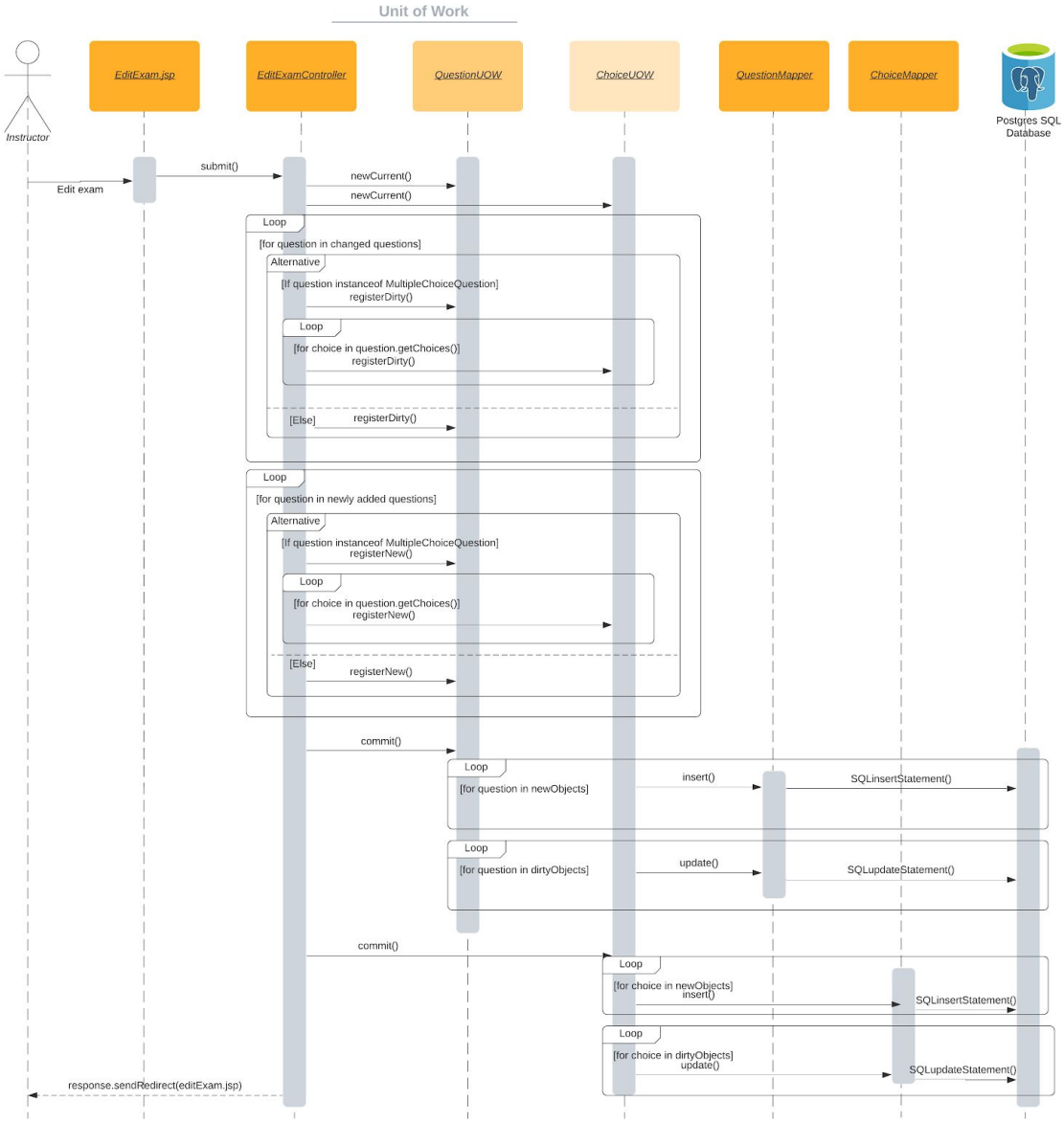

Figure 6.3.2 System Sequence Diagram for the use of UnitOfWork

6.4 Lazy Load

Lazy load pattern is the inverse of the unit of work pattern for it does not require the object to contain all of the data needed but only knows the method to get it, which reduces the amount of data read from the database by reading only what it needs. With the usage of using the lazy load, the loading of an object is delayed to the time when it is required, rather than initialized at the beginning.

In this system, the lazy load pattern is used when getting question lists in the exams when necessary. There are mainly four ways to implement the pattern, and ghost implementation is used in this system. The ghost implementation is suitable for this system because the Question objective is a relatively complicated object and has many fields, and ghost implementation can initialize all of the fields at the same time with a single query.

The question list contained in the exams is defined as a dummy object at the beginning which contains no data, and it would not be initialized until it is used when students take exams or instructors mark submissions which both require the question list to display in the pages.

The sequence diagram below is the example of how lazy load is used in getting the question list when a student is taking an exam, and it shows the ordered process of interactions. The studentAnswerQuestion.jsp would get the basic exam information like exam title and description from the Exam class, then when it requires additional information, like the questions to render the page, it uses the lazy load pattern and calls getQuestions() method in Exam class to load the question list.

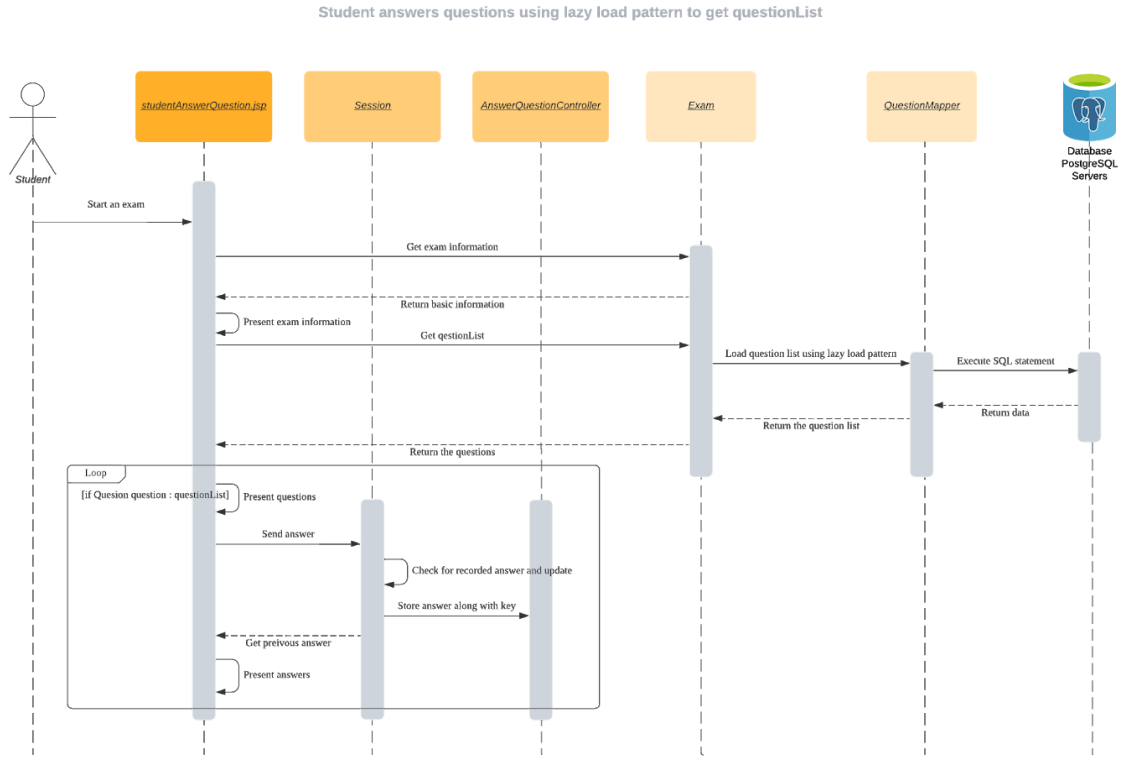

Figure 6.4.1 An example of a lazy load sequence diagram.

Design Rationale

The lazy load pattern contributes to improving the system's efficiency by initializing dummy objects that are empty and only reading the corresponding data when trying to access it. This significant benefit is due to some of the data is rarely used and it would not be accessed until needed.

However, the overuse of lazy load would increase the complexity of the system since implementing a lazy load pattern would add more methods and clauses. Furthermore, the pattern would cause confusion on the type that the object is created when dealing with the cases that involve inheritance, especially when using lazy initialization or ghost. Last but not least, an arisen problem is with ripple loading. When using the lazy load, one calling method may iterate over a collection of objects and they all need to be created when being accessed. If not being designed properly, it would waste time on the calling method. The decision of how many objects to load seems to be vital in this situation.

After comprehensive consideration, the lazy load pattern is used in some specified cases in the system. As a question list is part of an exam, but it is not a necessary field. In most of the pages, for example, the student view exam page, the question list is never sued. In these scenarios, initializing the question list seems to be redundant and meaningless, so the lazy load pattern is introduced to improve efficiency. Same as the figure 6.4.1 above shows, getting the question list used lazy load is also used when an instructor marks the exams in both detailed view and table view.
The use of lazy load pattern in this system improves the overall efficiency and performance because our design is reasonable. It is used only when it is truly needed, and it follows the structure of the database table so that it can get all the fields in a question object at the same time with one single query.

6.5 Identity Field

The identity field design pattern ensures that the domain objects map to the correct row in the database by including an identity field in the domain object. It therefore plays an important role in software that involves the process of reading rows from a database, modifying them, and writing them back to the database.

For the online exam system, this process is commonly used in the edit exam functionality. To ensure the update query can be executed and is updating the correct row, an integer value of the exam ID needs to be stored in each Exam object and assigned the correct value upon creation. In this scenario, the keys are meaningless serial values automatically generated by the database. It corresponds to a single database field and is a table-unique key.

Figure 6.5.1 illustrates the edit exam process in a sequence diagram. The EditExamController first gets an Exam object from ExamMapper, then modifies it using its mutators. After all modifications, it calls the update method in the ExamMapper and passes the Exam object to it as a parameter.

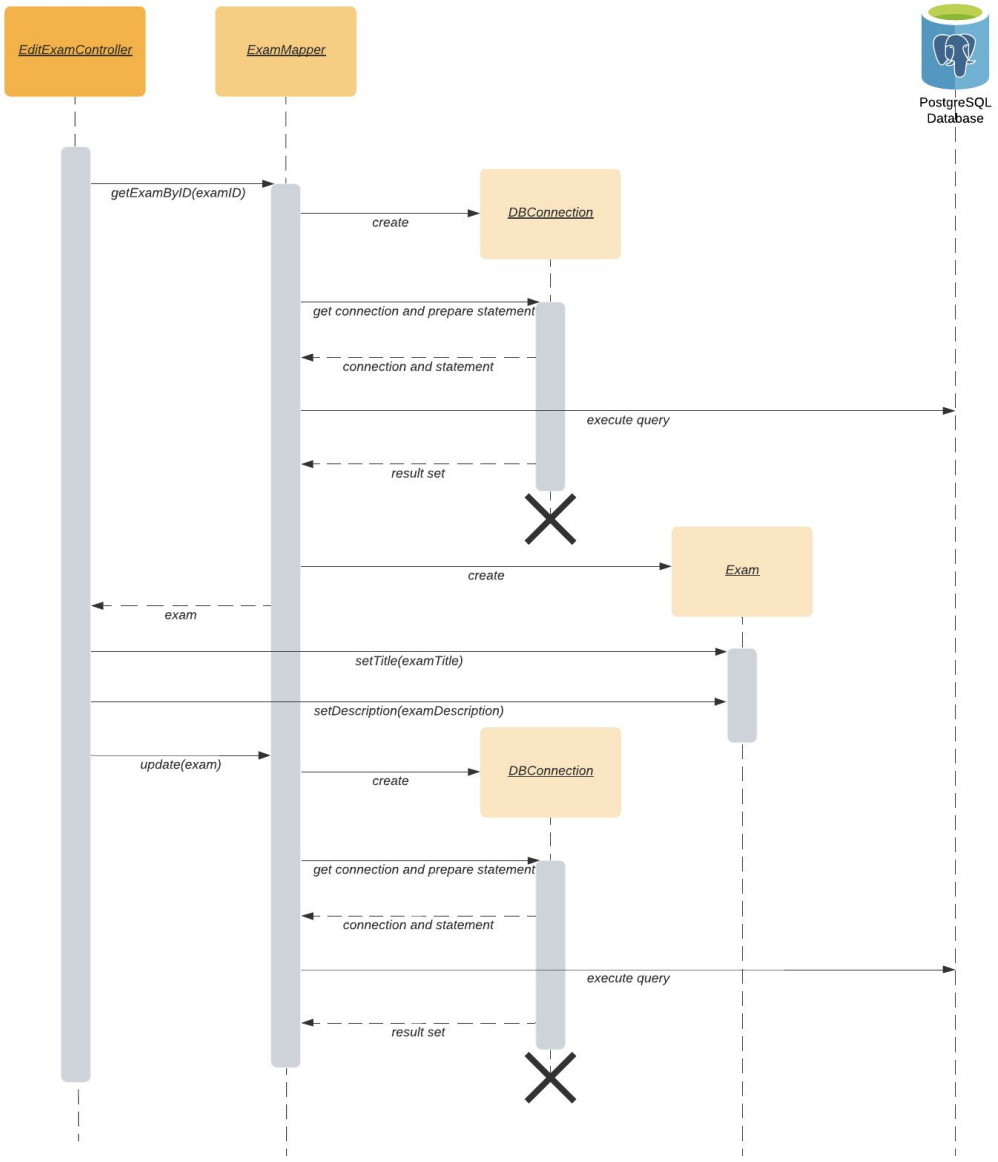

Figure 6.5.1 Sequence diagram of using identity field

6.6 Foreign Key Mapping

The application utilised foreign key mapping to handle a number of relationships between objects. One example is the one-to-many relationship between questions and choices. In the database schema, information of exam questions including its exam ID, question number, title, description and marks are stored in the Question table. For the multiple choice questions, their choices are stored in a separate Choice table. This design decision is made since not all questions have choices and the number of choices is not fixed. In the Choice table, the exam ID field and the question number field act as the foreign keys that refer to the corresponding question in the Question table.

In the domain layer of the software, when reading a question from the database, its choices are also read from the Choice table and stored in the Question object as a list. Since all situations require the choices of a multiple choice question to be displayed along with its other information, this approach rarely leads to an overhead. Applying this design pattern results in high simplicity as it eliminates the

necessity to explicitly call ChoiceMapper when getting choices of a question. Instead, the program can simply call the get choices method of the question object.

The following sequence diagram illustrates an example of getting questions from an Exam object.

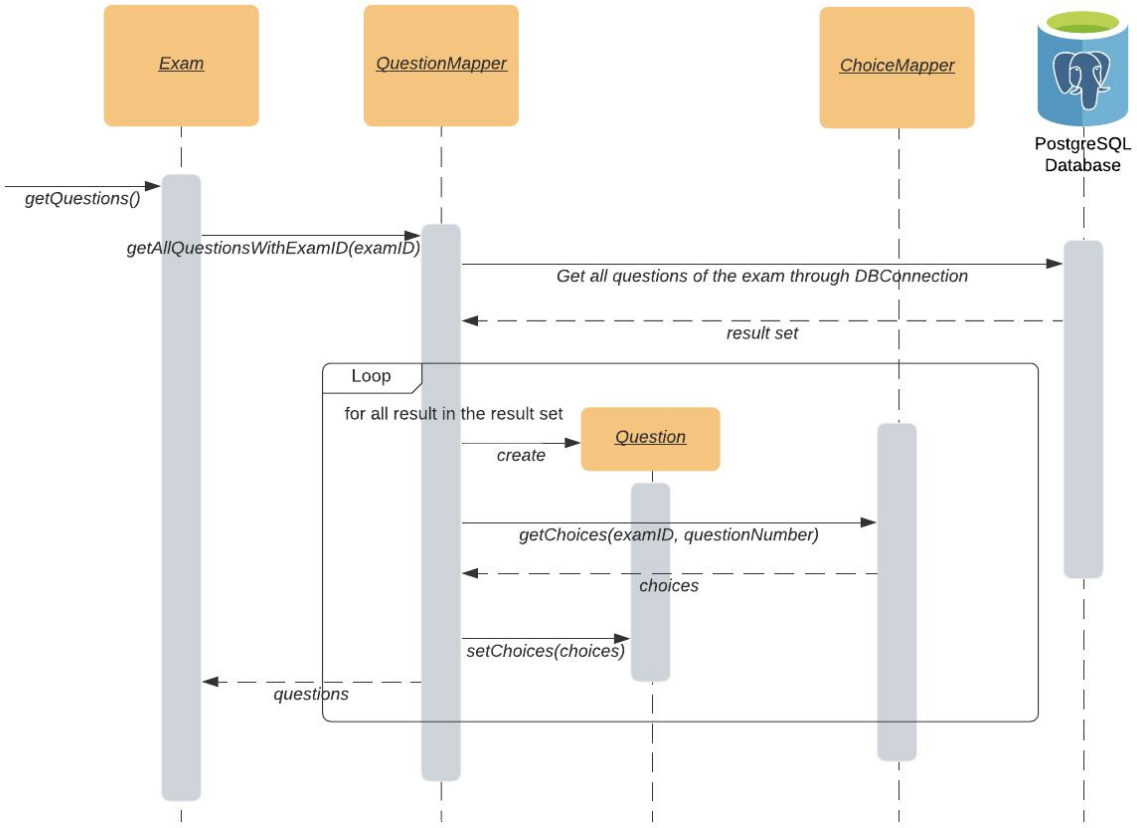

Figure 6.6.1 Sequence diagram of using foreign key mapping

6.7 Association Table Mapping

As the project described, a subject must have one or more instructors and one or more students, indicating that they have a many-to-many relationship. As the foreign key pattern has drawbacks in unable to handle the many-to-many relationship, association table mapping based on the identity field pattern is introduced.

In this system, a table called user_has_subject is created to represent their mapping relation. It records pairs of foreign keys, user ID and subject Code respectively, to indicate that the user has this subject. By telling the type of the user we can know that the user is an instructor and he/ she is the coordinator of this subject or he/ she is a student and has enrolled in this subject.

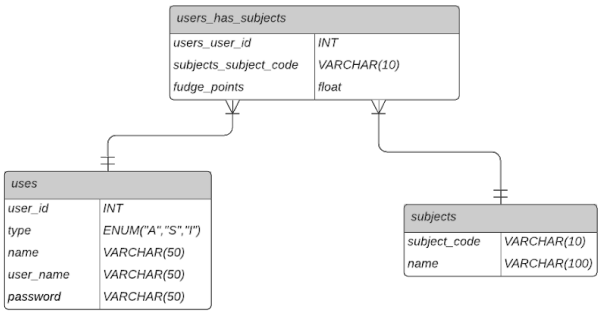

Figure 6.7.1 Database design using association table mapping

6.8 Embedded Value

In the system, it is required to store the total marks of all submissions and fudge points for each combination of students and the subjects they are enrolled in. Therefore, a table named student_subject_marks is designed to store this information, with the relationship ID field being the foreign key that refers to the user_has_subjects table, the association table between users and subjects table (figure n). However, this design would add complexity to the system. To simplify this, we decided to adopt the embedded value pattern and embed the marks and fudge points in the user_has_subjects table, as shown in figure m.

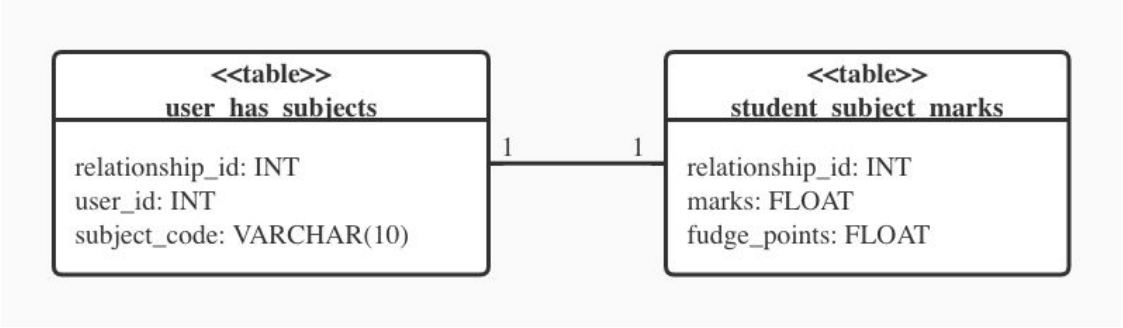

Figure 6.8.1 Original design

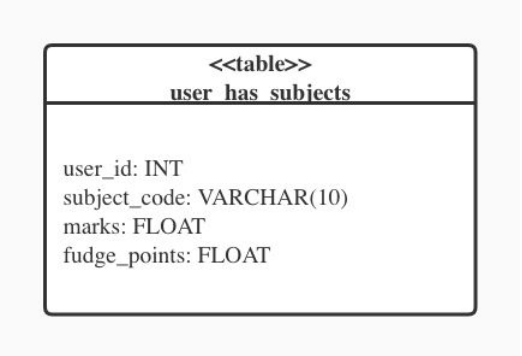

Figure 6.8.2 Storing marks and fudge points as embedded value

6.9 Single Table Inheritance

There are three places inheritance is used in the domain model, however, the class ShortAnswerQuestion does not have any attribute in it, so it is ignored in this section.

There are mainly three ways to implement an inheritance relationship in SQL databases:

1. Single table inheritance
2. Class table inheritance
3. Concrete table inheritance

Both of the inheritance relationships are represented in the database as single table inheritance.

The following sections will discuss respectively why single table inheritance is chosen.

6.9.1 User

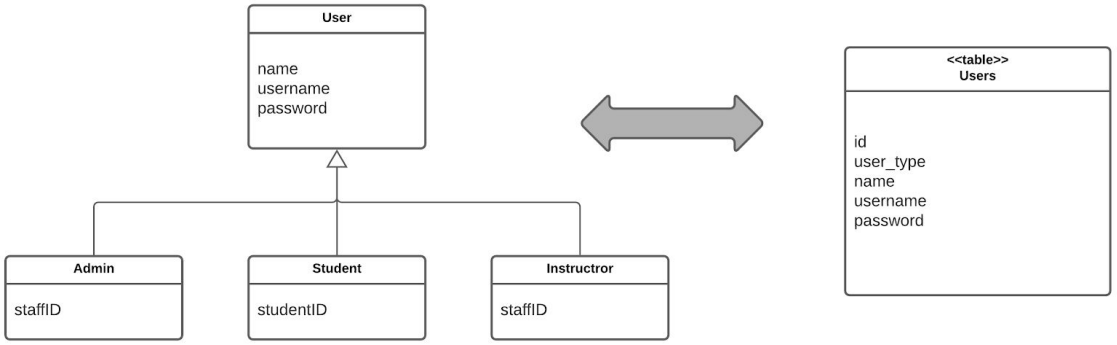

Figure 6.9.1 User, Admin, Student and Instructor classes with Users table

The *concrete table inheritance* is first disregarded due to the technical difficulty in maintaining unique keys between the tables.

There is not much of a point to using the *class table inheritance* as all three of the classes actually share the field ID. Even though these IDs may have different underlying meanings, there is a good chance that these IDs are unique across the system. Under this assumption, these three classes can be easily implemented into one table without introducing any new field that can be potentially NULL. Thus, the *single table inheritance* is chosen here.

If the assumption does not hold, a database specific ID can be added to the table to be used as the primary key and the respective student ID or staff ID for the user can be just used as an attribute.

6.9.2 Submitted Question

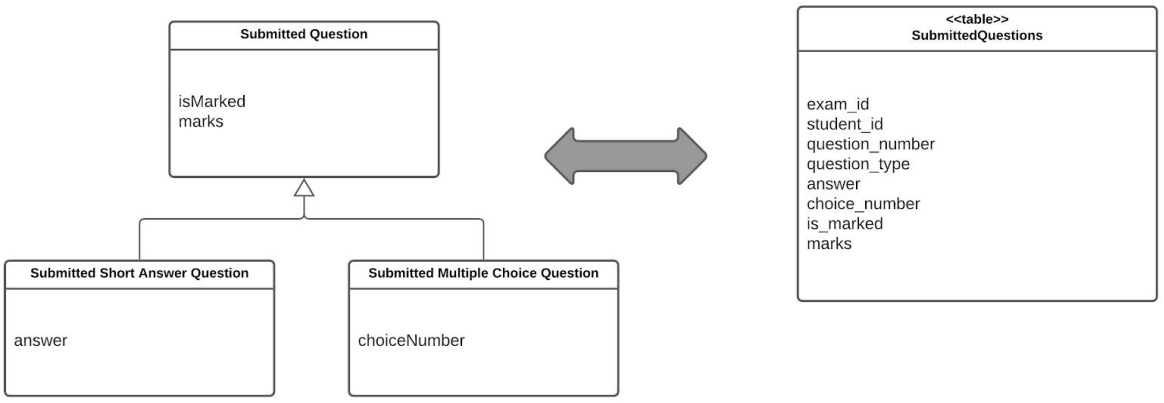

Figure 6.9.2 Submitted Question, Submitted Short Answer Question classes with Submitted Questions Table

The *concrete table inheritance* is disregarded here for the aforementioned reason.

In this case, both of the *single table inheritance* and *class table inheritance patterns* could be used. The class table inheritance is better in terms of table design. It does not contain any potential NULL field and this solves the issue of possible confusions as well as memory waste in the database. However, because this is a relatively simple case where there are only 2 child classes and each of the child classes only has one unique attribute.

Therefore, the *single table inheritance* is chosen mainly for two reasons:

1. **Simplicity**: One table holds all the data. No need to write two data mappers.
2. **No joins**: table joins are not required which again not only improves the performance but also simplifies the implementation.

7. Deployment to Heroku

Link to Heroku deployed app: swen90007-magic-pigeons.herokuapp.com

8. Test Data

8.1 User account data: <users>

Description: This table corresponds to the users table in the database which stores all information related to the users. Correct usernames and passwords need to be used to login into the system.				
user_id	**user_type**	**name**	**username**	**password**
0	ADMIN	Administrator	admin	admin
1	INSTRUCTOR	Eduardo Oliveira	eduardo	000000
2	INSTRUCTOR	Maria Rodriguez Read	maria	000000
713551	STUDENT	Jiayu Li	jiayul3	111111
904601	STUDENT	Simai Deng	simaid	111111
1049166	STUDENT	Yiran Wei	yirwei	111111

Table 8.1.1 Testing data for users

8.2 Subjects data: <subjects>

Description: This table corresponds to the subjects table in the database that stores the data of all subjects. Users of ADMIN type are allowed to add new entries to this table and add instructors to existing subjects.	
subject_code	**name**
SWEN90007	Software Design and Architecture
SWEN90009	Software Requirement Analysis

Table 8.2.1 Testing data for subjects

8.3 Association table of users and subjects: <users_has_subjects>

Description:
This table corresponds to the users_has_subjects table in the database which stores the subjects that each user is involved in (instructors teaching subjects or students enrolled in subjects). It also stores marks and fudge points for students as embedded values.

user_id	subject_code	marks	fudge_points
1	SWEN90007	-1 (default)	0 (default)
2	SWEN90007	-1 (default)	0 (default)
713551	SWEN90007	-1 (default)	0 (default)
904601	SWEN90007	-1 (default)	0 (default)
1049166	SWEN90007	-1 (default)	0 (default)
1	SWEN90009	-1 (default)	0 (default)
713551	SWEN90009	-1 (default)	0 (default)
904601	SWEN90009	-1 (default)	0 (default)
1049166	SWEN90009	-1 (default)	0 (default)

Table 8.3.1 Testing data for users and subjects

8.4 Exams data: <exams>

Description:
This table illustrates the information regarding the exams. Note that only subject_code, exam_title and exam_status are the rows that are actually stored in the exams table in the database. Other rows are for explanatory purpose only. Exam description is not shown in this table for simplicity.

subject_code	exam_title	exam_status	Number of Questions
SWEN90007	Week 3 Quiz	Closed	2
SWEN90007	Week 5 Quiz	Published	1
SWEN90007	Final exam	Unpublished	1
SWEN90009	Mid-Sem exam	Published	3
SWEN90009	Final exam	Unpublished	1

Table 8.4.1 Testing data for exams

As per requirements, students can see the exams with published or closed status, and can only answer published exams. Instructors can edit any unpublished exam, publish any unpublished exam, close any published exam, mark any closed exam, and delete any exam that has no submission.

The default state of newly created exams is unpublished.

8.5 Submission data: <submission>

Description:
This table illustrates the information regarding the exam submissions. Note that some rows are for explanatory purpose only and are not actually stored in the submissions table in the database.

exam_id	Exam Title	submission_time	is_marked	marks	fudge_points
1	Week 3 Quiz	2020-09-28 01:00:00	false (default)	-1 (default)	0 (default)
1	Week 3 Quiz	2020-09-28 01:00:00	false (default)	-1 (default)	0 (default)
1	Week 3 Quiz	2020-09-28 01:00:00	false (default)	-1 (default)	0 (default)

Table 8.5.1 Testing data for submissions

For more details about the data, please refer to the SQL script in Appendix A.

9. Git Release Tag

Git repository: https://github.com/hedgehog7453/SWEN90007_2020_MagicPigeons

Git release tag: SWEN90007_2020_Part2_MagicPigeons

Appendix A. Database implementation

```
-------------------------------------------------------------------------------
--                                subjects                                   --
-------------------------------------------------------------------------------
CREATE TABLE subjects (
   subject_code VARCHAR(20) NOT NULL UNIQUE,
   name VARCHAR(100) NOT NULL,
   PRIMARY KEY (subject_code)
);

INSERT INTO subjects VALUES ('SWEN90007', 'Software Design and Architecture');
INSERT INTO subjects VALUES ('SWEN90009', 'Software Requirement Analysis');

-------------------------------------------------------------------------------
--                                  users                                    --
-------------------------------------------------------------------------------
CREATE TYPE user_type AS ENUM ('ADMIN', 'STUDENT', 'INSTRUCTOR');

CREATE TABLE users (
   user_id INT NOT NULL UNIQUE,
   type user_type NOT NULL,
   name VARCHAR(50) NOT NULL,
   username VARCHAR(50) NOT NULL UNIQUE,
   password VARCHAR(50) NOT NULL,
   PRIMARY KEY (user_id)
);

INSERT INTO users VALUES (000000, 'ADMIN', 'Administrator', 'admin', 'admin');
INSERT INTO users VALUES (000001, 'INSTRUCTOR', 'Eduardo Oliveira', 'eduardo',
'000000');
INSERT INTO users VALUES (000002, 'INSTRUCTOR', 'Maria Rodriguez Read', 'maria',
'000000');
INSERT INTO users VALUES (904601, 'STUDENT', 'Simai Deng', 'simaid', '111111');
INSERT INTO users VALUES (713551, 'STUDENT', 'Jiayu Li', 'jiayul3', '111111');
INSERT INTO users VALUES (1049166, 'STUDENT', 'Yiran Wei', 'yirwei', '111111');

-------------------------------------------------------------------------------
--                           users_has_subjects                              --
-------------------------------------------------------------------------------
CREATE TABLE users_has_subjects (
```

```
    user_id INT NOT NULL REFERENCES users(user_id),
    subject_code VARCHAR(20) NOT NULL REFERENCES subjects(subject_code),
    marks FLOAT DEFAULT -1,
    fudge_points FLOAT DEFAULT 0
);

INSERT INTO users_has_subjects VALUES(000001, 'SWEN90007', DEFAULT, DEFAULT);
INSERT INTO users_has_subjects VALUES(000001, 'SWEN90009', DEFAULT, DEFAULT);
INSERT INTO users_has_subjects VALUES(000002, 'SWEN90007', DEFAULT, DEFAULT);
INSERT INTO users_has_subjects VALUES(904601, 'SWEN90007', DEFAULT, DEFAULT);
INSERT INTO users_has_subjects VALUES(904601, 'SWEN90009', DEFAULT, DEFAULT);
INSERT INTO users_has_subjects VALUES(713551, 'SWEN90007', DEFAULT, DEFAULT);
INSERT INTO users_has_subjects VALUES(1049166, 'SWEN90007', DEFAULT, DEFAULT);
INSERT INTO users_has_subjects VALUES(713551, 'SWEN90009', DEFAULT, DEFAULT);
INSERT INTO users_has_subjects VALUES(1049166, 'SWEN90009', DEFAULT, DEFAULT);

-------------------------------------------------------------------------------
--                                  exams                                    --
-------------------------------------------------------------------------------
CREATE TYPE exam_status AS ENUM ('UNPUBLISHED', 'PUBLISHED', 'CLOSED');

CREATE TABLE exams (
    exam_id SERIAL NOT NULL UNIQUE,
    subject_code VARCHAR(20) REFERENCES subjects(subject_code),
    title VARCHAR(100) NOT NULL,
    description VARCHAR(200) NOT NULL,
    status exam_status NOT NULL DEFAULT 'UNPUBLISHED',
    PRIMARY KEY (exam_id)
);

INSERT INTO exams VALUES (DEFAULT, 'SWEN90007', 'Week 3 Quiz', 'A quiz about data
source layer', 'CLOSED');
INSERT INTO exams VALUES (DEFAULT, 'SWEN90007', 'Week 5 Quiz', 'A quiz about object
to relational structural patterns', 'PUBLISHED');
INSERT INTO exams VALUES (DEFAULT, 'SWEN90007', 'Final exam', '2020S2 Final exam of
Software Design and Architecture (60% of total)', DEFAULT);
INSERT INTO exams VALUES (DEFAULT, 'SWEN90009', 'Mid-Sem exam', '2020S1 Mid
semester exam', 'PUBLISHED');
INSERT INTO exams VALUES (DEFAULT, 'SWEN90009', 'Final exam', '2020S1 Final exam of
Software Requirement Analysis', DEFAULT);
```

```
-------------------------------------------------------------------------
--                              questions                              --
-------------------------------------------------------------------------
CREATE TYPE question_type AS ENUM ('MULTIPLE_CHOICE', 'SHORT_ANSWER');

CREATE TABLE questions (
   exam_id INT REFERENCES exams(exam_id) ON DELETE CASCADE,
   question_number INT NOT NULL,
   question_type question_type NOT NULL,
   title VARCHAR(45) NOT NULL,
   description VARCHAR(500) NOT NULL,
   marks FLOAT NOT NULL,
   PRIMARY KEY (exam_id, question_number)
);

INSERT INTO questions VALUES (1, 1, 'SHORT_ANSWER', 'Question 1', 'What is the
object that wraps a row in a DB table or view, encapsulates the DB access, and adds
domain logic on that data?', 50);
INSERT INTO questions VALUES (1, 2, 'MULTIPLE_CHOICE', 'Question 2', 'What is the
layer of software that separates the in-memory objects from the database?', 50);

INSERT INTO questions VALUES (2, 1, 'SHORT_ANSWER', 'Question 1', 'How to
structurally map our domain objects to a relational database?', 100);

INSERT INTO questions VALUES (3, 1, 'SHORT_ANSWER', 'Question 1', 'What does unit
of work do?', 100);

INSERT INTO questions VALUES (4, 1, 'SHORT_ANSWER', 'Question 1', 'What is software
engineering?', 20);
INSERT INTO questions VALUES (4, 2, 'SHORT_ANSWER', 'Question 2', 'What is software
requirements analysis?', 50);
INSERT INTO questions VALUES (4, 3, 'MULTIPLE_CHOICE', 'Multiple Choice 1', 'Choose
the WRONG statement.', 30);

INSERT INTO questions VALUES (5, 1, 'SHORT_ANSWER', 'Question 1', 'What is
requirement elicitation?', 100);

-------------------------------------------------------------------------
--                               choices                               --
-------------------------------------------------------------------------
CREATE TABLE choices (
   exam_id INT,
```

```
    question_number INT,
    choice_number INT NOT NULL,
    choice_description VARCHAR(200) NOT NULL,
    PRIMARY KEY (exam_id, question_number, choice_number)
);

INSERT INTO choices VALUES (1, 2, 1, 'Table data gateway');
INSERT INTO choices VALUES (1, 2, 2, 'Row data gateway');
INSERT INTO choices VALUES (1, 2, 3, 'Active record');
INSERT INTO choices VALUES (1, 2, 4, 'Data mapper');

INSERT INTO choices VALUES (4, 3, 1, 'Software engineering is meaningless.');
INSERT INTO choices VALUES (4, 3, 2, 'Software engineering is the systematic
application of engineering approaches to the development of software.');

-------------------------------------------------------------------------------
--                                submissions                                --
-------------------------------------------------------------------------------
CREATE TABLE submissions (
    exam_id INT REFERENCES exams(exam_id),
    user_id INT REFERENCES users(user_id),
    submission_time TIMESTAMP NOT NULL,
    is_marked BOOLEAN NOT NULL DEFAULT FALSE,
    marks FLOAT DEFAULT -1,
    fudge_points FLOAT DEFAULT 0,
    PRIMARY KEY (exam_id, user_id)
);

INSERT INTO submissions VALUES (1, 904601, '2020-09-28 01:00:00', DEFAULT, DEFAULT,
DEFAULT);
INSERT INTO submissions VALUES (1, 713551, '2020-09-28 01:00:00', DEFAULT, DEFAULT,
DEFAULT);
INSERT INTO submissions VALUES (1, 1049166, '2020-09-28 01:00:00', DEFAULT,
DEFAULT, DEFAULT);

-------------------------------------------------------------------------------
--                            submitted questions                            --
-------------------------------------------------------------------------------
CREATE TABLE submitted_questions (
    exam_id SERIAL REFERENCES exams(exam_id),
    user_id INT REFERENCES users(user_id),
```

```
    question_number SMALLINT NOT NULL,
    question_type question_type NOT NULL,
    choice_number SMALLINT DEFAULT null,
    short_answer VARCHAR(500) DEFAULT null,
    is_marked BOOLEAN NOT NULL DEFAULT FALSE,
    marks FLOAT DEFAULT -1,
    PRIMARY KEY (exam_id, user_id, question_number)
);

INSERT INTO submitted_questions VALUES (1, 904601, 1, 'SHORT_ANSWER', DEFAULT,
'Active record', DEFAULT, DEFAULT);
INSERT INTO submitted_questions VALUES (1, 904601, 2, 'MULTIPLE_CHOICE', 3,
DEFAULT, DEFAULT, DEFAULT);
INSERT INTO submitted_questions VALUES (1, 713551, 1, 'SHORT_ANSWER', DEFAULT,
'Active record', DEFAULT, DEFAULT);
INSERT INTO submitted_questions VALUES (1, 713551, 2, 'MULTIPLE_CHOICE', 2,
DEFAULT, DEFAULT, DEFAULT);
INSERT INTO submitted_questions VALUES (1, 1049166, 1, 'SHORT_ANSWER', DEFAULT,
'Data mapper', DEFAULT, DEFAULT);
INSERT INTO submitted_questions VALUES (1, 1049166, 2, 'MULTIPLE_CHOICE', 4,
DEFAULT, DEFAULT, DEFAULT);
```