Architecture Document

SWEN90007 Software Design and Architecture
Semester 2, 2020

Team – Super Girls

1049113 Jianjing Yao (jianjingy)
jianjingy@student.unimelb.edu.au

1054195 Lu Wang (lw2)
lu.wang4@student.unimelb.edu.au

1095044 XuelingLiu (xuelingl1)
xuelingl1@student.unimelb.edu.au

Revision History

Date	Version	Description	Author
23/09/2020	01.00-D01	Design the content of this document	Jianjing Yao
24/09/2020	01.00-D02	Add Section 1 details to the document	Jianjing Yao
25/09/2020	01.00-D03	Add Section 2 details to the document	Jianjing Yao
26/09/2020	01.00-D04	Add Section 3 basic structure	Jianjing Yao
27/09/2020	01.00-D05	Add class diagram and sequence diagram	Jianjing Yao Xueling Liu
28/09/2020	01.00	Add Section 3 details First version of the document Review and discuss details in Section 1-3	Jianjing Yao Xueling Liu Lu Wang
29/09/2020	02.00-D01	Add references to the document	Jianjing Yao
30/09/2020	02.00-D02	Adjust class diagram and sequence diagram	Xueling Liu Jianjing Yao
01/10/2020	02.00-D03	Adjust details in section 1-3	Xueling Liu Jianjing Yao
02/10/2020	02.00	Adjust details in section 4-5 Review and test the system	Xueling Liu Jianjing Yao Lu Wang
03/10/2020	03.00-D01	Check grammar and adjust details Review the document and add git tag	Xueling Liu Jianjing Yao Lu Wang
04/10/2020	03.00	Final version of this document	Xueling Liu Jianjing Yao Lu Wang

Table of Contents

1. Introduction

1.1 Project Overview

This project is about an online examination application. The aim is to manage the exams online from the perspective of students and instructors. Based on the requirements given by the teaching team of SWEN90007, students can take exams of associated subjects and check results. Instructors can manage exams, including creating, editing, publishing and marking exams.

This document is designed for provides a comprehensive overview of the architecture of this project, which serves as a communication medium between the software architect and the team members regarding architecturally significant decisions which have been made on the project.

In this document, target users and conventions are introduced in section 1. All design patterns are listed in section 2. Architectural representation (4+1 View) is introduced in section 3. User manual which guides the end users to access the system is shown in section 4, followed by the appendix including Github link, tag and Heroku link.

1.2 Target Users

This document is designed for team Super Girls, teaching team of SWEN90007, and potential end users who may use this online exam application in the learning management system (LMS).

1.3 Conventions, terms and abbreviations

This section explains the concept of some important terms that will be used throughout this document. These terms are detailed alphabetically in the following table.

Term	Description
Architectural Pattern	An architectural pattern is a general, reusable solution to a commonly occurring problem in software architecture. A basic set of architectural patterns is important for the implementation of the project. [1]
Domain Logic	Domain logic is the part of the program that encodes the real-world business rules that determine how data can be created, stored, and changed. It is contrasted with the lower-level details of managing a database or displaying the user interface. [2]
Data Source	Data source allows elegant querying of the data-source layer from the domain layer without exposing the implementation details of the data-source layer. [3]
Architectural Representation	Architectural representation describes the adopted model in the system. "4+1" framework is always used as a base. [4]
4+1 View	Logical view, process view, development view, physical view and scenario. [5]
User Manual	User manual is intended to tell people how to use the system in a correct way. [6]

2. Architectural Patterns

2.1 Domain Logic

There are three patterns of domain logic, namely transaction script, domain model and table module. For this project, we apply the domain model to describe the domain logic (business logic) in the system. Figure 1 shows the domain model of our system.

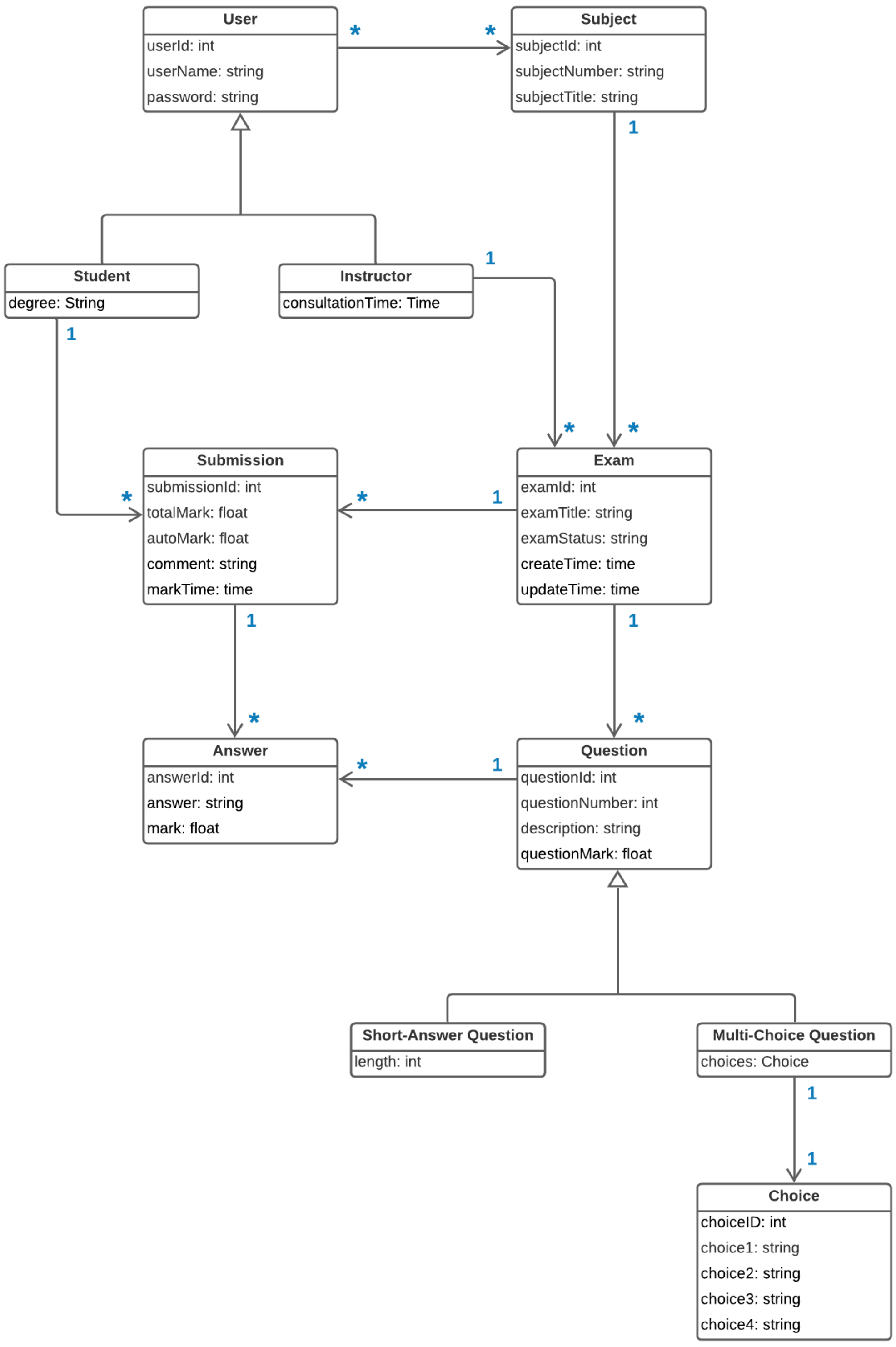

Figure 1 Domain Model

Business logic can be very complex in a software system. Rules and logic describe many different cases and behaviours, and the objects in domain model are designed to work with the complexity. The domain model creates a network of interconnected objects, where each object represents some meaningful individual [7]. Domain model guides us to create a Java package named "domain", and each object is designed as a Java class within that package. Table 1 shows the description of each object.

Object	Description
User	User is a general type in the system, which is inherited by Student and Instructor.
Student	Student is one role of the system. They can take the exams and see the results of their submissions.
Instructor	Instructor is another role in the system. They can create, edit exams and mark submissions.
Subject	Subjects can be associated to students and instructors, which means subjects can be enrolled by students and can be taught by instructors.
Exam	One exam is always associated to a particular subject, and one subject can have more than one exam.
Question	Question is unit located in each exam. The sum of the assigned mark (e.g. 50, 50) of all questions (e.g. q1, q2) in an exam (e.g. exam E1) should be the total mark (e.g. 100).
Short-Answer Question	Short-Answer Question is a type of questions. Students will be given a question description, and they need to input some sentences or paragraphs to answer the question. A length can be applied in this type of question because answers' length should be limited.
Multiple-Choice Question	Multiple-Choice Question is another type of questions. Students will be given a question description and a set of choices, and they need to select a choice among the set. We assume the maximum number of choices is four. And instructors need to add at least two choice when they create the exam.
Choice	Choice means a set of choices of a particular multiple-choice question.
Submission	When students submit an exam, they create a submission. As students can only take an exam once, the submission of a student for a particular exam cannot be duplicated.
Answer	Answer is a unit corresponding to each question in each submission. Taking the example in Question above, Student A and B submits exam E1, then we have a1, a2 for q1, q2 of Student A, and a3, a4 for q1, q2 of Student B.

Table 1 Objects in Domain Model

2.2 Object to relational structural design

It is vital to think about how we can map our domain objects into a relational database. Object to relational structural design helps us to design the database in a reasonable way.

2.2.1 Identity field

In a relational database, rows are distinguished from each other by their keys. In programming languages, objects or data structures are distinguished using memory addresses or object identities. When in-memory objects correspond to a row in a database, we need a method for ensuring that the correct objects are written back to the correct rows. [7]

The identity field pattern specifies a method such that, for any object that corresponds to a row, the object explicitly stores the primary key to the corresponding row, and all we need to do is to store the primary key of the relational database table in the object's fields. [3]

As you can see in our domain model, objects User, Subject, Exam, Question, Submission, Answer and Choice have an ID as the first attribute. Base on this, we create a class called "DomainObject" who has a field called "ID". Every object who needs to have an ID in domain model will inherit this class. For the choice of the primary key, we choose to use integer starting from 1 and increase by one when a new row is added.

2.2.2 Foreign key mapping

Objects contain references to other objects in the domain model, and often it is necessary for these references to persist in the database. The foreign key mapping pattern aims to solve this problem by using the identity field pattern. For example, if an object, o1 refers to another object o2, and this association should persist in the database, then in the database schema, map the association using a foreign key. In other words, the row corresponding to object o1 should have a foreign key field that refers to the corresponding row for object o2. [3]

Foreign key mapping is mostly used in 1-to-many relationship. According to our domain model, we have 1-to-many relationship in the following situations shown in Table 2. The specific database scheme is associated with each situation.

Situation	1 Side	Many Side	Database Scheme
1	Student	Submission	In "Submission" table, there is a foreign key called "studentID", which means the submission is submitted by a student.
2	Instructor	Exam	In "Exam" table, there is a foreign key called "instructorID", which means the exam is created by an instructor.
3	Subject	Exam	In "Exam" table, there is a foreign key called "subjectID", which means the exam belongs to a subject.
4	Exam	Submission	In "Submission" table, there is a foreign key called "examID", which means the submission is corresponding to an exam.

5	Exam	Question	In "Question" table, there is a foreign key called "examID", which means the question belongs to an exam.
6	Question	Answer	In "Answer" table, there is a foreign key called "questionID", which means the answer is corresponding to a question.
7	Submission	Answer	In "Answer" table, there is also a foreign key called "submissionID", which means the answer belongs to a submission.

Table 2 Foreign key mapping

2.2.3 Association table mapping

For a one-to-many mapping, we can use a foreign key for the single-valued end of the association. However, for a many-to-many relationship, there is no single-valued end, so this will not suffice. The association table mapping pattern uses a strategy from relational database design, which is to create a new table to represent the mapping. The guideline is to save an association as a table with foreign keys to the tables that are linked by the association. [3]

In our project, we have a many-to-many relationship, namely User to Subject. As mentioned above, we should create a new table in the database to handle this relationship. Thus, we create a new table called "UserAndSubject" which records the association relationship between User and Subject.

There is an example shown in Table 3. Assuming that there are four users in User table, two of them are students (S1, S2), and others are instructors (I1, I2). There are four subjects in Subject table. Then the UserAndSubject table means that S1 enrols the subject Software Patterns and Distributed System, and I1 teaches the subject Software Patterns and Master Project.

<Table>
User

User ID	User Name
1	S1
2	S2
3	I1
4	I2

<Table>
Subject

Subject ID	Subject Title
1	Software Patterns
2	Testing
3	Master Project
4	Distributed System

<Table>
UserAndSubject

User ID	Subject ID
1	1
1	4
3	1
3	3

Table 3 Example of Association table mapping

2.2.4 Embedded value

In some domain models, it makes sense to have small objects for recording information that do not make sense to represent in a database. For example, we may have a class called Money that records the cost of a purchase in a particular currency. However, it does not make sense to have a corresponding table for this class just to record purchase amounts. An embedded value maps the values of an object into the fields of its owner. When the owning object is loaded or saved, the corresponding embedded values are loaded or saved as well. [3]

In our project, we have the embedded value within the relationship of Choice and Multi-Choice Question. Choice is not meaningful if it is an independent table, so Choice is combined into the Multi-Choice Question table which is shown in Table 4. Multi-Choice Question inherits Question, which will be discussed above.

<Table> Multi-Choice Question				
Question ID	**choice1**	**choice2**	**choice3**	**choice4**
1	Beijing	Shanghai	Shenzhen	Hongkong
2	Lazy load	Identity Map	Unit of Work	Identity Field

Table 4 Embedded Value

2.2.5 Single table inheritance

Relational databases do not support inheritance, but we need to determine how to structure the relational database such that our inheritance hierarchy is preserved when objects are re-loaded. The single table inheritance pattern maps all fields of all classes in the hierarchy into a single table, recording the type of each object, and leaving empty all the fields in row that are not in the corresponding object. When loading a row into an in-memory object, the type field in the database is read first to determine which type of object to create, and the relevant fields are extracted from the row to create the object. All instances are saved in the one table. [7]

There are two inheritance relationships in our domain model, namely Student and Instructor inherit User, and Short-answer Question and Multi-choice Question inherit Question. For these two inheritance relationships, we both choose to use single table inheritance. Table 5 shows the details of single table inheritance pattern.

Super Class	Sub Class	Database Scheme
User	Student Instructor	In order to distinguish from these two types of users, we add an attribute "role" to User table. "role" can either be *student* or *instructor*.
Question	Multi-Choice Question Short-Answer Question	In order to distinguish from these two types of questions, we add an attribute "type" to Question table. "type" can either be *multi-choice* or *short-answer*.

Table 5 Single Table Inheritance

2.3 Data Source

Accessing a database from the domain layer presents a series of problems. One such problem is how to query the database. One solution is for a developer to encode the SQL query as a string, and send this string to the database, via some connection, directly from the domain layer. However, this approach breaks about every design principle ever proposed. Most critically, it breaks the design principle of maintaining loose coupling between the domain layer and the underlying database. [7]

To solve this problem, we need to introduce another layer called Data Source Layer. There are four patterns for this layer, namely table data gateway, row data gateway, active record and data mapper. As we choose to use Domain Model to represent the domain logic, the first choice for database interaction is Data Mapper. The reason is that data mapper will help keep the Domain Model independent from the database and is the best approach to handle cases where the Domain Model and database schema diverge. [7]

The Data Mapper is a layer of software that separates the in-memory objects from the database. Its responsibility is to transfer data between the two and also to isolate them from each other. With Data Mapper, the in-memory objects needn't know even that there's a database present, and they need no knowledge of the database schema. [3]

In our project, we have a package called "mapper", which includes all data mappers in this project. As there are four basic operations (insert, update, delete, find) that we can do with the connection of database, we create an interface called "DataMapper" which includes these basic operations. In addition, for different data mappers, we need to add some additional functions according to the requirements. For example, we need to get all subjects in the system in "SubjectMapper". Table 6 shows the functions we have in each mapper.

Mapper	Functions
DataMapper <interface>	insert, update, delete, findById
UserMapper	insert, update, delete, findById, Login, FindAllInstructor, FindAllStudentsBySubjectId, findAllStudentsAndSubmissions
SubjectMapper	insert, update, delete, findById, FindAllSubject, FindAllSubjectByUserId
ExamMapper	insert, update, delete, findById, FindAllExams, FindAllExamsBySubjectId
QuestionMapper	insert, update, delete, findById, FindAllQuestion, FindQuestionByExamId
SubmissionMapper	insert, update, delete, findById, FindByStudentId, FindSubmissionsByExamId, FindAllSubmission, FindSubmissionsByUserId_ExamId
AnswerMapper	insert, update, delete, findById, FindAnswersBySubmissionId

Table 6 Data Mapper

2.4 Object to relational behavioural design

After we have designed the data source layer and domain layer, we have a question at this stage. How do we get the data from the data source and load it into some domain objects, and then do the reverse once we have made some changes? People may answer that we can just read records from a database, and parse them into domain objects, using the data mapper, and then write them all back to the database when we are done. [3]

However, let's imagine the situation that when a large collection of records is loaded, and if we modify only a small number of them, we then write all of them back to the database, which is not efficient. In fact, we can only write back those records that have changed. To do this, we need a way of keeping track of which objects have changed. Object to relational behavioural design help us to achieve this. In our project, we have a package called "shared" to deal all issues related to this kind of patterns.

2.4.1 Unit of Work

The unit of work pattern describes a way to keep track of which domain objects have changed (or new objects created), so that only those objects that have changed need to be updated in the database. It maintains a list of objects affected by a business transaction and coordinates the writing out of changes.

The unit of work keeps four lists of objects as shown below.

- **new** - a list of new objects that have been created and must be inserted into the database.
- **dirty** - a list of existing objects that have changed since they were read from the database.
- **clean** - a list of existing objects that have not changed since they were read from database.
- **deleted** - a list of existing objects that need to be removed from the database.

Each object that is created from a query must be in exactly one of these lists. In practice, the clean list need not be created, as the set of clean objects is just the set of all objects not in the new, dirty, or deleted lists. [3]

In "shared" package, we have a class called "UnitOfWork" which has three lists to record the new, dirty and deleted objects separately. And in data source layer, namely the classes in the package "mapper", we import "UnitOfWork" when we need to make changes to the database. Table 7 shows the places where we use "UnitOfWork".

Mapper	Operation	Design Rationale
ExamMapper	insert, update, delete	Add new exams to the "new" list; Add the changed exams to the "dirty" list; Add the deleted exams to the "delete" list.
QuestionMapper	insert, update, delete	Add new questions to the "new" list; Add the changed questions to the "dirty" list; Add the deleted questions to the "delete" list.
SubmissionMapper	insert, update	Add new submissions to the "new" list; Add the changed submissions to the "dirty" list.
AnswerMapper	insert, update	Add new answers to the "new" list; Add the changed answers to the "dirty" list.

Table 7 Design Rationale of Unit of Work

2.4.2 Identity Map

The identity map pattern is a simple pattern that helps to maintain data integrity. This pattern aims to prevent the same data from being loaded into more than one object. If this is not prevented, two or more of these objects may be changed in different ways, and then during the next update, there will be a clash of values, and we will not be able to determine the correct values to update the database with. To solve this, the identity map simply maintains a mapping from the unique ID assigned by the database, to a pointer to the instance of that object. Before an object is read from the database, a lookup is performed on the identity map to see if an object with that ID already exists. If it does, the map returns that instance. Otherwise, the database is queried and a new instance is created. [3]

In our project, we have a class called "IdentityMap" in the package shared, which includes an attribute of type Map. It is also implemented in the Singleton Pattern because we need to ensure that only one instance of "IdentityMap" class exists. In "IdentityMap" class, we have two methods, one is "put", which means we need to put the objects into the Identity Map when the objects do not exist. The other one is "get", which means we need to get the objects from the Identity Map when the objects exist. The "IdentityMap" class is used in every method within the package "mapper". This means every time before we query from our database and load objects, we need to check if the objects are already in the Identity Map. If the objects exist, we will get them from the Identity Map. If the objects do not exist, we will query the database and then add them to the Identity Map.

2.4.3 Lazy Load

The lazy load pattern is the inverse of unit of work, which aims to reduce the amount of data read from database by only reading what it needs. A lazy loader loads some type of dummy objects that contain no data, and only loads the corresponding data when we try to access that data. There are four main ways that the lazy load pattern can be implemented, namely lazy initialization, virtual proxy, value holder and ghost. [3]

Our team chooses to use ghost pattern which comes from lazy initialization. In lazy initialization, each field in a class is initialized to "null", and each field must contain a getter method that first checks if the field is null, and if so, loads the object. If the field is non-null, it returns the previously-initialised value. A constraint is that all other code, including methods within the class, must use the getter method to access the field.

Ghost pattern is the same as lazy initialization, except that all fields (or multiple fields) are initialised when any of the fields are accessed for the first time. In our project, we apply ghost pattern when we deal with Question List in exams and Answer List in submissions. The design of lazy load is described in Table 8.

Situation	Design Rationale
When view all exams	Initialize all questions as "null"
When view an exam	Load the questions of that exam
When view all submissions	Initialize all answers as "null"
When view a submission	Load the answers of that submission

Table 8 Design Rationale of Lazy Load

3. Architectural Representation

Architectural representation describes the adopted model in the system. "4+1" framework is always used as a base. In this section, we include the logical view, process view, development view, physical view and scenario. Figure 2 shows the basic structure of 4+1 framework [8].

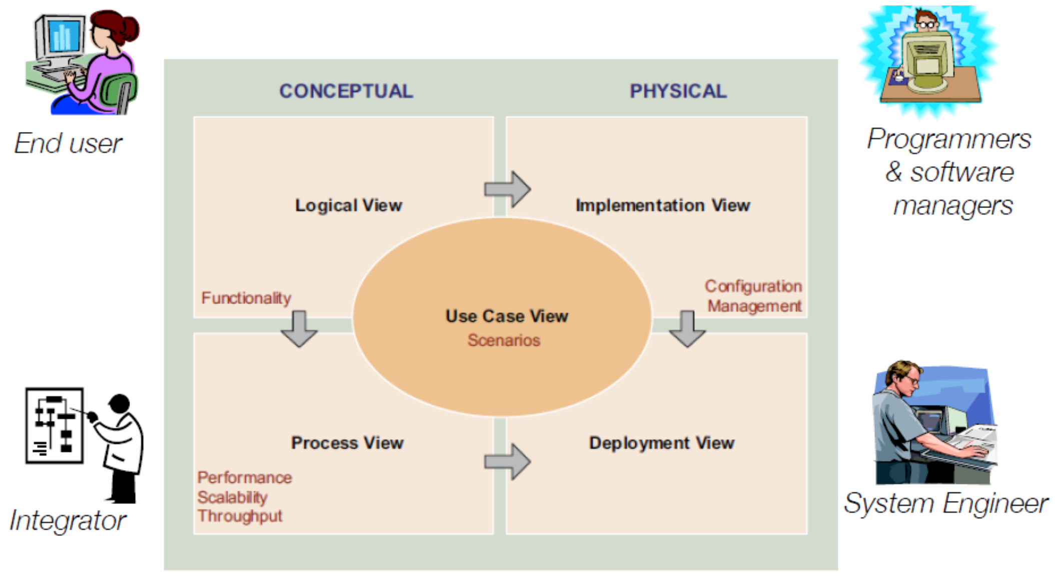

Figure 2 "4+1" Framework

3.1 Logical View

The logical view is an object-oriented decomposition. The viewer of logical view is the end-users. It considers more on functional requirements, especially what the system should provide in terms of services to its users. This view shows the components (objects) of the system as well as their interactions and relationships.

3.1.1 Class Diagram

In software engineering, a class diagram in the Unified Modelling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

The class diagram is the main building block of object-oriented modelling. It is used for general conceptual modelling of the structure of the application, and for detailed modelling translating the models into programming code. Class diagrams can also be used for data modelling. The classes in a class diagram represent both the main elements, interactions in the application, and the classes to be programmed. [9]

As shown in Figure 3, there are mainly five kinds of relationships between classes, namely dependency, realization, aggregation, composition and inheritance.

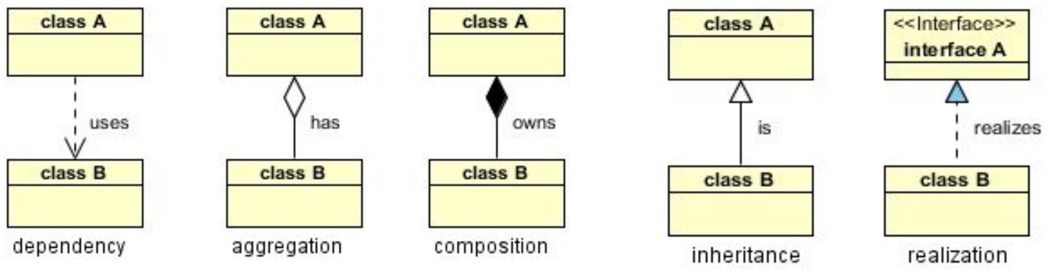

Figure 3 UML Class Diagram Relationships

Moreover, in a class, "-" represents private, "+" represents public, "#" represents protected, underscore represents static and italic represents abstract.

In our class diagram, purple represents package database, blue represents package domain, pink represents package enumeration, orange represents package mapper, white represents package service, gray represents package "serviceImp", dark green represents package servlet, light green represents package shared and red represents package util.

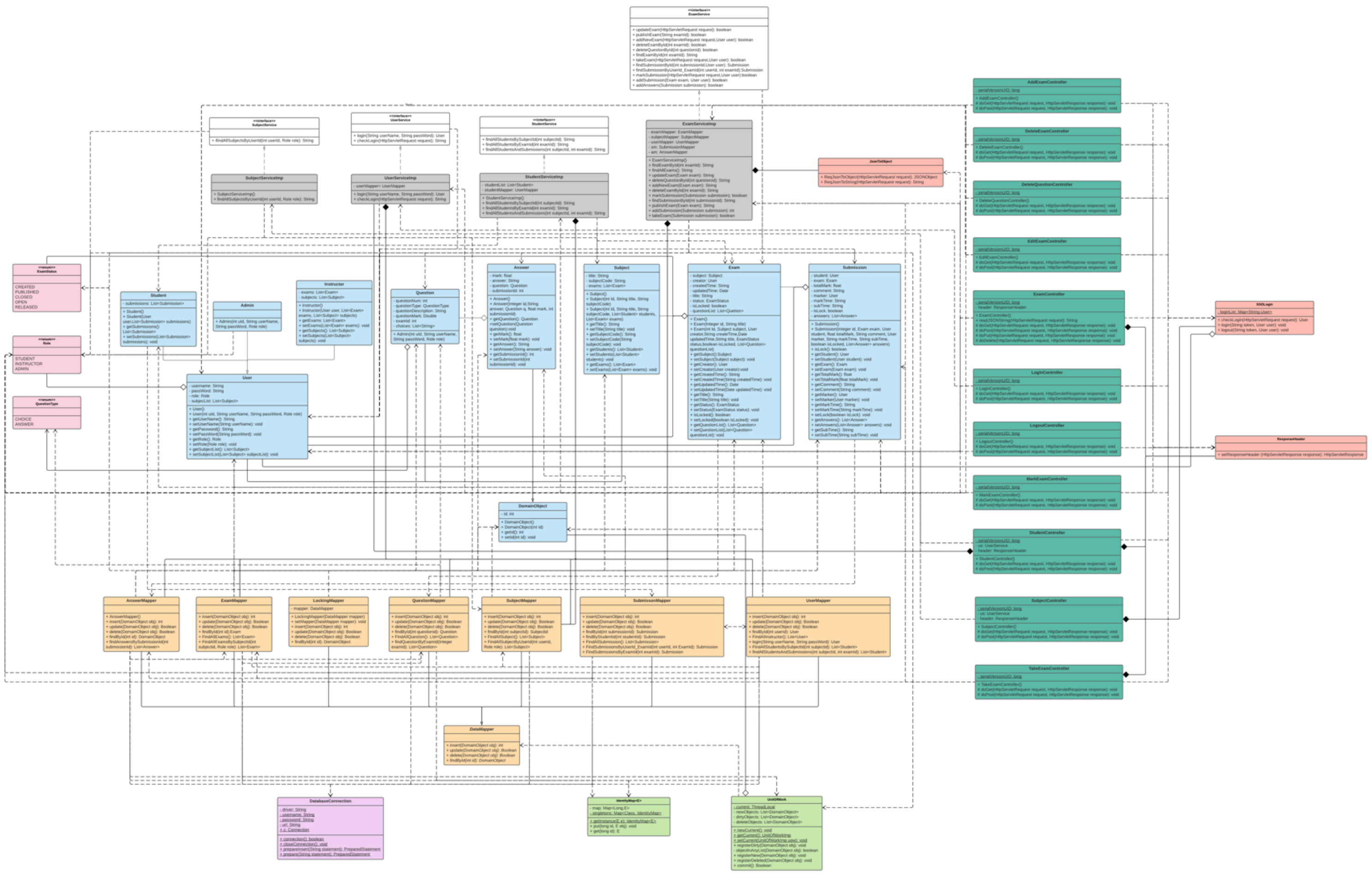

Figure 4 Class Diagram

(https://github.com/Olivia0012/SWEN90007_2020_SuperGirls/blob/master/docs/Part2_Figures/Figure%204%20Class%20Diagram.png)

The following tables show the specific relationships between classes.

Package	Class	Associated Class	Relationship
domain	Admin	User	Inheritance
		Role	Dependency
	Answer	DomainObject	Inheritance
		Question	Aggregation
	Exam	DomainObject	Dependency
		Subject; User; ExamStatus	Aggregation
	Instructor	User	Inheritance
	Question	DomainObject	Inheritance
		QuestionType	Aggregation
	Student	User	Inheritance
		Role	Aggregation
	Subject	DomainObject	Inheritance
		UnitOfWorkImp	Dependency
	Submission	DomainObject	Inheritance
		User; Exam	Aggregation
	User	DomainObject	Inheritance
		Role	Aggregation

Table 9-1 Package "domain"

Package	Class	Associated Class	Relationship
mapper	AnswerMapper	DataMapper	Inheritance
		DatabaseConnection; Answer; DomainObject; Question; Submission; QuestionMapper; ExamMapper; IdentityMap; UnitOfWork	Dependency
	DataMapper	DomainObject	Dependency
	ExamMapper	DataMapper	Inheritance
		DatabaseConnection; DomainObject; Exam; Question; Subject; User; ExamStatus; SubjectMapper; QuestionMapper; IdentityMap; UnitOfWork	Dependency
	LockingMapper	DataMapper	Inheritance

		DomainObject	Dependency
	QuestionMapper	DataMapper	Inheritance
		DatabaseConnection; DomainObject; Question; QuestionType; IndentityMap;	Dependency
	SubjectMapper	DataMapper	Inheritance
		DatabaseConnection; DomainObject; Exam; Subject; ExamMapper; Role; IdentityMap	Dependency
	SubmissionMapper	DataMapper	Inheritance
		DatabaseConnection; DomainObject; Answer; Exam; Submision; User; ExamMapper; UserMapper; AnswerMapper; IdentityMap	Dependency
	UserMapper	DataMapper	Inheritance
		DatabaseConnection; DomainObject; Exam; Student; Submission; User; SubjectMapper; SubmissionMapper; ExamMapper; Role; IdentityMap	Dependency

Table 9-2 Package "mapper"

Package	Class	Associated Class	Relationship
service	ExamService	Exam; Submission; User	Dependency
	SubjectService	Role	Dependency
	UserService	User	Dependency

Table 9-3 Package "service"

Package	Class	Associated Class	Relationship
serviceImp	ExamServiceImp	ExamService	Realization
		ExamMapper; SubjectMapper; UserMapper; SubmissionMapper; AnswerMapper; QuestionMapper; JsonToObject	Composition

		Answer; Exam; Question; Submission; User; ExamStatus; Role; UnitOfWork	Dependency
	StudentServiceImp	StudentService	Realization
		UserMapper; ExamMapper; SubjectMapper	Composition
		Exam; Student; Subject	Dependency
	SubjectServiceImp	SubjectService	Realization
		Subject; SubjectMapper; Role	Dependency
	UserServiceImp	UserService	Realization
		UserMapper	Composition
		User	Dependency

Table 9-4 Package "serviceImp"

Package	Class	Associated Class	Relationship
Servlet	AddExamController	User; ExamServiceImp; ResponseHeader; SSOLogin	Dependency
	DeleteExamController	User; ExamServiceImp; ResponseHeader; SSOLogin	Dependency
	DeleteQuestionController	User; ExamServiceImp; ResponseHeader; SSOLogin	Dependency
	EditExamController	User; Role; ExamServiceImp; ResponseHeader; SSOLogin	Dependency
		ResponseHeader	Composition
		Exam; User; ExamServiceImp; JsonToObject; SSOLogin	Dependency
	LoginController	User; UserServiceImp; ResponseHeader; SSOLogin	Dependency
	LogoutController	User; ResponseHeader; SSOLogin	Dependency
	MarkExamController	Submission;User; Role; ExamServiceImp; ResponseHeader; SSOLogin	Dependency
	StudentController	UserService; ResponseHeader	Composition
		User; Role; StudentServiceImp; UserServiceImp; SSOLogin	Dependency
	SubjectController	ResponseHeader; SSOLogin	Composition
		User; SubjectServiceImp	Dependency

	TakeExamController	ResponseHeader	Composition
		Submission; User; Role; ExamServiceImp; SSOLogin	Dependency

Table 9-5 Package "servlet"

Package	Class	Associated Class	Relationship
shared	UnitOfWork	DomainObject	Aggregation
		DataMapper	Dependency

Table 9-6 Package "shared"

Package	Class	Associated Class	Relationship
shared	SSOLogin	User	Aggregation

Table 9-7 Package "util"

3.2 Process View

The process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the runtime behaviour of the system. Process view can solve the problems of concurrency, distribution, integrators, performance and scalability. The sequence diagram is a way to achieve the goal of process view.

3.2.1 Sequence Diagram

A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place. Sequence diagrams describe how and in what order the objects in a system function. Sequence diagram includes some components shown below.

a. Actor

System roles can be people, other systems or subsystems. It is represented by a little icon of a person.

b. Object

The object is at the top of the sequence diagram and is represented by a rectangle.

c. Lifeline

In the sequence diagram, each object has a vertical dashed line at the bottom centre, which is the lifeline (timeline) of that object.

d. Activation

The activation represents the operation performed at a certain period of time on the object timeline in the sequence diagram. It is represented by a very narrow rectangle.

e. Message

Message represents the information sent between objects. There are two types of messages in our sequence diagrams.

- *Synchronous Message* - The sender of the message passes control to the receiver of the message and then stops the activity, waiting for the receiver of the message to return the control. It is used to represent the meaning of "synchronization". It is represented by a solid arrow in solid lines.

- *Return Message* - It is necessary to return message of procedure calls. It is represented by a normal arrow in dashed lines.

f. Combination fragment

Several types of fragments exist in sequence diagram such as alternative (if-elseif-else or if-else), option (if) and loop (while). We only use "alternative" in our sequence diagram. It is represented as "Alt". There is only one sequence happens according to the condition.

As our system has two roles (instructor and student), we design a sequence diagram for each of them. Figure 5 and Figure 6 shows the sequence diagrams of our project.

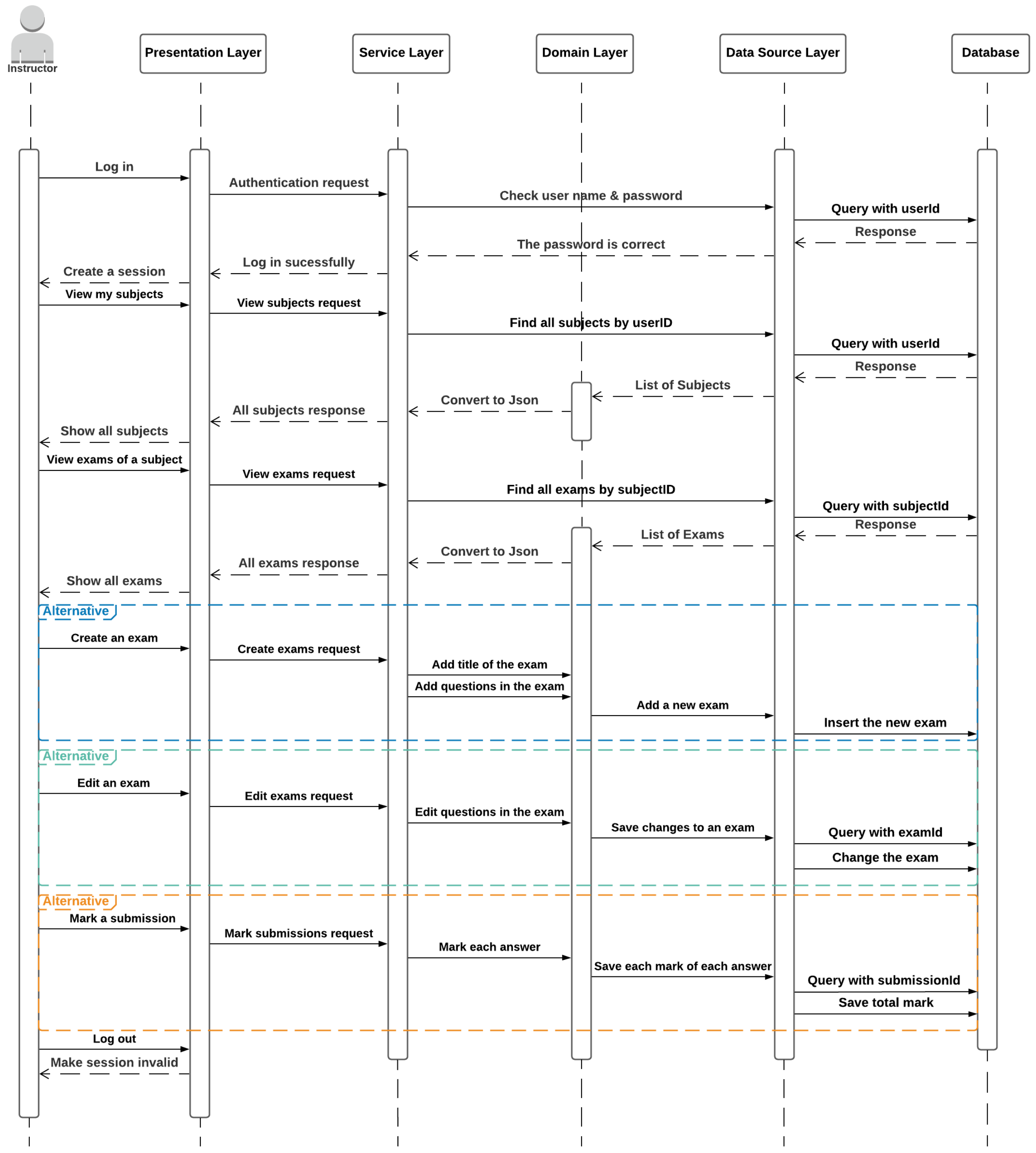

Figure 5 Sequence Diagram of Instructor

An instructor can log in to the system first. Then all associated subjects and all exams of each subject are displayed. The instructor can choose to either create an exam, edit an exam or mark a submission. Finally, the instructor can log out from the system.

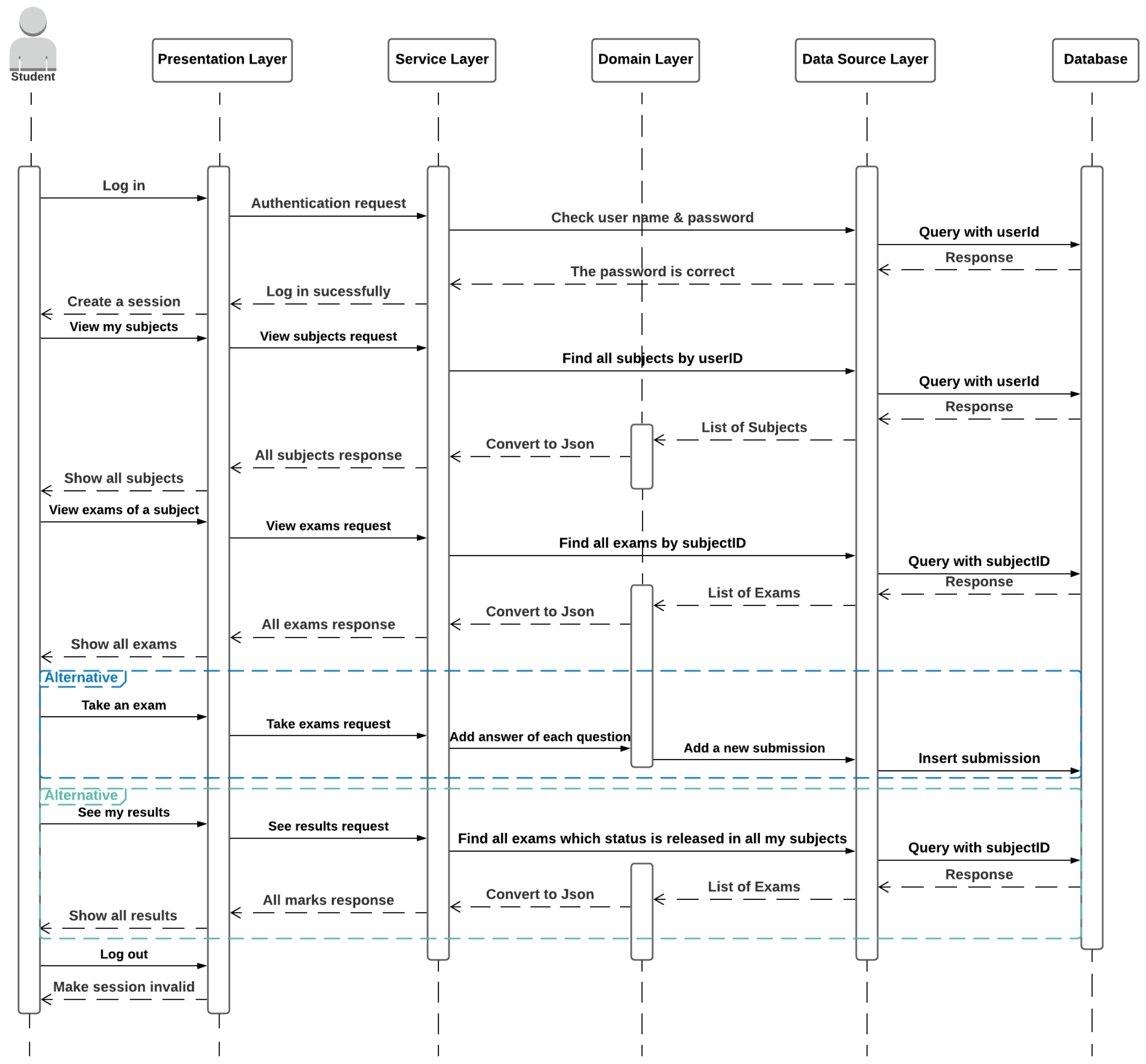

Figure 6 Sequence Diagram of Student

A student can log in to the system first. Then all associated subjects and all exams of each subject are displayed. The student can choose to either take an exam or see the results of the submissions. Finally, the student can log out from the system.

As is shown above, we develop our sequence diagram with tracking the process of information between different layers. Table 10 shows the description of different layers in the system.

Object in Sequence Diagram	Description
Instructor / Student	Instructor and Student are wo types of end users.
Presentation Layer	Presentation Layer mainly contains front-end of the system. Front-end is implemented by React framework.

Service Layer	Service Layer contains servlets which receive requests from front-end and deal with the requests.
Domain Layer	Domain layer contains domain model. Domain objects can be returned after querying from the database.
Data Source Layer	Domain layer contains data mappers which have interaction with the database directly.
Database	We use a cloud database on Heroku to store data of the system.

Table 10 Objects in Sequence Diagram

3.3 Development View

The development view (also called implementation view) is the subsystem decomposition. The viewers are programmers and software managers. It considers software module organization including components, subsystems and the relationships between them. It also gives a building block view of the system. The component diagram can be applied to describe the implementation view. [10]

3.3.1 Component Diagram

The component diagram depicts how components are connected together to form larger components or software systems. We create the component diagram for back-end shown in Figure 7. The component diagram contains several parts shown below.

- *Component* - the components in our system (specified with a component symbol)
- *Subsystem* - the subsystem in our system (specified as <<subsystem>>). A subsystem can contain several components.
- *Interface* - a set of operations provided by one class to another (specified as a small circle)
- *Relationship* - association and interface realization (specified as a line)
- *Port* - a port is used to connect subsystems and components (specified as a small rectangle)

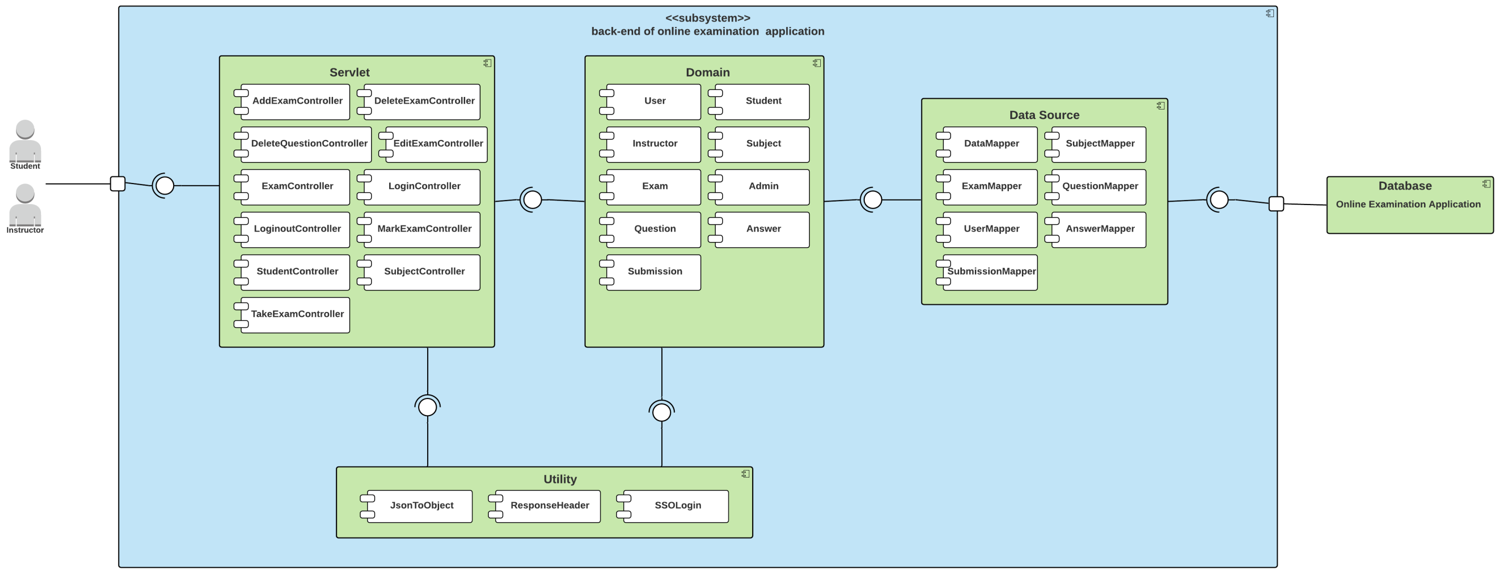

Figure 7 Component Diagram

3.3.2 Frameworks and Libraries

Our team chooses to use React framework to implement front-end, and to use J2E framework to implement back-end as required. The connection between front-end and back-end is achieved by using REST API in Json format. The libraries and the description are shown in the table below.

Jar	Description
postgresql-42.2.1.4.jar	Connect to PostgreSQL database
fastjson-1.1.6.jar	Convert data to Json format
servlet-api.jar	Support servlets to get and post requests

Table 11 Libraries

3.3.3 Development Environment

Development environment includes all IDEs and all components we use to run the system. Table 12 shows the description of each development environment.

Development Environment	Description
Eclipse 2020-06	Back-end development IDE
J2E	Back-end development framework
JDK 14	Specific JDK version to develop back-end
Apache Tomcat 9.0	Specific Tomcat version to run the project
VS Code 1.49	Front-end development IDE
React 16.13.1	Front-end development framework

Table 12 Development Environment

3.4 Physical View

The physical view depicts the system from a system engineer's point of view. It is concerned with the topology of software components on the physical layer as well as the physical connections between these components. This view is also known as the deployment view. UML diagrams used to represent the physical view include the deployment diagram. [10]

3.4.1 Deployment Diagram

Our team deploys the front-end and back-end of the system to the Heroku platform. We also launch a cloud database on Heroku platform. When an end-user visits the system through a web browser, an HTTP request is sent to the Heroku server. Our front-end and back-end can communicate with each other through REST API, and our back-end can connect to the database by using JDBC. This process is shown in Figure 8-1. We also deploy the front-end to Amazon EC2 server by using Docker. Then the front-end can access the back-end remotely through REST API. This process is shown in Figure 8-2. The end-users can access either the Amazon server or Heroku server.

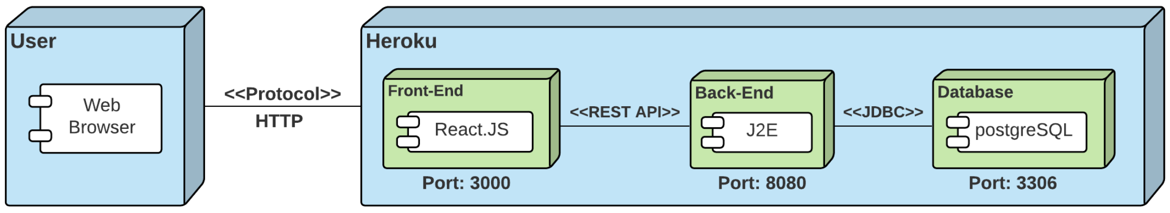

Figure 8-1 Deployment Diagram – 1

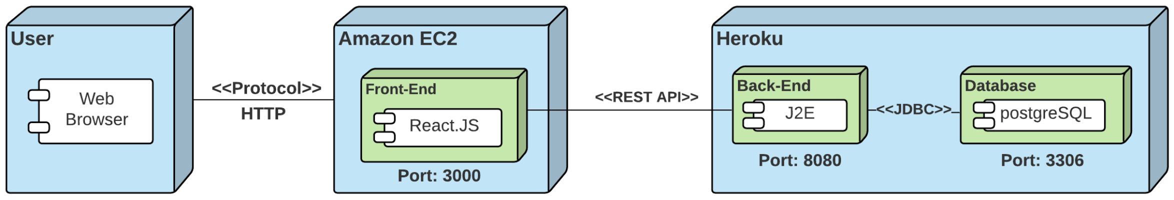

Figure 8-2 Deployment Diagram – 2

3.5 Scenario

The description of an architecture is illustrated using a small set of use cases, or scenarios, which become a fifth view. The scenarios describe sequences of interactions between objects and between processes. They are used to identify architectural elements and to illustrate and validate the architecture design. They also serve as a starting point for tests of an architecture prototype. This view is also known as the use case view. [10]

3.5.1 Use Case Diagram

Figure 9 shows the use case diagram. In the online exam application, there are two modules, namely subject module and exam module. In each module, there are different use cases which are designed for instructors and students.

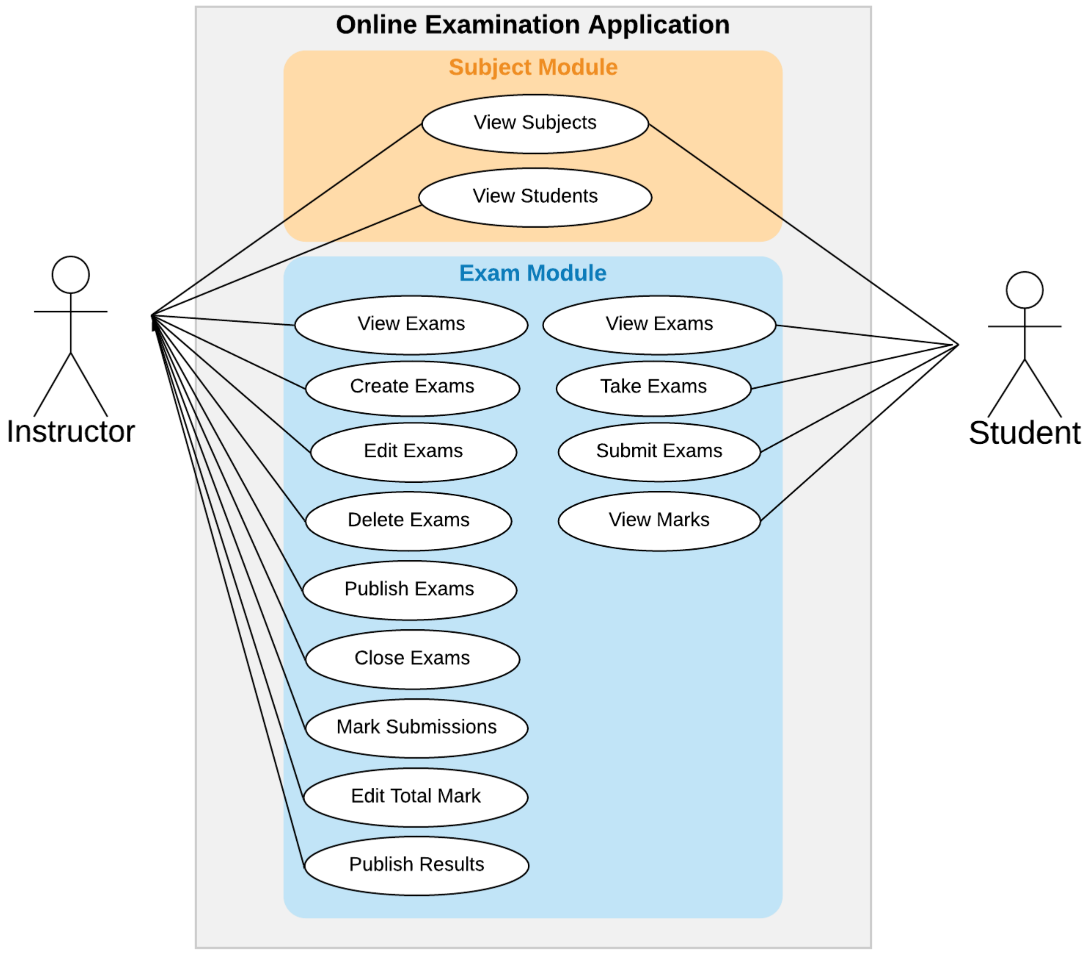

Figure 9 Use Case Diagram

4. User Manual

4.1 Log in as an Instructor

- Enter https://swen90007-frontend.herokuapp.com/ or http://35.174.205.208:3000/ in web browser
- Enter the username *Edu* and password *111* to log in.
- Click "Subjects" on the left, the instructor can see all associated subjects and all exams of each subject.
- Choose a subject, click the green "+" button and add the title of a new exam.
- Choose an exam in "CREATED" status and click the title of it. The instructor can view and edit it.
  - Click "EDIT" button at the top right, the instructor can edit the exam.
  - For a new question, click "ADD NEW QUESTION", then enter the question description, choose the question type and enter the mark.
  - For an existing question, edit the question description, question type or question mark.
  - Click "SAVE" button to save all the changes.
  - Click "CANCEL" button to cancel all the changes.
  - Click "PUBLISH" at the top right to publish the exam.
  - Click "DELETE" at the top left to delete the exam.

- Choose an exam in "PUBLISHED" status and click the title of it. The instructor can view the exam and close it.
    - Click "CLOSE" button to close an exam.
- Choose an exam in "CLOSE" status and click the title of it. The instructor can view the exam and release the results of it.
    - Click "RELEASE" button to release the results of an exam.
- Choose a subject and an exam, click "VIEW" button, the instructor can view all students enrolled in a subject.
    - To mark a submission, click "MARK" button, enter the mark of each question and click the "SUBMIT".
    - Click "VIEW" button, the instructor can view a marked exam.
- Click the button on the top right in the navigation bar, the instructor can log out from the system.

4.2 Log in as a Student

- Enter https://swen90007-frontend.herokuapp.com/ or http://35.174.205.208:3000/ in web browser
- Enter the username *Olivia* and password *111* to log in
- Click "Subjects" on the left, the student can see all associated subjects and published or released exams of each subject
- Choose an exam in "RELEASED" status and click "VIEW" button, the student can view the mark and comment of this exam.
- Choose an exam in "PUBLISHED" status and click "TAKE" button. The student only has one chance to take the exam. If the student has already taken the exam, click "TAKE" and the system will prompt that you have taken the exam.

5. Appendix

5.1 Link to the Application

https://swen90007-frontend.herokuapp.com/ or http://35.174.205.208:3000/

5.2 Github Link

https://github.com/Olivia0012/SWEN90007_2020_SuperGirls

5.3 Git Release Tag

SWEN90007_2020_Part2_SuperGirls

6. References

[1] "Architectural pattern", En.wikipedia.org, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Architectural_pattern. [Accessed: 23- Sep- 2020].
[2] "Business logic", En.wikipedia.org, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Business_logic. [Accessed: 23- Sep- 2020].
[3] The University of Melbourne, "Software Design and Architecture Subject Notes for SWEN90007", 2020.
[4] The University of Melbourne, "Architecture document", 2020.
[5] Kruchten, P. B. "The 4+ 1 view model of architecture", IEEE software, 12(6), 42-50
[6] "User guide", En.wikipedia.org, 2020. [Online]. Available:

https://en.wikipedia.org/wiki/User_guide. [Accessed: 23- Sep- 2020].
[7] M. Fowler, Patterns of enterprise application architecture. Boston, Mass.: Addison-Wesley, 2015.
[8] The University of Melbourne, “SWEN90014 Masters Software Engineering Project. Lecture 7 ARCHITECTURE: 4+1 VIEW MODEL”, 2019.
[9] “Class diagram”, En.wikipedia.org, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Class_diagram. [Accessed: 23- Sep- 2020].
[10] “4+1 architectural view model”, En.wikipedia.org, 2020. [Online]. Available: https://en.wikipedia.org/wiki/4%2B1_architectural_view_model. [Accessed: 23- Sep- 2020].