Architecture Document

SWEN90007: Software Design and Architecture

The Team
SWEN90007 SM2 2020 Project
Github: https://github.com/emilylm/swen90007-project.git
Heroku: https://swen90007-exam-app.herokuapp.com

In charge of: Emily Marshall <emilylm@student.unimelb.edu.au>, emilylm, 587580
Luke Rosa <lrosa@student.unimelb.edu.au>, lrosa, 319522

THE UNIVERSITY OF MELBOURNE

SCHOOL OF COMPUTING & INFORMATION SYSTEMS

Revision History

Date	Version	Description	Author
16/09/2020	01.00-D01	Initial draft	Emily Marshall Luke Rosa
19/09/2020	01.00-D02	Added 4+1 architecture views	Emily Marshall Luke Rosa
22/09/2020	01.00-D03	Added patterns used in project	Emily Marshall Luke Rosa
01/10/2020	01.00-D04	Added rationales for patterns used in the project	Emily Marshall Luke Rosa
11/10/2020	01.00-D05	Finished writing pattern justification	Emily Marshall Luke Rosa
11/10/2020	01.00-D06	Final review of document	Emily Marshall Luke Rosa
11/10/2020	01.00	Final version of this document	Emily Marshall Luke Rosa

Table of contents

1. Introduction

This document specifies the SWEN90007 project system architecture describing its main standards, module, components, frameworks, and integrations.

1.1 Proposal

The purpose of this document is to give, in high level overview, a technical solution to be followed, emphasizing the components and frameworks that will be reused and researched, as well as the interfaces and integration of them.

1.2 Target users

This document is aimed at the project team and teaching team of SWEN90007, with a consolidated reference to the research and evolution of the system with the main focus on technical solutions to be followed.

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
Component	Reusable and independent software element with well-defined public interface, which encapsulates numerous functionalities and which can be easily integrated with other components.
Module	Logical grouping of functionalities to facilitate the division and understanding of software.

1.4 Glossary of synonyms

Synonyms
Marks, points, grades
Examination Application, system

2. Architectural representation

The specification of the system's architecture Examination Application follows the framework "4+1" **Error! Reference source not found.**, which defines a set of views, as shown in Figure 1. Each of these views approaches aspects of architectural relevance under different perspectives:

- The **logical view** shows the significant elements of the project for the adopted architecture and the relationship between them. Between the main elements are modules, components, packages and the application main classes;
- The **process view** shows the concurrency and synchronization aspects of the system, mapping the elements of the logical view to processes, *threads* and execution tasks;
- The **development view** focuses on aspects relating to the organization of the system's source code, architectural patterns used and orientations and the norms for the system's development;
- The **physical view** shows the hardware involved and the mapping of the software elements to the hardware elements in the system's environment.
- The **scenarios** show a subset of the architecturally significant use cases of the system.

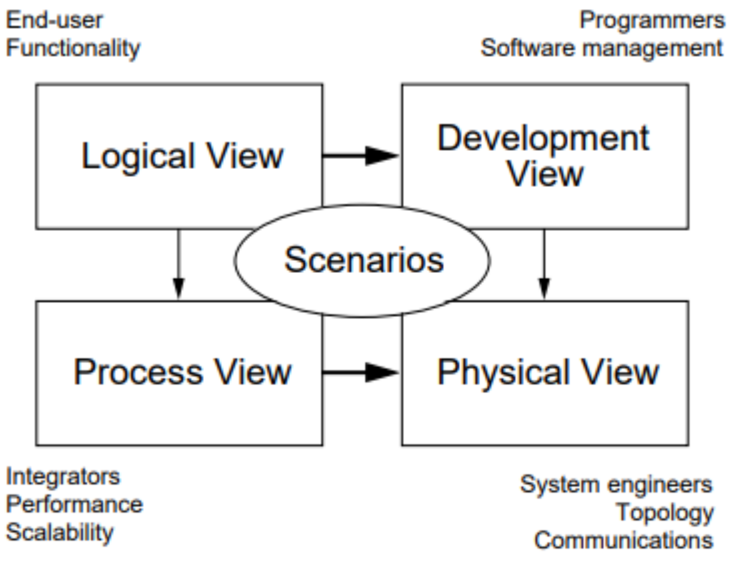


Figure 1. Views of *framework* “4+1” [1]

3. Architectural objectives and restrictions

The defined architecture’s main objective is to develop a system that fully meets all functional requirements of the application specification (see specification document).

3.1 Qualities

The qualities of service of the Examination Application cover performance, tolerance to imperfections, usability, security, maintainability, portability, and concurrency.

- The system is expected to perform to a level to guarantee its usability – there are no strict performance timeframes that have placed on the system but the general qualities are that the application should feel responsive and not freeze nor crash.
- The system should also have a tolerance for imperfections. Wherever possible, the application should be tolerant of non-well-formed inputs and should provide feedback to the user (for example, when inserting new student users, university conventions for student IDs should be followed and
- Data integrity should be guaranteed by the system – data stored should be accurate, complete, and reliable.
- Business rules for the business domain should be followed – for example, administrators should not be able to register in classes, students should not be able to instruct classes, instructors should not be able.
- The application is expected to guarantee the security of the system – all users have been given a role which defines the system components they have access to and the transactions they may perform. User passwords should also be encrypted in the database and not stored in plaintext, as per best practice.
- There are no strict usability requirements but the user interface and presentation layer should enable (and not impede) users’ use of the application and should be clearly laid out so user’s know how to operate the application.
- The application should be maintainable and extendible in future should the need arise. It should also be portable and accessible from any desktop computer and web browser.
- The application should support concurrency where users can perform actions on the same database object without these transactions violating ACID compliance for

databases. This will be further extended in part 3 submission and is constrained (detailed below).

- The system should support high cohesion and low coupling – the responsibilities of an element of the system should be focused and strongly related and elements should be as independent as possible.

3.2 Constraints

- The application does not currently support concurrency where there is more than one instructor performing transactions on the same exam.
- As per the project specifications:

  All exams allow a single attempt, meaning students can only start and submit their exam once.

  This has been interpreted by the project team to mean students can start and submit an exam but does not account for situations where an exam is commenced by a student and the browser is closed or the application crashes prior to submission. In this case, no responses will be recorded and so no attempt is persisted in the database.

3.3 Principles

- To support database querying efficiency, lazy load and unit of work patterns have been used to make database querying efficient. In-memory domain objects have been used to store database queries in order to optimise read time of the application. Write time has been optimised through the unit of work, where certain transactions create a unit of work to commit to the database at the completion of a transaction. This also ensures data integrity and supports ACID compliance.
- To ensure a tolerance for imperfections, the system has been developed with error handling so will prompt the user when inputs are not well formed. Not well formed inputs include inputs that do not meet database requirements (for example, user IDs must be an integer) or conflict with business domain rules (for example, an exam must one or more questions).
- In order to guarantee the security of the system (authorisation and authentication), user roles have been implemented for access control to system transactions. PostgreSQL's encryption extension, PGCrypto[1], has been implemented on the database in order to encrypt user passwords prior to storing. Best practice dictates passwords should never be saved in a database as plain text.
- Concurrency has been implemented in part 2 submission through the normalisation of the database of the application – each student transaction commits data to their own database row of a table and so does not conflict with any other students.

3.4 Requirements of architectural relevance

This section lists the requirements that have impact on the system's architecture and the treatment given to each of them.

Requirement	Impact	Treatment
Security	The application must implement basic security behaviours: • Authentication: login using university credentials and a password;	• Passwords have been persisted in the database. They have been stored as a binary hash, calculated using a Blowfish (cipher)[1].

	• Authorisation: according to their role, users must be granted permission to perform some specific actions (e.g. instructors can create exams); • Passwords must be encrypted on the database and not stored as plain-text; and • Data sent across the network must not be modified by a tier.	• User object has been created with a role attribute to determine each user's role and their associated privileges.
Data persistence	Data persistence will be addressed by using a relational database.	A PostgreSQL relational database will be used to persist data.
Extensibility	The system should support future additions and changes.	The domain model pattern has been used to support this extensibility.

3.5 Reuse approach

In order to meet the architectural requirements of this system, a number of components will be reused, as they are commonly available and thoroughly tested, and a number of components will be developed by the project team.

Component	Development	Comments
Data persistence	Reuse	Build on PostgreSQL.
Database driver	Reuse	Utilise JDBC.
Authentication	Reuse, develop	Build on PostgreSQL's PGCrypto extension to encrypt passwords stored in the database. Develop authentication for a login/logout process.
Authorisation	Develop	Develop access control for different user tiers.
Session control	Reuse	Build on Java Servlet HttpSession.
Log	Reuse	Make use of Java's Logger for system logging.
Presentation layer	Reuse	Build on JQuery and JTSL.

4. Logical View

This section shows the system's organisation from a functional point of view. The main elements, like modules and main components are specified. The interface between these elements is also specified.

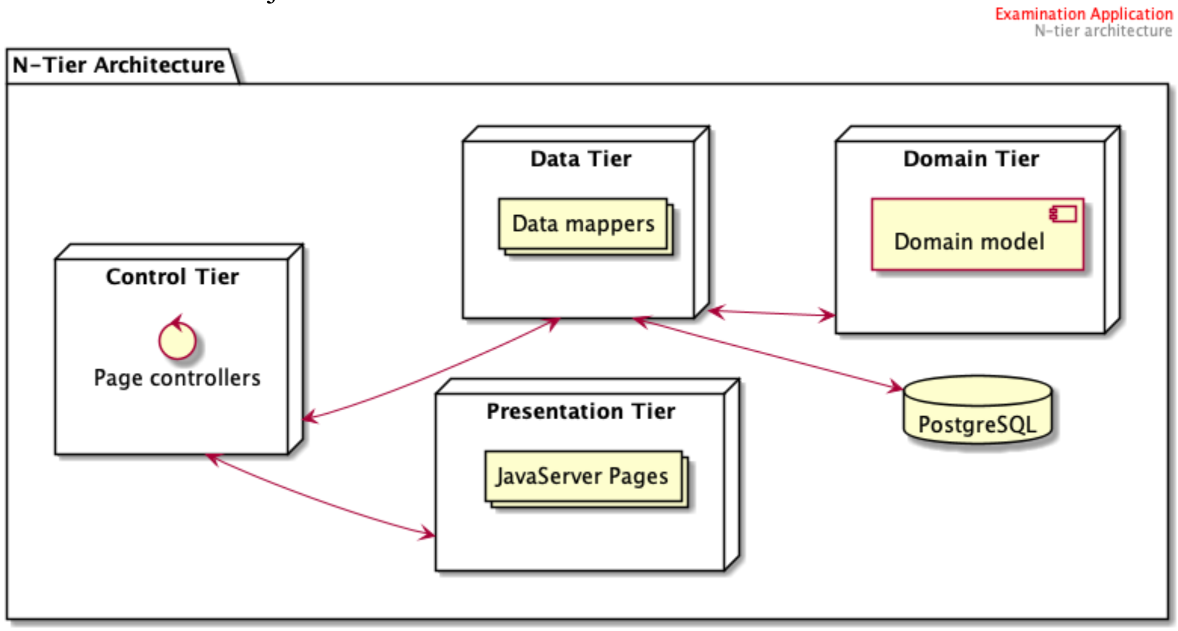
The Examination Application is divided into layers based on N-tier architecture – it is based on a responsibility layering strategy that associates each layer with a particular responsibility. This strategy has been chosen for a number of reasons:

- It provides for data integrity as it isolates system responsibilities from one another;
- It allows for changes to one layer without impacting others, support exensibility; and
- The tiers allow the project team to encode finer grain security policies, especially policing roles (student, instructor, administrator) of users.

Each layer has specific responsibilities:

- The **presentation layer** deals with the presentation logic and rendering pages;
- The **control layer** manages the access to the domain layer;
- The **data layer** is responsible for the access to the PostgreSQL database; and
- The **domain layer** is related to the business logic and manages the accesses to the resource layer.

In the presentation layer, Java Server Pages (JSP) have been created – this is a server-side technology that enables the creation of dynamic, platform-independent method for building Web-based applications
Through user input, these JSPs speak to the controllers which introduce servlet functions in the control layer. Through the use of servlet requests and responses, the servlets in the control layer pass on user requests to the data layer.
The data mapper layer receives requests from the control layer, accesses the database and instantiates domain objects.



5. Process View

There's only one process to take into account – the Examination Application does not make use of threads.

6. Development view

6.1 Architectural patterns

6.1.1 Summary of architectural patterns

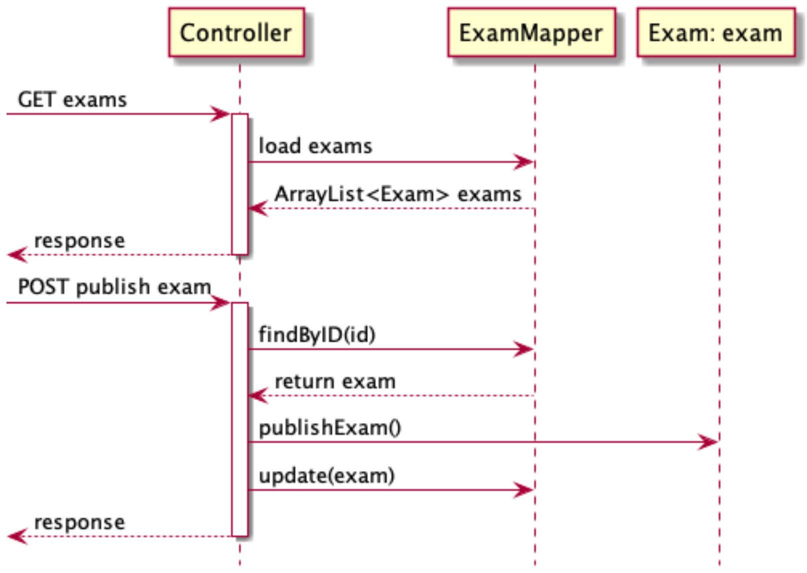
There were a number of patterns used in this project to support the architectural goals of the system. The patterns used are as follows:

- MVC with Page Controller;
- Domain model;
- Data mapper;
- Identity field;
- Foreign key mapping;
- Unit of work;
- Lazy load;
- Association table mapping;
- Embedded value
- Single table inheritance

6.1.2 Detailed view of architectural patterns

MVC with Page Controller
The application implements an *MVC*, or '*Model View Controller'*, architecture. The Controller acts as a mediator that interacts with the model, encapsulating the domain layer in this context, and the view, consisting of JSP files serving HTML pages. The controller implements the 'Page Controller' pattern, where individual pages or actions are mapped to a single controller. In this case, that controller is a Java Servlet. The page controllers generally handle a single action, or closely related set of actions. They vary in complexity accordingly. For example, the '*EditExamController'* handles all actions relating to editing an exam, which includes handling multiple POST requests. On the other hand the 'View*SubjectController'* handles the single responsibility of displaying a subject list. The choice to use the Page Controller pattern was made to improve simplicity and understandability of the code. It clearly delineates the role of each controller, mapping closely to each view. However, this pattern also introduces complexity and repetition in the controller code. This trade off was considered and accepted by the project team. The general flow of events is as follows: 1. The controller makes a call to the appropriate data mapper load some data 2. The data mapper returns the loaded objects 3. The controller instructors the domain layer to perform some logic on the domain objects 4. The controller passes the updated objects to the appropriate data mapper to persist changes

The following is an example of this flow of events for publishing an exam:



The class diagram for the **controllers** package can be found in the appendix.

Domain model
The domain of the Examination Application has been created using a description of the domain from the perspective of classification by objects. The domain has been decomposed into objects through the identification of entities, attributes, and associations that are considered noteworthy to be modelled in the system. The project team has used noun and noun phrase identification in the textual description of the domain in order to identify potential candidates for entities, attributes, or associations.
Reasons for use
Domain model supports the extensibility of the application – in order to extend the application in future, it can be easily done by extending or adding to the domain objects. The complicated business rules (for example, students can only submit one response), are better handled using the domain model. The domain model more naturally fits the business domain so allowed the project team to more easily model business rules between domain objects. Modelling the specifications using the domain model also provides benefits in implementing security requirements – users can be modelled in the domain as administrators, students or administrators allowing the project team to more easily implement the authorisation requirements for different tiers of users (for example, on instructors can create an exam).

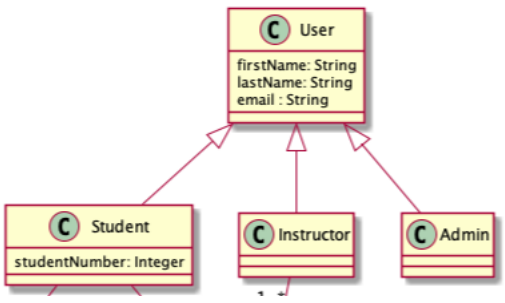
Examples of use

Based on the specifications provided for the Examination Application, the system entities, attributes, and business rules can be summarised as:

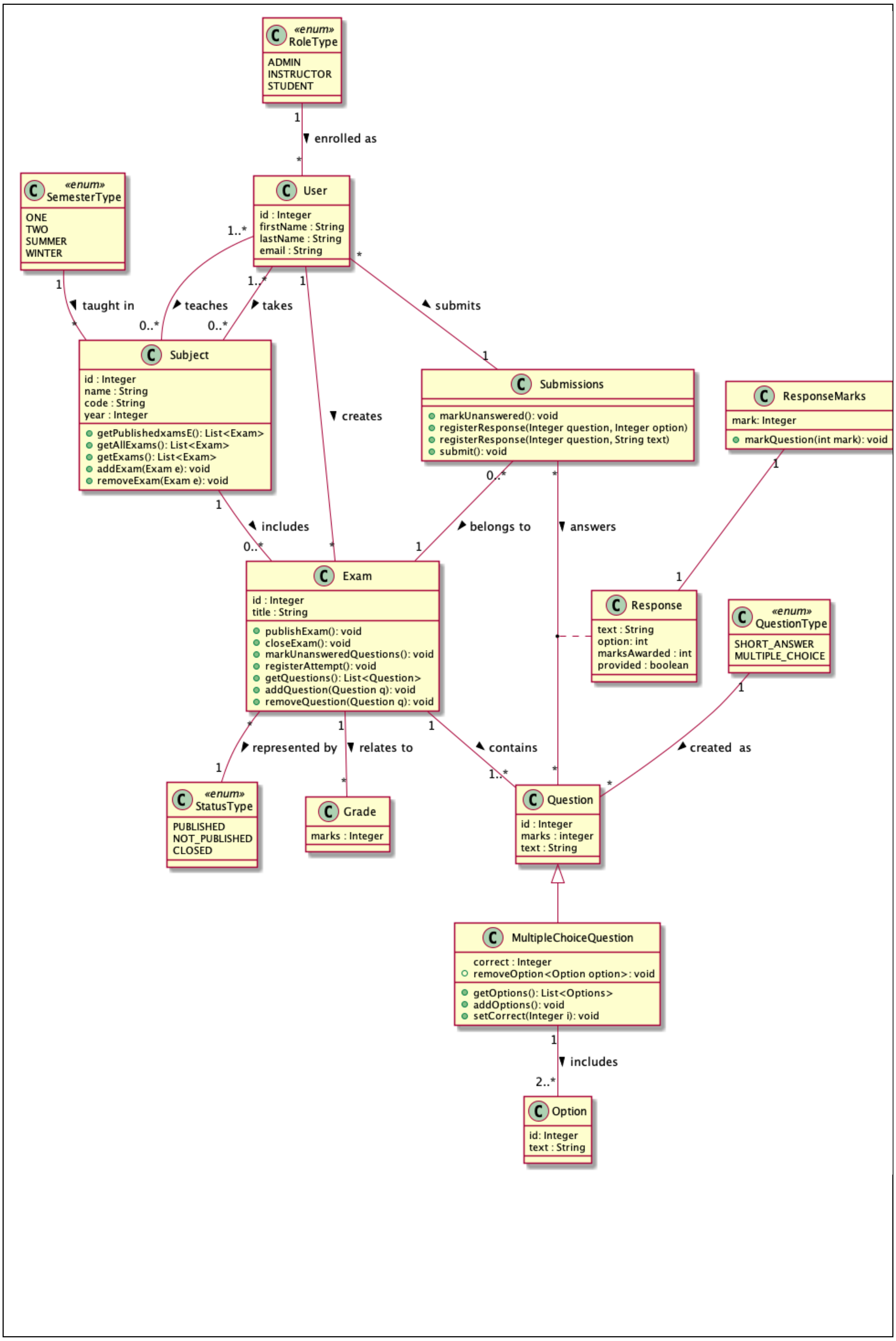
- **Users** can be either an **Administrator**, **Instructor** or **Student;**
- Only Administrators can create **Subjects** and user accounts;
- *One or more* Instructors can instruct a Subject;
- *One or more* Students are enrolled in a Subject;
- Only Instructors can create **Exams;**
- Exams have *one or more* **Questions;**
- Questions *are either* **Short-Answer** or **Multiple-Choice;**
- Multiple-Choice Questions have *one or more* **Options;**
- All exams *allow a single attempt* by a student; and
- Instructors assign a **Mark** for exam attempts

Entities have been **bolded**; attributes have been underlined; and important associations have been *italicised*.

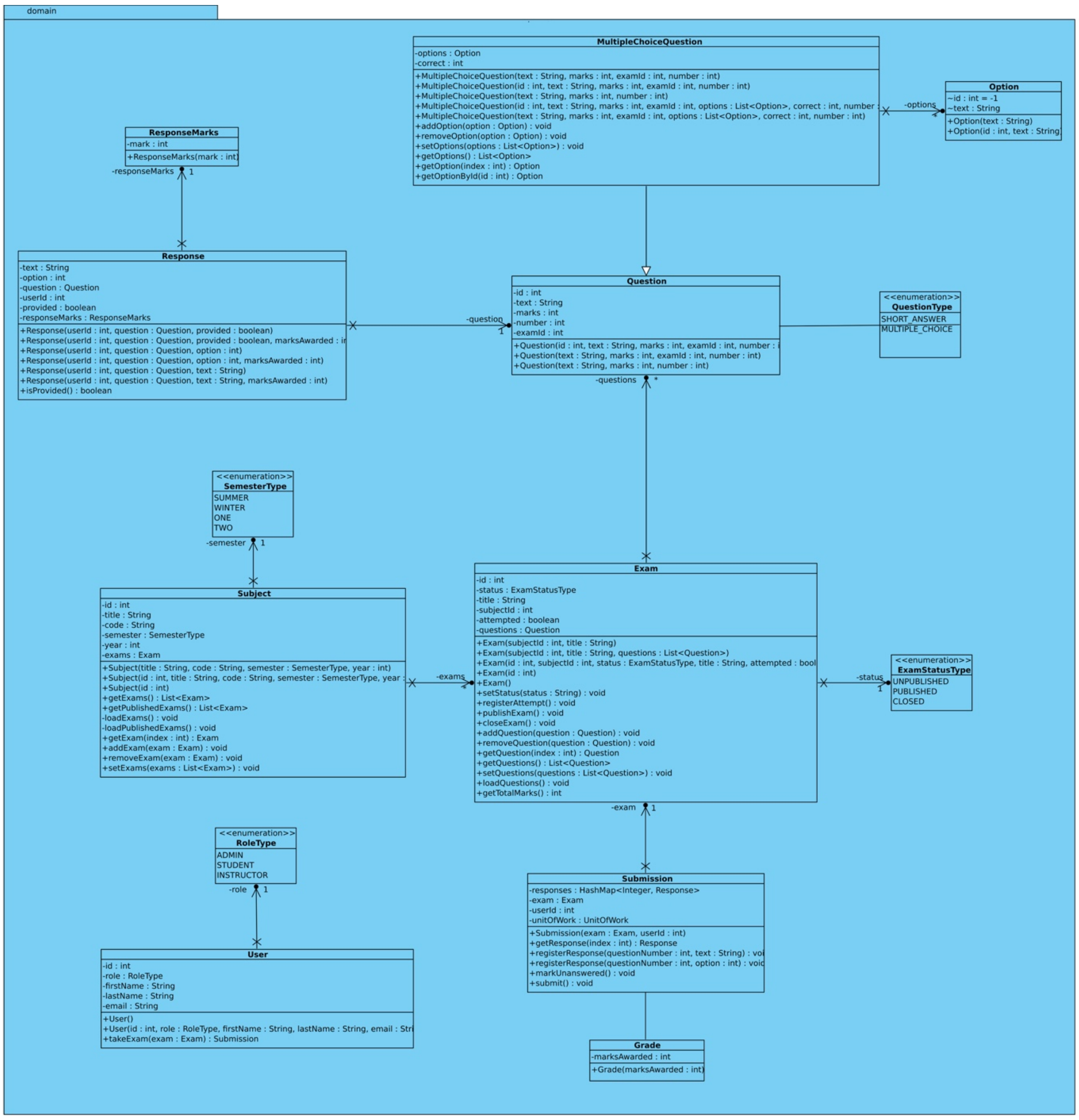
The original domain modelled Instructors and Students as subclasses of a superclass User:



However, during implementation, it was decided that Instructors, Students and Admin users would not be implemented as separate classes, but rather that Users would be assigned an enumerated 'role' type of 'STUDENT', 'INSTRUCTOR' or 'ADMIN'. This role would be determined after authentication and used to authorise any restricted actions in the application. The decision stems from the fact that each user role has limited variability. Additionally, this allowed the ease of selecting role-based views in the controller layer, which was deemed beneficial for integrating authorization into the application in future stages. As a result, the revised domain model is as follows:


«enum»
RoleType
ADMIN
INSTRUCTOR
STUDENT
1
enrolled as
*
«enum»
SemesterType
ONE
TWO
SUMMER
WINTER
User
id : Integer
firstName : String
lastName : String
email : String
1..*
1..*
1
*
1
taught in
*
teaches
0..*
takes
0..*
submits
1
Subject
id : Integer
name : String
code : String
year : Integer
getPublishedxamsE(): List<Exam>
getAllExams(): List<Exam>
getExams(): List<Exam>
addExam(Exam e): void
removeExam(Exam e): void
creates
Submissions
markUnanswered(): void
registerResponse(Integer question, Integer option)
registerResponse(Integer question, String text)
submit(): void
ResponseMarks
mark: Integer
markQuestion(int mark): void
1
includes
0..*
*
0..*
belongs to
1
*
answers
1
1
Exam
id : Integer
title : String
publishExam(): void
closeExam(): void
markUnansweredQuestions(): void
registerAttempt(): void
getQuestions(): List<Question>
addQuestion(Question q): void
removeQuestion(Question q): void
Response
text : String
option: int
marksAwarded : int
provided : boolean
«enum»
QuestionType
SHORT_ANSWER
MULTIPLE_CHOICE
1
*
1
1
represented by
relates to
contains
created as
1
*
1..*
*
*
«enum»
StatusType
PUBLISHED
NOT_PUBLISHED
CLOSED
Grade
marks : Integer
Question
id : Integer
marks : integer
text : String
MultipleChoiceQuestion
correct : Integer
removeOption<Option option>: void
getOptions(): List<Options>
addOptions(): void
setCorrect(Integer i): void
1
includes
2..*
Option
id: Integer
text : String

The class diagram for the domain layer is:


domain
MultipleChoiceQuestion
-options : Option
-correct : int
+MultipleChoiceQuestion(text : String, marks : int, examId : int, number : int)
+MultipleChoiceQuestion(id : int, text : String, marks : int, examId : int, number : int)
+MultipleChoiceQuestion(text : String, marks : int, number : int)
+MultipleChoiceQuestion(id : int, text : String, marks : int, examId : int, options : List<Option>, correct : int, number :
+MultipleChoiceQuestion(text : String, marks : int, examId : int, options : List<Option>, correct : int, number : int)
+addOption(option : Option) : void
+removeOption(option : Option) : void
+setOptions(options : List<Option>) : void
+getOptions() : List<Option>
+getOption(index : int) : Option
+getOptionById(id : int) : Option
-options
*
Option
~id : int = -1
~text : String
+Option(text : String)
+Option(id : int, text : String)
ResponseMarks
-mark : int
+ResponseMarks(mark : int)
-responseMarks
1
Response
-text : String
-option : int
-question : Question
-userId : int
-provided : boolean
-responseMarks : ResponseMarks
+Response(userId : int, question : Question, provided : boolean)
+Response(userId : int, question : Question, provided : boolean, marksAwarded : i
+Response(userId : int, question : Question, option : int)
+Response(userId : int, question : Question, option : int, marksAwarded : int)
+Response(userId : int, question : Question, text : String)
+Response(userId : int, question : Question, text : String, marksAwarded : int)
+isProvided() : boolean
-question
1
Question
-id : int
-text : String
-marks : int
-number : int
-examId : int
+Question(id : int, text : String, marks : int, examId : int, number : i
+Question(text : String, marks : int, examId : int, number : int)
+Question(text : String, marks : int, number : int)
-questions
*
<<enumeration>>
QuestionType
SHORT_ANSWER
MULTIPLE_CHOICE
<<enumeration>>
SemesterType
SUMMER
WINTER
ONE
TWO
-semester
1
Subject
-id : int
-title : String
-code : String
-semester : SemesterType
-year : int
-exams : Exam
+Subject(title : String, code : String, semester : SemesterType, year : int)
+Subject(id : int, title : String, code : String, semester : SemesterType, year :
+Subject(id : int)
+getExams() : List<Exam>
+getPublishedExams() : List<Exam>
-loadExams() : void
-loadPublishedExams() : void
+getExam(index : int) : Exam
+addExam(exam : Exam) : void
+removeExam(exam : Exam) : void
+setExams(exams : List<Exam>) : void
-exams
*
Exam
-id : int
-status : ExamStatusType
-title : String
-subjectId : int
-attempted : boolean
-questions : Question
+Exam(subjectId : int, title : String)
+Exam(subjectId : int, title : String, questions : List<Question>)
+Exam(id : int, subjectId : int, status : ExamStatusType, title : String, attempted : bool
+Exam(id : int)
+Exam()
+setStatus(status : String) : void
+registerAttempt() : void
+publishExam() : void
+closeExam() : void
+addQuestion(question : Question) : void
+removeQuestion(question : Question) : void
+getQuestion(index : int) : Question
+getQuestions() : List<Question>
+setQuestions(questions : List<Question>) : void
+loadQuestions() : void
+getTotalMarks() : int
-status
1
<<enumeration>>
ExamStatusType
UNPUBLISHED
PUBLISHED
CLOSED
-exam
1
<<enumeration>>
RoleType
ADMIN
STUDENT
INSTRUCTOR
-role
1
User
-id : int
-role : RoleType
-firstName : String
-lastName : String
-email : String
+User()
+User(id : int, role : RoleType, firstName : String, lastName : String, email : Stri
+takeExam(exam : Exam) : Submission
Submission
-responses : HashMap<Integer, Response>
-exam : Exam
-userId : int
-unitOfWork : UnitOfWork
+Submission(exam : Exam, userId : int)
+getResponse(index : int) : Response
+registerResponse(questionNumber : int, text : String) : voi
+registerResponse(questionNumber : int, option : int) : voic
+markUnanswered() : void
+submit() : void
Grade
-marksAwarded : int
+Grade(marksAwarded : int)

Alternatives to the domain model

An alternative to the domain model is the transaction script – this was not utilised by the project team as the business logic was too complex for this pattern and would not allow as easily for extensibility in future.

In a project team implementation, transaction script is also not well suited as it could result in each group member implementing it in different ways. Domain model (with the use of data mapper, discussed in the next section) allowed the project team to more easily separate development while also not resulting in duplicated code.

The project team is also far more experienced developing with the domain model.

Data mapper

The data mapper pattern is a mechanism for moving data between domain objects and the database. The data mapper layer is responsible for loading database rows into domain objects and persisting these objects back into the database. It facilitates any data manipulation that may be required, such as inserting, deleting and updating rows.

Reasons for use

The benefits to this project of using a data mapper are that the domain objects can be separated from database logic. The domain layer does not need to know how to communicate with the database – this allows the project team to work concurrently on developing the domain objects (which do not need SQL logic) while others develop the SQL logic in the mappers. It also means that the database schema can be changed without impacting the domain layer, so long as the contract between the domain and data mapper remain intact.

The data mapper pattern is a natural complement for the domain model pattern and allows for greater re-use of the underlying database and domain layer. The advantage of taking this approach is that the project team develop a sophisticated domain layer that takes advantage of object oriented constructs, without this layer having to deal with the database layer itself.

Given the non-trivial domain of this system, one data mapper for all objects is unsuited. It does not support the architectural goal of extensibility, as it is hard to maintain as the domain model grows in size.

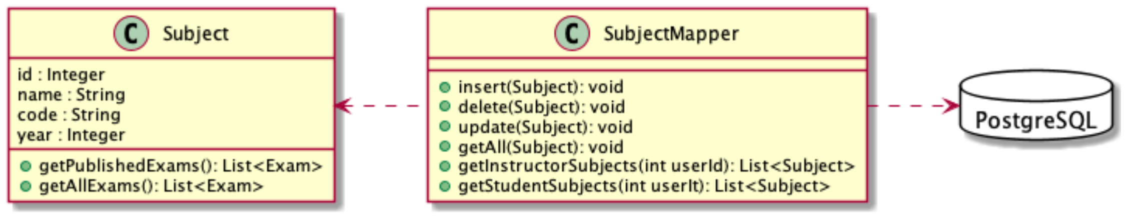
A data mapper for each database table has been created, mapping each table entry to the relevant domain objects. This more naturally supports extensibility, handles non-trivial domain logic, and allows the project team to write a larger variety of database queries in the mapper.

It also further supports the loose coupling of the system, as identified previously as an architectural goal.

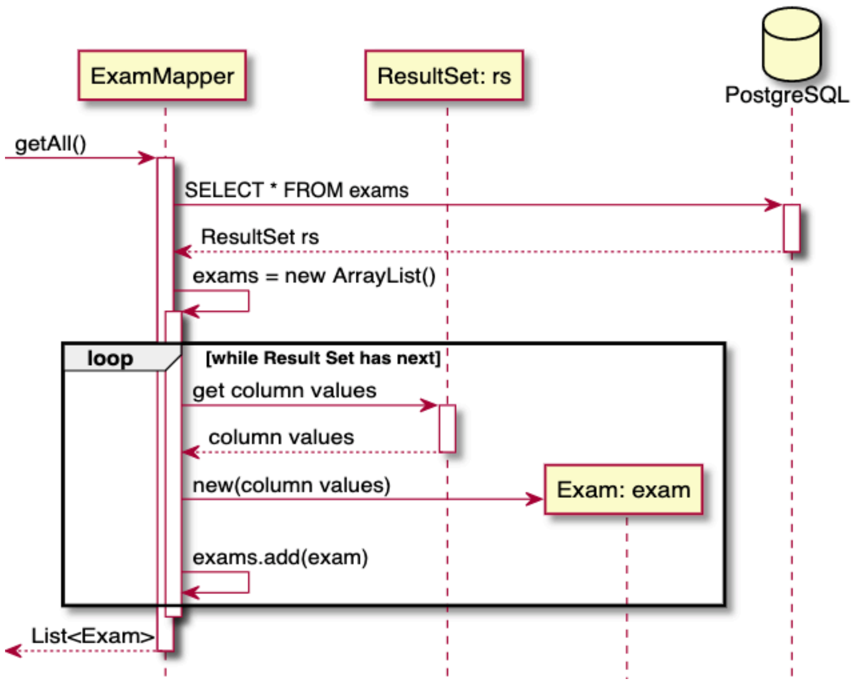
Implementation

Each data mapper has an *insert*, *update* and *delete* method which is fed an object of that type. This generic implementation aids implementation of the later mentioned Unit of Work. It also means that the calling code only needs to feed an updated domain object to the mapper.

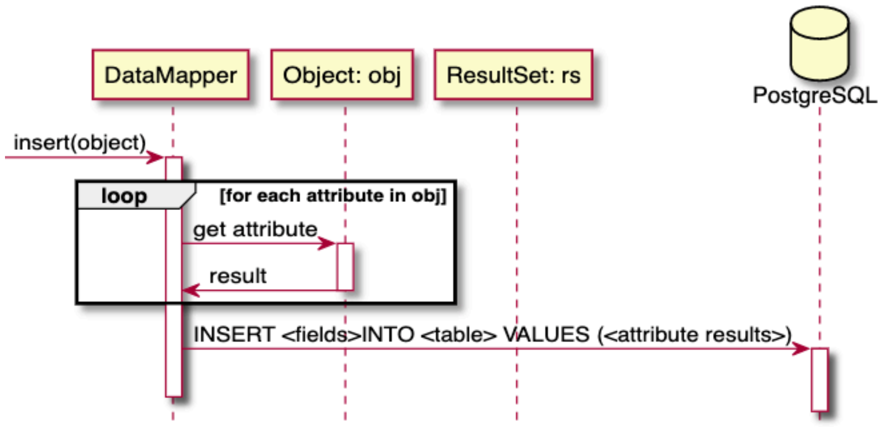
The domain objects remain mostly unaware of the data mapper's existence. The only dependence between the domain layer and data mapper exists when the domain layer has to lazy load an object, or in the unique case when a **User** has to mark an exam as attempted upon starting an exam. This means that dependency flows outwards from the data mapper to the domain objects and database, for example:

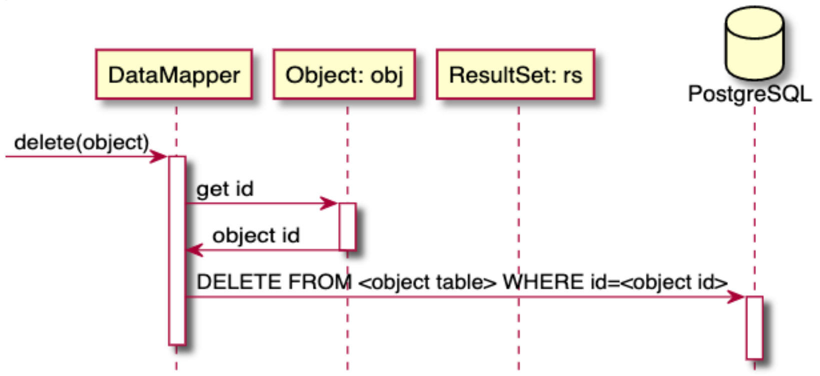


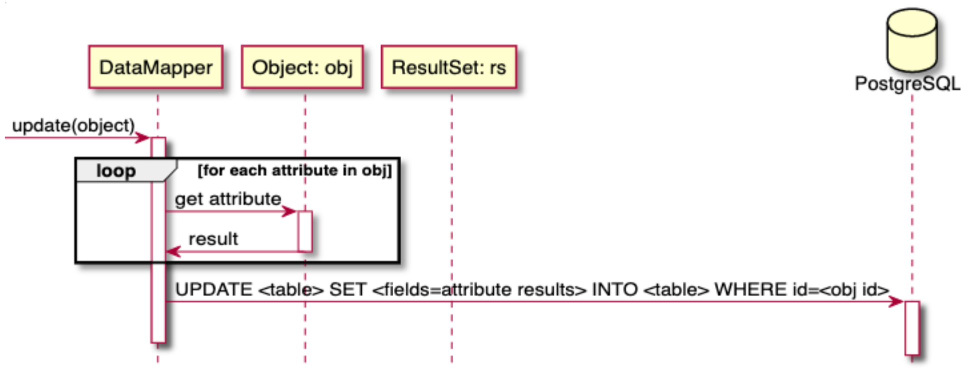
The following sequence diagram includes the flow of actions for loading data into the exam.



The generic flow for *insert(obj), update(obj)* and *delete(obj)* are as follows:

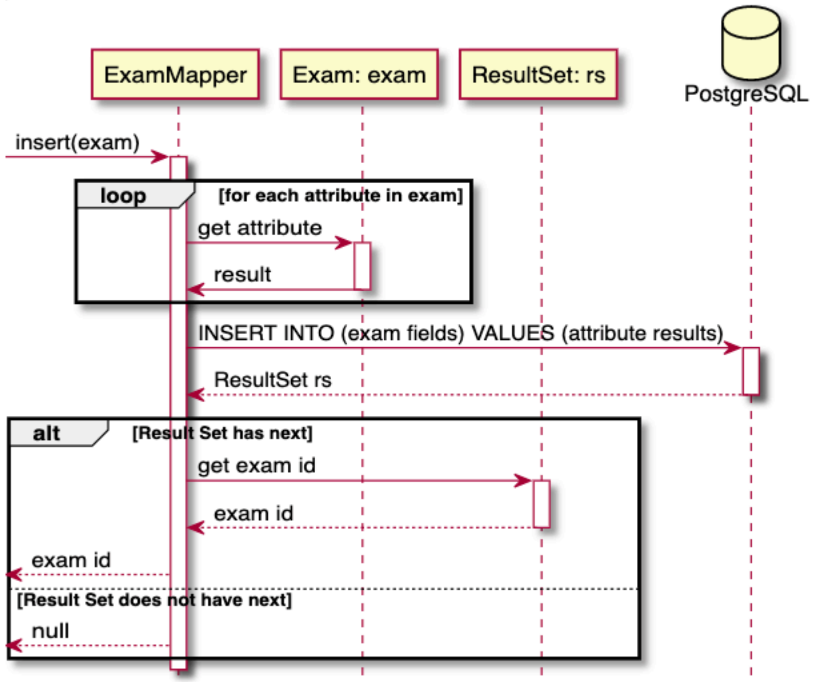

DataMapper
Object: obj
ResultSet: rs
PostgreSQL
insert(object)
loop
[for each attribute in obj]
get attribute
result
INSERT <fields>INTO <table> VALUES (<attribute results>)





DataMapper
Object: obj
ResultSet: rs
PostgreSQL
update(object)
loop
[for each attribute in obj]
get attribute
result
UPDATE <table> SET <fields=attribute results> INTO <table> WHERE id=<obj id>

This flow is mostly the same across objects, with a few special cases. First is *ExamMapper.insert*() which needs to retrieve the returned database id after insert and send this to the questions it contains, so they can be saved next to the correct exam:



Second is the *QuestionMapper*, which for Multiple Choice Questions, has an additional step of calling the *OptionMapper* to persist any changes to the options it contains.

In the cases where the Unit of Work pattern is implemented to manage a business transaction, the controller instead registers objects with the Unit of Work layer. This is detailed in the proceeding Unit of Work section.

The class diagram for the **controllers** package can be found in the appendix.

Alternatives to the data mapper
Active record pattern was not used for a number of reasons. It encourages close coupling with the database which restricts the domain modelling, in that it must more closely model the database schema. The project team has made use of inheritance in the domain model – active cases makes it significantly more difficult to use object-oriented constructs, such as inheritance. Row data gateway suffers from similar drawbacks for use in this project – it requires closer coupling with the database. It is also incompatible with domain model and so has not been used in this project. Table data gateway is similarly incompatible with domain model, and is not well suited to scalability for systems like this with complex domain logic.

Identity field
The identity field has been used to save a database ID field in an object to maintain object identity between an in-memory object and a database row.
Examples of use
The identity field has been used in four classes in the Examination Application: 1. Exam class object The exam class domain object has been given an attribute ID which represents the database key for the exam row. The key has been created to be: • Meaningless key – it is not intended for human use and has been created as a surrogate key in order to minimise problems of identifying exams. Using a meaningful key would have resulted in placing needless restrictions on title or introduce new fields like date of exam in order to differentiate exams of the same subject. Using a meaningless key was a more straightforward and workable approach; • Simple key – it is represented as an integer; • Table unique – the key is unique to the exam table (but is potentially replicated in other database tables); and • Auto-generated – the exam ID is an auto-generated serial integer created by the database at the time of insert. 2. Question class object The exam class domain object has been given an attribute ID which represents the database key for the question row. The key has been created to be: • Meaningless key – it is not intended for human use and has been created as a surrogate key in order to minimise problems of identifying questions. Using a meaningful key would have resulted in placing needless restrictions on title or introduce new fields like order of question on exam (which would have complicated re-ordering of questions and editing exams) in order to differentiate questions of the same exam. Using a meaningless key was a more straightforward and workable approach; • Simple key – it is represented as an integer;

- Table unique – the key is unique to the question table (but is potentially replicated in other database tables); and
- Auto-generated – the exam ID is an auto-generated serial integer created by the database at the time of insert.

3. Subject class object

- Meaningless key – it is not intended for human use and has been created as a surrogate key in order to minimise problems of identifying subjects. Subject code could not have been used as an identifying key as it is possible for subjects to repeat across semesters; therefore a unique key of (subject code + semester + year) would have had to have been used which would complicate database retrieval and logic surrounding subject creation. Using a meaningless key was a more straightforward and workable approach;
- Simple key – it is represented as an integer;
- Table unique – the key is unique to the subject table (but is potentially replicated in other database tables); and
- Auto-generated – the exam ID is an auto-generated serial integer created by the database at the time of insert.

4. User class object

- Meaningful key – user ID has been chosen as student or instructor university credentials. Given that university ID is expected to be unique across the university anyway, it has been taken advantage of to create a database key to represent the user;
- Simple key – it is represented as an integer; and
- Database unique – the key is unique to the user table .

Reasons of use

Identity field pattern also complements the domain model pattern used in this project – the identity fields are able to be stored in-memory and written back

While it does increase coupling between the domain layer and the data layer, this was deemed necessary in order to ensure data integrity by storing the explicit database row in-memory. This method ensure the correct objects are written back to the correct database rows.

Alternatives to the identity field

Identity field is required when using the domain model.

Foreign key mapping

The domain has been modelled with objects containing references to other objects in the domain layer. In order to persist these relationships, they are persisted in the database – simply saving the object identities is insufficient as these identities are lost between sessions. In circumstances where the relationship is one-to-one, this has been address by placing a foreign key identified in the dependent object.

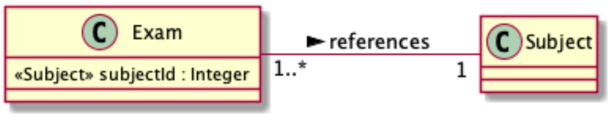
In circumstances where an objects refers to a collections of other objects (a one-to-many relationship), as relational databases can only reference single values, the foreign key has been mapped to the many side (the objects on the many side reference a single object through the use of a foreign key).

Examples of use

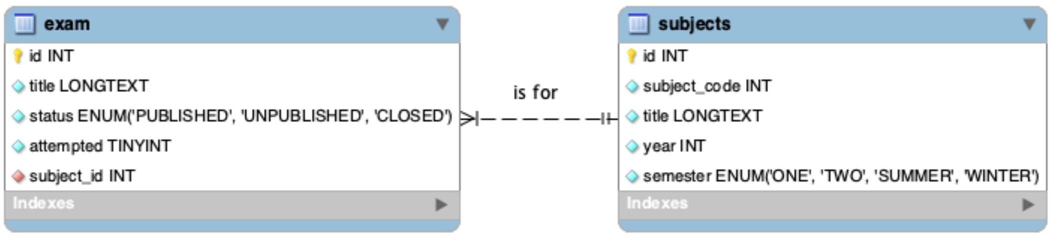
The identity field has been used in three classes in the Examination Application:

1. Exam class subject

The below shows an excerpt of the domain model for this system. The association from exam to subject is implemented as single-valued reference:



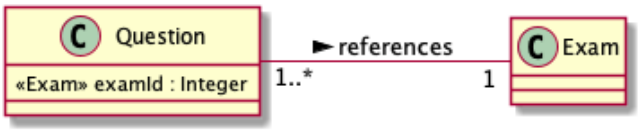
This has been modelled with a foreign key in the database:



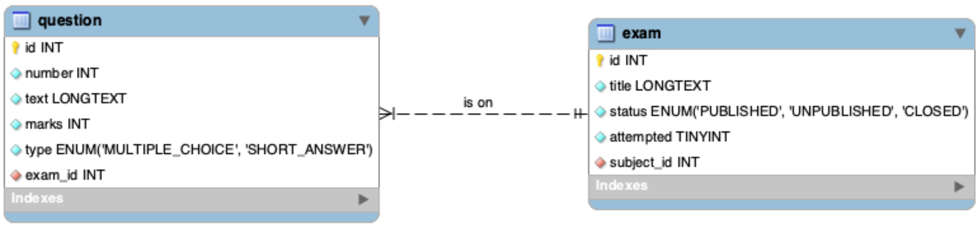
The foreign key has been mapped to the one side of the relationship between question and exam.

2. Question class object

The below shows an excerpt of domain model for this system. The association from question to exam is implemented as single-valued reference:



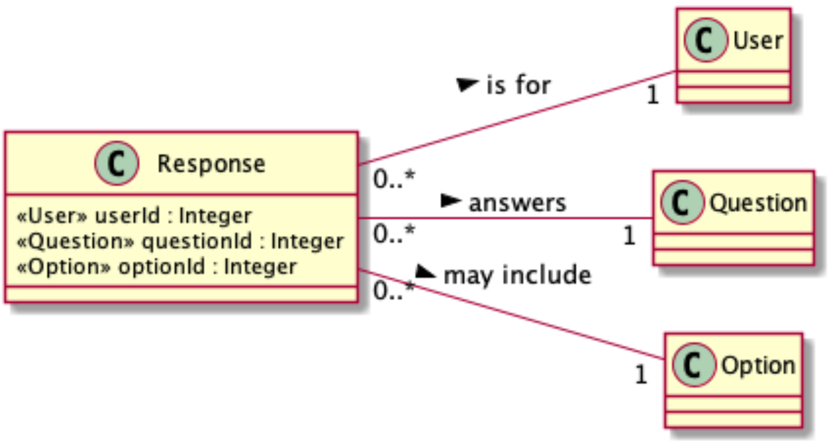
This has been modelled with a foreign key in the database:

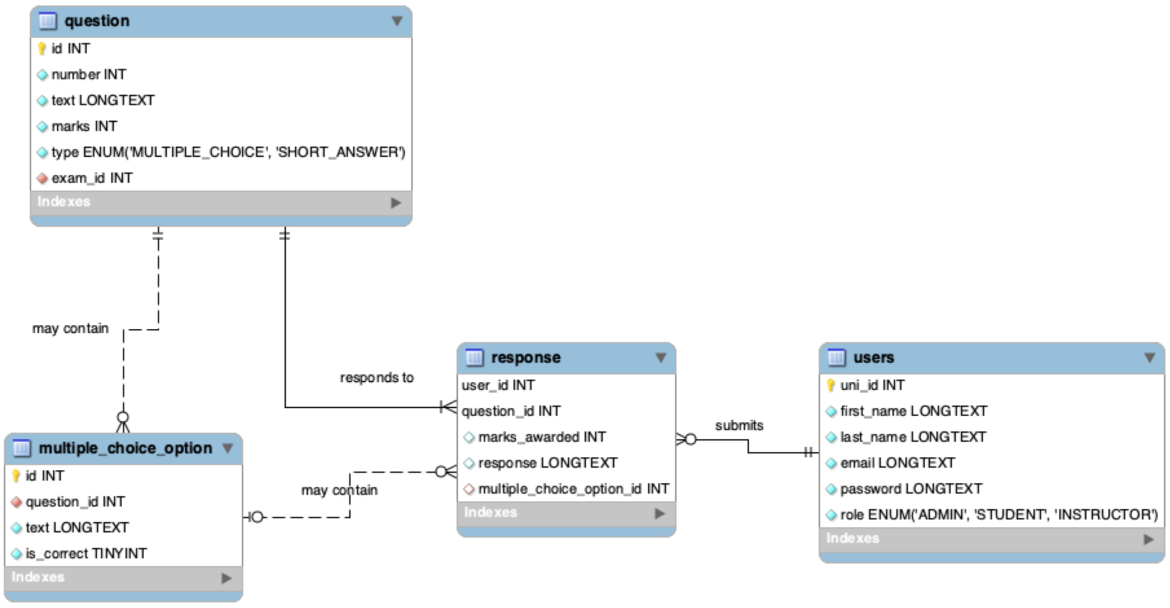


The foreign key has been mapped to the one side of the relationship between question and exam.

3. Response class object

The below shows an excerpt of domain model for this system. The association from user to response is implemented as single-valued reference:





The foreign key has been mapped to the one side of the relationship between user, question, and multiple choice option, and response.

Reasons of use

In order to track the relationship between these objects between sessions, foreign key mapping has been used to persist these relationships in the database.

Alternatives to the identity field

Association table mapping cannot be used in this instance as they are not many-to-many relationships.

Unit of Work

In essence, the Unit of Work aims to reduce the amount of data written back to the data source. After objects are read from the database, the Unit of Work entity keeps a register of which objects have been created, updated or deleted. Then, upon instruction from its calling code, the Unit of Work saves the changes to all objects in its register in a single database transaction.

Justification of use

The Unit of Work improves the efficiency of database saves when used effectively. Since a number of use cases in this application involve changing multiple database rows, across multiple tables, this efficiency was considered worth the trade off of implementation. Careful design, as well as thorough testing and code reviews were employed to avoid '*forgetfulness*' issues.

The concise and reusable object maintains the order of changes. This simplified the logic in our data mapper for persisting the creation and updating of objects which contain references to other objects.

Design Rationale

The Unit of Work is an efficient solution when a large number of objects are loaded from the database, but only a handful of changes are made. For this reason, it was chosen for implementing:

1. *An instructor marks an exam via the table view*

In this case, an instructor chooses to update the marks or 'grades' for an exam. At this point, a grade object for every student in that exam has been loaded, and any number of their grades can be updated by the instructor, before they choose to save. Since there could be a large number of students taking the exam, and the instructor may choose to only update a few grades at a time, it was deemed inefficient to commit every loaded grade back to the database. Unit of Work was instead implemented, to keep track of which grades were changed, and only committing those to the database.

Unit of Work is also efficient when the objects loaded are likely to be changed multiple times before they need to be saved back to the database. For this reason, it was chosen for implementing:

2. *An instructor edits an exam and its questions*
3. *A student takes an exam*

In the first case, the instructor can update the exam details and add, delete and update as many questions as they like before finally saving the exam. In the second case, the student may revise their responses to each question as many times as they like before finally submitting the exam. For each of these circumstances, saving every change may result in frequent redundant queries that will be soon overridden.

Another useful property of Unit of Work is that it maintains the order of changes. This was useful for the final use case, which required creating exams and the questions they referenced in the same transaction:

4. *An instructor creates an exam and its questions*

Here, each question needed to be saved with a reference to the exam it belongs to, but the exam should not be saved until the questions are added. The Unit of Work allowed this behaviour whilst maintaining referential integrity in the database.

Lastly, the objects involved in each of these four transaction are updated at a single point in time, so deferring the persistence of their changes is important to ensure consistency in the database. If one fails, the rest should rollback. This behaviour is appropriate to implement with the Unit of Work pattern.

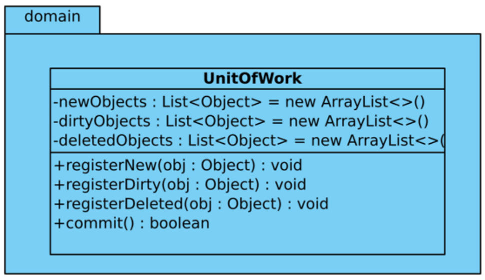
Implementation

The Unit of Work has been implemented as an object, with the new, updated and deleted objects stored in array list instance variables:

1. *newObjects*
2. *dirtyObjects*
3. *deletedObjects*

It has four methods:

1. *registerNew(Object obj)* : add object to new list
2. *registerDirty(Object obj)* : add object to dirty list
3. *registerDeleted(Object obj)* : add object to deleted list
4. *commit()* : loop through the objects in the new, dirty and deleted lists, consecutively. Send each object to the corresponding method (*insert*, *update* or *delete*) on the object's corresponding data mapper. Rollback all transactions if an exception is caught.


domain
UnitOfWork
-newObjects : List<Object> = new ArrayList<>()
-dirtyObjects : List<Object> = new ArrayList<>()
-deletedObjects : List<Object> = new ArrayList<>(
+registerNew(obj : Object) : void
+registerDirty(obj : Object) : void
+registerDeleted(obj : Object) : void
+commit() : boolean

The Unit of Work Class

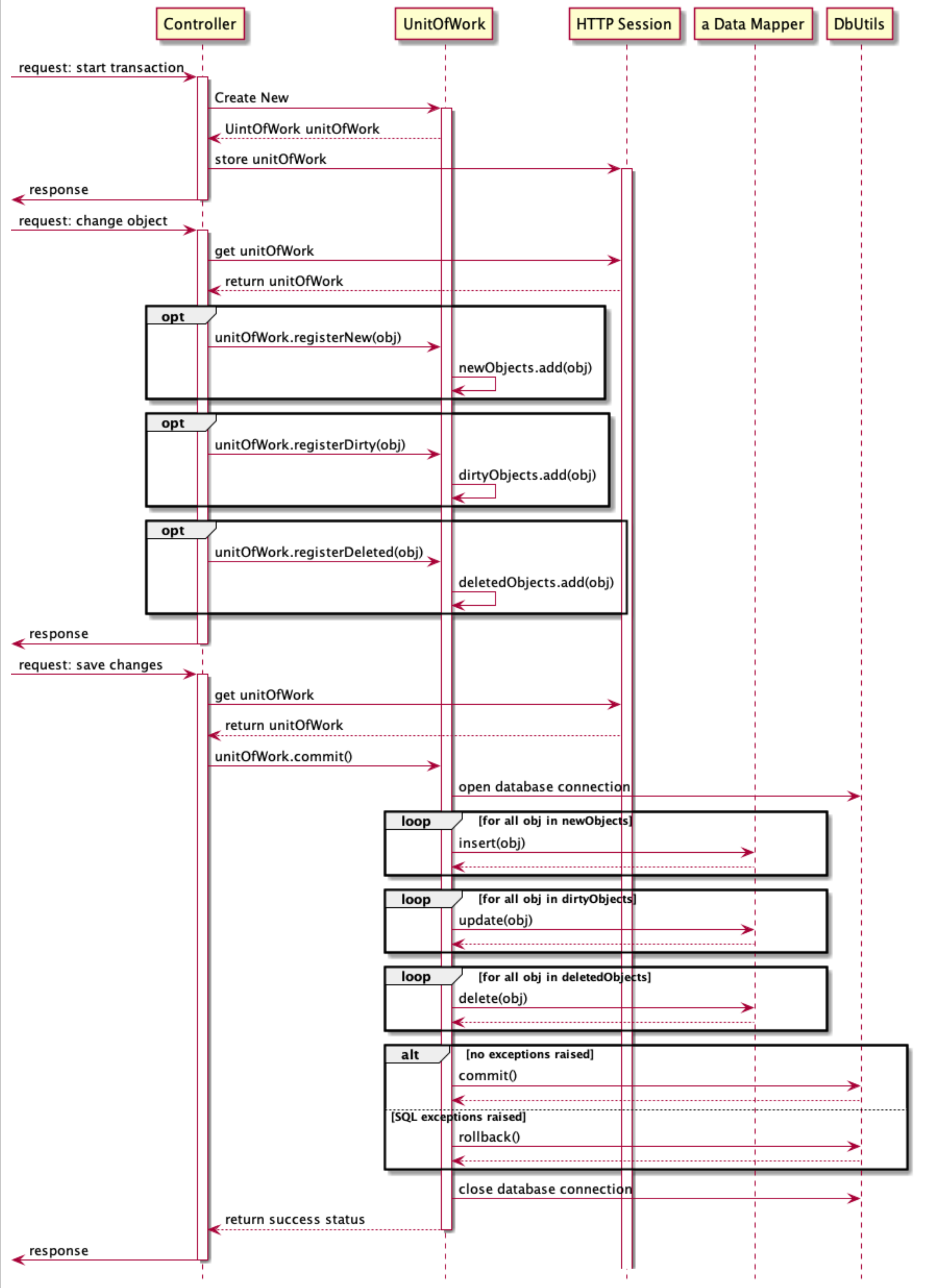
The team designed the Unit of Work to be generic and accept all objects involved in the four transactions it handles (defined above in 'Design Rationale'). This was facilitated by the choice to use the Data Mapper pattern, as each Data Mapper implements a generic *insert*, *update* and *delete* functions, accepting the relative object as the sole argument. Therefore, when *commit()* is triggered, the Unit of Work simply runs through the new, dirty, and removed lists consecutively, sending each object to the correct mapper.

The implementation uses 'caller registration', meaning the calling code is responsible for registering objects with the Unit of Work. This method was chosen due to simplicity of implementation: limiting Unit of Work registration to a single layer was found to improve code comprehension. It also helped ensure that domain objects can operate without dependence on the Unit of Work.

The sequence of interaction between the Unit of Work and the other application components is as follows:

1. The controller loads objects from the Data Mapper
2. The controller instantiates the Unit of Work object at the beginning of a business transaction
3. The controller hands control to the domain layer to execute some behaviour on its objects
4. Each update to the domain objects is registered with the Unit of Work
5. Once the transaction is complete, the controller instructs the Unit of Work to commit the transaction

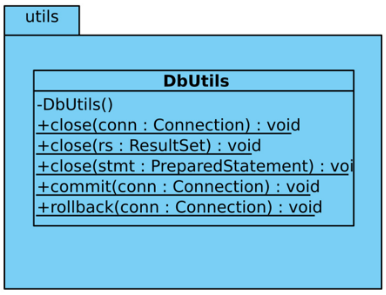
This flow generic to the four controllers that implement the unit of work: *EditExamController*, *AddQuestionController.java*, *TakeExamController.java*, and *EditGradesTableController*. The following sequence diagram represents this flow of events:



Note that for readability in the above diagram, the 'Data Mapper' represents multiple data mapper style objects. The Unit of Work will send objects to the corresponding mapper.

The Unit of Work object stores data specific to the logged in user and exists for the duration of a single business transaction. Given the complex logic of the application, the Unit of Work is also required across multiple http requests. For these reasons, it made sense to store the Unit of Work in the *HttpSession* object which is instantiated at login. At the start of each transaction, a new Unit of Work is initialised. It is then accessed and updated by various methods, and removed upon a *commit()* or when the page is navigated away from.

The Unit of Work is responsible for triggering the opening and closing of a database connection for the *commit()* transaction set. It does this by calling the respective method in DbUtils - a collection of methods created to make working with JDBC easier. If an SQL exception is raised, *commit()* triggers a rollback for every query in the set.


utils
DbUtils
-DbUtils()
+close(conn : Connection) : void
+close(rs : ResultSet) : void
+close(stmt : PreparedStatement) : voi
+commit(conn : Connection) : void
+rollback(conn : Connection) : void

The DbUtils package

Lazy Load

Lazy Load is an approach for improving the efficiency of reading data from the database. In essence, it enforces that object data only be loaded as the object is needed. Typically, lazy load is implemented by instantiating some sort of placeholder objects which are then loaded with actual data when accessed.

This concept is useful in many cases. Often, an object might be loaded that has reference to one or more objects, however the objects it references may not be required for a particular view or action. Furthermore, these objects may have further references to other objects, and so on. Loaded the data for every referenced object in these cases may prove wasteful.

Design Rationale

Lazy load is useful when a related object requires an extra database call, and the related object is not presently needed. In the complex domain logic of the project application, some objects have references to a collection of objects, which reference further objects. Since data is loaded from the data source with each view, and only a subset of data is generally accessed to populate a single view, lazy loading data was identified as an important tool for improving efficiency.

There were two cases where lazy load was implemented:

1. *List<Questions>* in **Exam**
2. *List<Exam>* in **Subject**

In both instances, the main class contains a reference to a list of referenced objects. When the main class is first loaded, it is loaded in a list and only requires a few details contained mostly in the same database row. However, the list of objects it references requires an extra database call. The referenced list may be large, and the application only ever needs to retrieve the objects in this list for one main object at a time. Therefore, loading all of the referenced objects was deemed wasteful. Instead, the list is initialised to '*null'*, and only loaded upon access.

More specifically, in the 'SubjectsController' and 'ExamsContoller', a list of Subjects or Exams is loaded into the view, respectively. At this point, the related Exams or Questions are not used and need not be loaded. Only upon selecting an action on a particular Subject or Exam, will these details be needed.

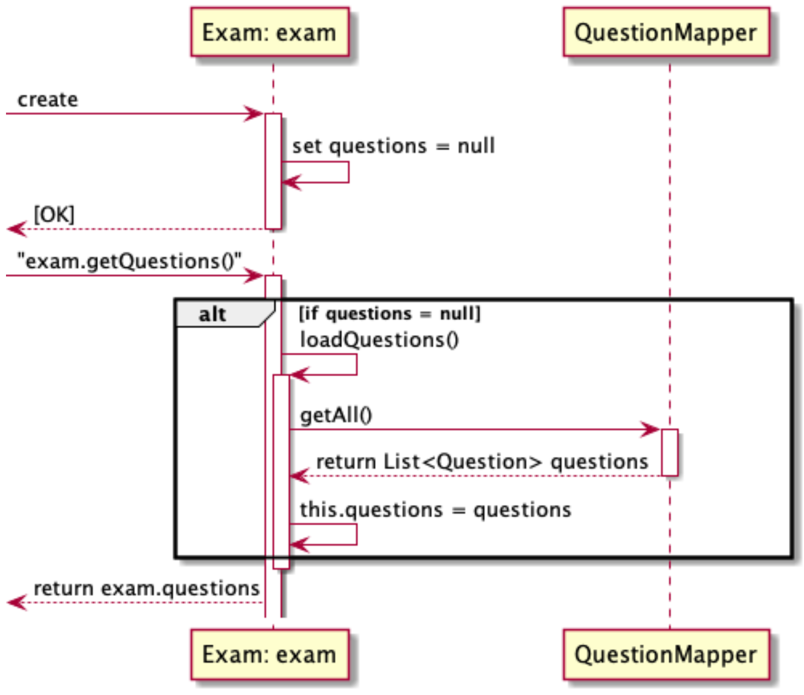
In another example, the 'TakeExamController' first loads the exam and displays its basic details, before the user can choose to start the exam if they are permitted, or view a previous submission otherwise. In many cases, the exam will not be started. The questions associated with the exam need not be loaded until the exam is started, so loading these exams upon loading the exam is unnecessary.

Implementation

Lazy load was implemented using *lazy initialization.* This approach was chosen for its simplicity and was deemed safe since '*null'* is not a valid value for any of the lazy loaded fields.

The references *List<Questions>* in **Exam**, and *List<Exam>* in **Subject** and initialised to null. Upon access via *Exam.getQuestions()* or *Subject.getExams()*, if the list is null, the access method calls a *loadQuestions()* or *loadExams()* which call the appropriate data mapper to load the objects into the list.

The flow can be visualised as follows for Exams:



Association table mapping

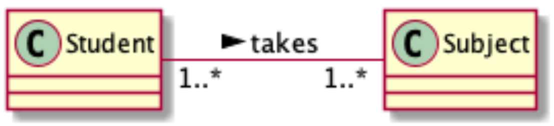
Association table mapping is a pattern to handle many-to-many associations in a database.

Examples of use

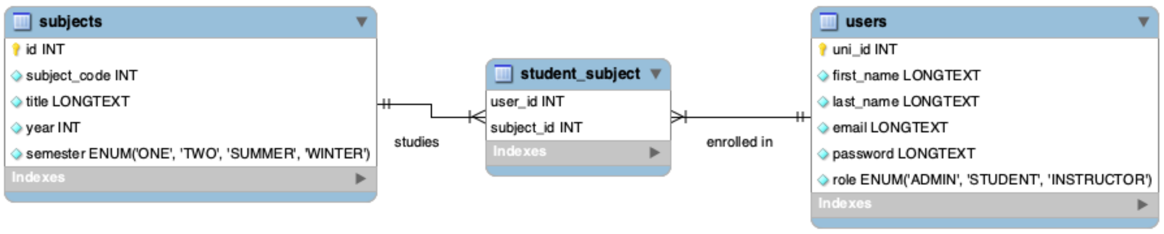
This pattern has been utilised in two places – to model the many-to-many relationships between students and subjects and instructors and subjects.

1. Students may be enrolled in one or more subjects and subjects may have one or more students enrolled.

The relationship has been modelled in the domain model as a many-to-many relationship between students and subjects:

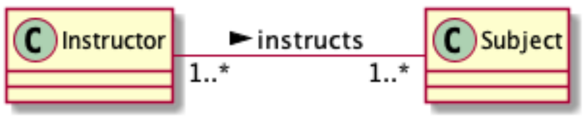


Mapping this using the association table mapping pattern has resulted in the table design shown below. A new student_subject table records the relationship and consists only of pairs of foreign keys:

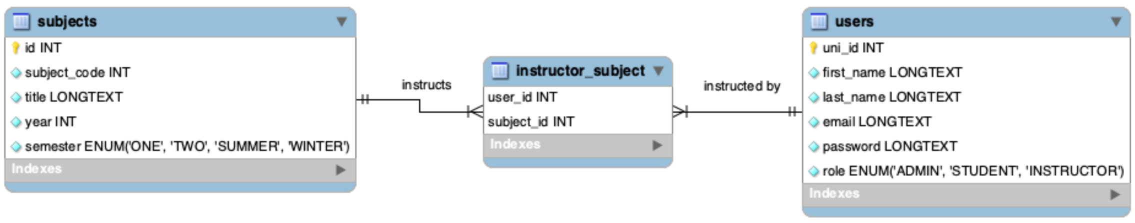


2. Instructors may instruct one or more subjects and subjects may have one or more instructors (concurrency amongst instructors transacting on the same exam object is not currently supported, please see constraints in section 3.2).

The relationship has been modelled in the domain model as a many-to-many relationship between instructors and subjects:



Mapping this using the association table mapping pattern has resulted in the table design shown below. A new instructor_subject table records the relationship and consists only of pairs of foreign keys:



To query the association tables of student_subject and instructor_subject, the UserMapper and SubjectMapper classes have been extended to issue a query to either table. Please see one example below, however, there are more methods using the association pattern in both UserMapper and SubjectMapper:

```
  */
public static boolean insertInstructor(int uniId, int subjectId) {
    String sql = "INSERT INTO instructor_subject(user_id, subject_id) VALUES (?, ?)";
    PreparedStatement insertStatement = null;
    boolean inserted = false;
    Connection connection = null;
```

Reasons of use

This pattern has been used as a way of enforcing the many-to-many relationships of the objects and the simplicity of it allows for mechanical mapping between domain objects and database tables.

Alternatives to the identity field

Association table mapping has been used in place of foreign key mapping, as foreign key mapping does not support many-to-many relationships like this where there is not just a single value for subject enrolments of a student or instructor.

Embedded value

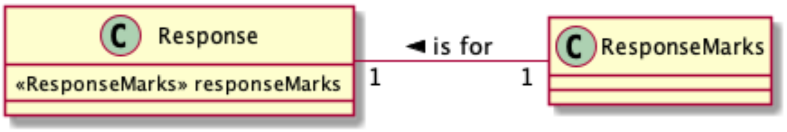
Embedded value is a pattern to represent objects in the domain model that do not make sense to map to their own table in a database schema. An embedded value maps the values of an object into the fields of its owner. When the owning object is loaded/saved, the corresponding embedded values are loaded/saved as well.

Examples of use

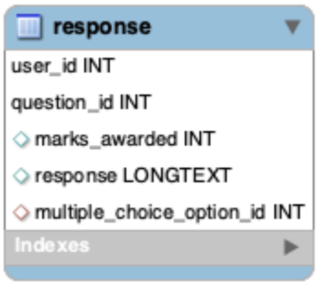
An object class ResponseMarks has been created to display and store the marks given for a student response.

When saving to the database, the value objects are mapped to the response table and when the owning object, Response, is loaded, the values are loaded/saved into ResponseMarks also.

Below is an excerpt of the domain model showing the mapping between Response and ReponseMarks.

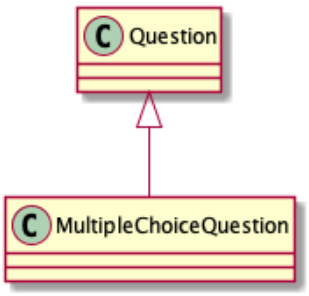
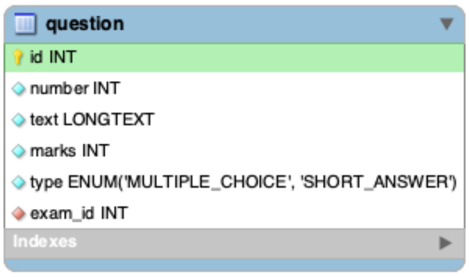


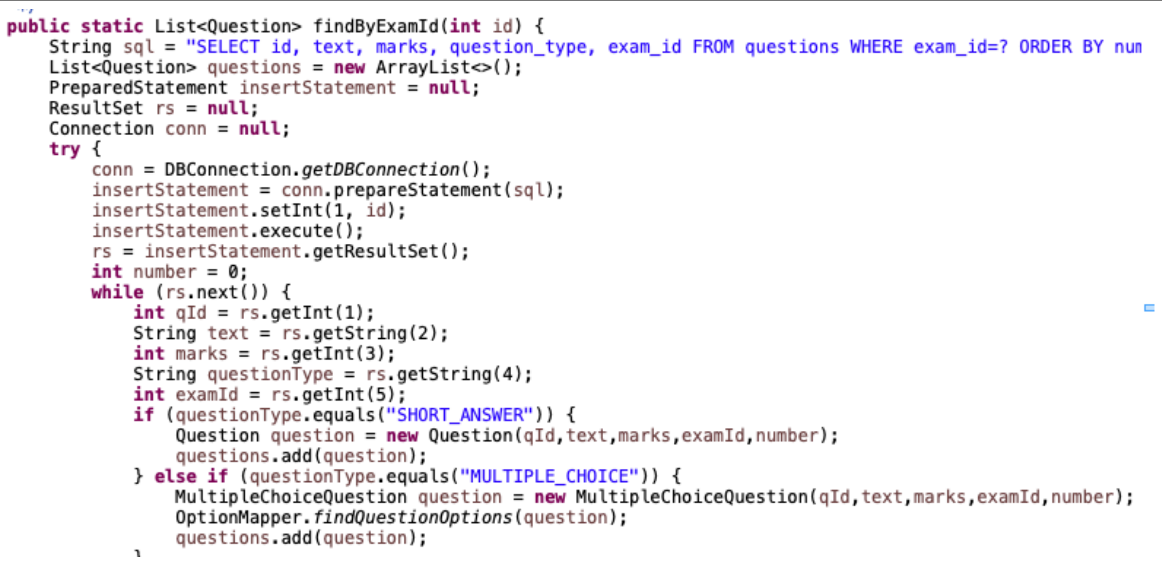
This has been represented in the database schema as a singular table:



response
user_id INT
question_id INT
◇ marks_awarded INT
◇ response LONGTEXT
◇ multiple_choice_option_id INT
Indexes

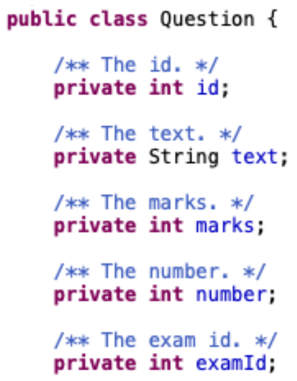
Reasons of use
The embedded value pattern has been used as both objects are always loaded and saved at the same time and the relationship between them is only ever one-to-one. Using the embedded value pattern has resulted in a much neater relational database schema and has resulted in the schema being much closer to the way it would be designed independent of the object-oriented domain model.

Single table inheritance
Single table inheritance pattern represents an inheritance hierarchy of classes as a single table that has columns for all fields of the various classes
Examples of use
Single table inheritance has been used to deal with structural mapping of inheritance hierarchies to relational models, as inheritance is not supported by relational databases. It has been used in order to preserve the inheritance hierarchy of Questions and Multiple Choice Questions:  The single table inheritance pattern maps all fields/attributes of both Question and Multiple Choice Question classes in the hierarchy into a single table, recording the type of each object, and leaving null/empty all the fields in row that are not in the corresponding object:  question id INT number INT text LONGTEXT marks INT type ENUM('MULTIPLE_CHOICE', 'SHORT_ANSWER') exam_id INT Indexes When loading a row from the Question table into an in-memory object (either a Question object or Multiple Choice Question 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:



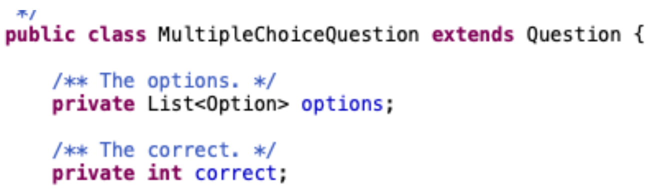
```
public static List<Question> findByExamId(int id) {
    String sql = "SELECT id, text, marks, question_type, exam_id FROM questions WHERE exam_id=? ORDER BY num
    List<Question> questions = new ArrayList<>();
    PreparedStatement insertStatement = null;
    ResultSet rs = null;
    Connection conn = null;
    try {
        conn = DBConnection.getDBConnection();
        insertStatement = conn.prepareStatement(sql);
        insertStatement.setInt(1, id);
        insertStatement.execute();
        rs = insertStatement.getResultSet();
        int number = 0;
        while (rs.next()) {
            int qId = rs.getInt(1);
            String text = rs.getString(2);
            int marks = rs.getInt(3);
            String questionType = rs.getString(4);
            int examId = rs.getInt(5);
            if (questionType.equals("SHORT_ANSWER")) {
                Question question = new Question(qId,text,marks,examId,number);
                questions.add(question);
            } else if (questionType.equals("MULTIPLE_CHOICE")) {
                MultipleChoiceQuestion question = new MultipleChoiceQuestion(qId,text,marks,examId,number);
                OptionMapper.findQuestionOptions(question);
                questions.add(question);
```

The Question object class records the following attributes into an in-memory object:



```
public class Question {

    /** The id. */
    private int id;

    /** The text. */
    private String text;

    /** The marks. */
    private int marks;

    /** The number. */
    private int number;

    /** The exam id. */
    private int examId;
```

While the Multiple Choice Question object class records different attributes into an in-memory object:



```
 */
public class MultipleChoiceQuestion extends Question {

    /** The options. */
    private List<Option> options;

    /** The correct. */
    private int correct;
```

The class Question records most attributes of the Question row, while the lower-level Multiple Choice Question object class records attributes a list of options and a Boolean to represent which option is correct, both of which are attributes specific to multiple choice questions.

Reasons of use

The single table inheritance has been chosen due to its simplicity in storing questions – it minimises duplication in the database and increases stability and supports extensibility. Any changes in future to the objects require only a refactoring of code and not necessarily any changes to the database table.

This pattern also minimises the number of joins performed by the database – joins are an operationally taxing operations for databases, especially with large datasets.

It also saves having to use one object class and set a number of different attributes to null.
Alternatives to the single table inheritance
Class table inheritance was not chosen as it increases joins the database needs to perform and anu future extensibility of the application (one of the goals of this architecture) would like require database schema changes. Concrete table inheritance similarly does not support extensibility as well as single table inheritance. It is not possible in this implementation to know the question that is about to be loaded on an exam – therefore separate concrete database tables for the two types of questions would have required global finds across both tables, resulting in extra processing and querying time.

6.2 Libraries and *Frameworks*

This section describes libraries and *frameworks* used by the Examination Application.

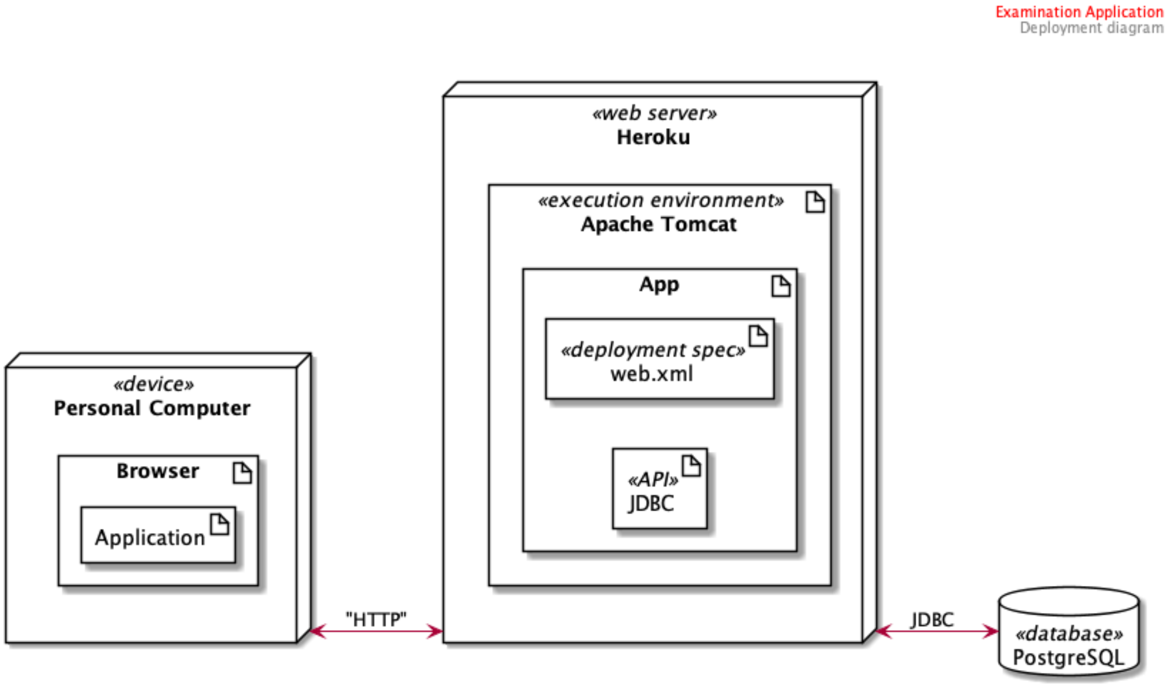
Library / *Framework*	Reason	Version
jQuery	• *This library is used to simplify Javascript code within the application, selected due to team member familiarity*	3.3.1
JavaServer Standard Tag Library (JSTL)	• *A set of tag libraries recommended for use by Oracle in its 'JSP Coding Conventions to reduce the need for JSP Scriptlets*	*N/A*

7. Physical View

This section describes the hardware elements of the system and the mapping between them and the software elements.

7.1 Production Environment

This section describes the production environment of the system and the mapping between the software elements and the available hardware. The figure below llustrates the physical view of the production environment.



7.1.1 Hardware

- Heroku PostgreSQL, is a Database as a Service (DaaS) used to persist data;
- Heroku, is a Platform as a Service (PaaS) used to deploy the Examination Application

7.1.2 Software

- Java Database Connectivity has been used as an Application Programming Interface (API) to define access to the database.
- Apache Tomcat has been used as the execution environment.

7.2 Development Environment

- Eclipse was primarily used as the Integrated Development Environment (IDE) for the Examination Application.

8. Data View

The database has been designed in order to support the architectural requirements and patterns in this report.

8.1 Database schema

Please see the Appendix for a database schema diagram and a database schema SQL creation script.

8.2 Test data

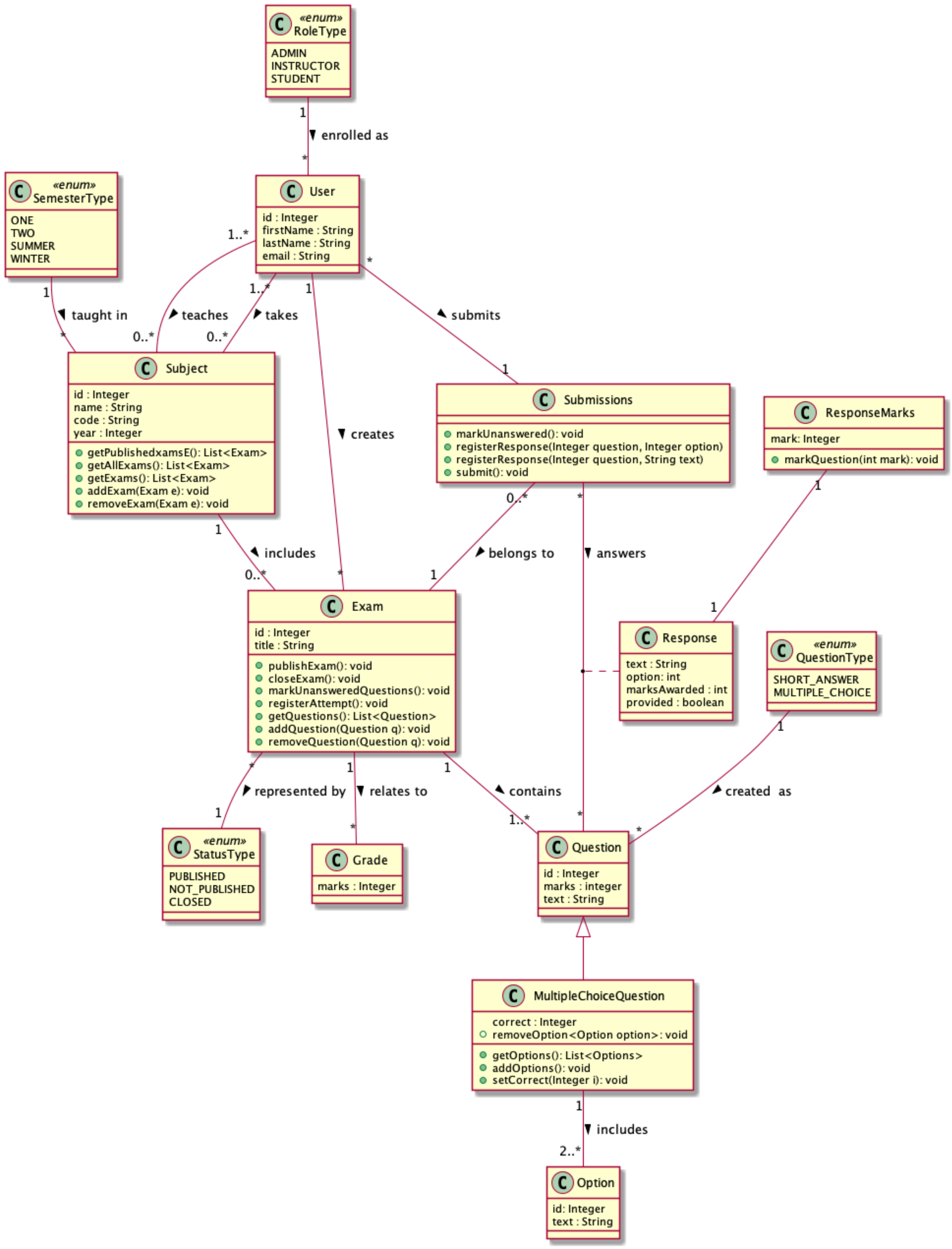
For test cases and data, please see Part 2 testing document.

9. References

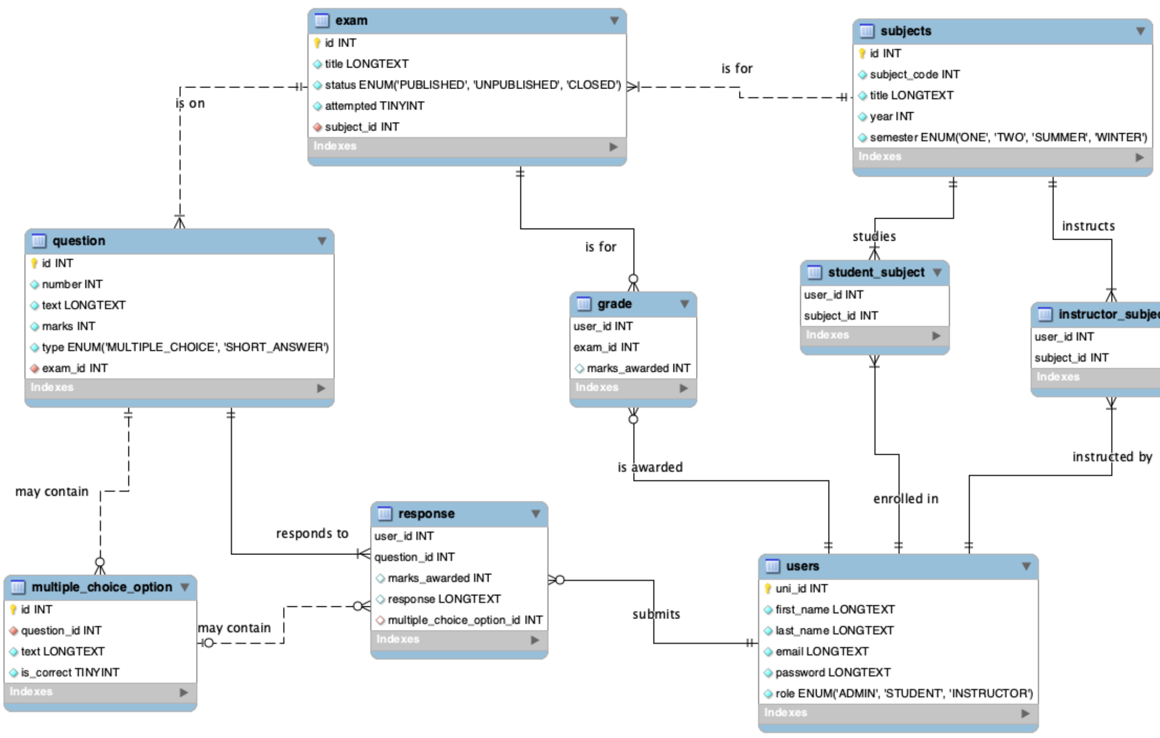
[1] *Blowfish (cipher)*. En.wikipedia.org. (2020). Retrieved 26 September 2020, from https://en.wikipedia.org/wiki/Blowfish_(cipher).

[2] Kruchten, P. B. (1995). The 4+1 view model of architecture. *IEEE software*, *12*(6), 42-50..

10. Appendix



Domain model

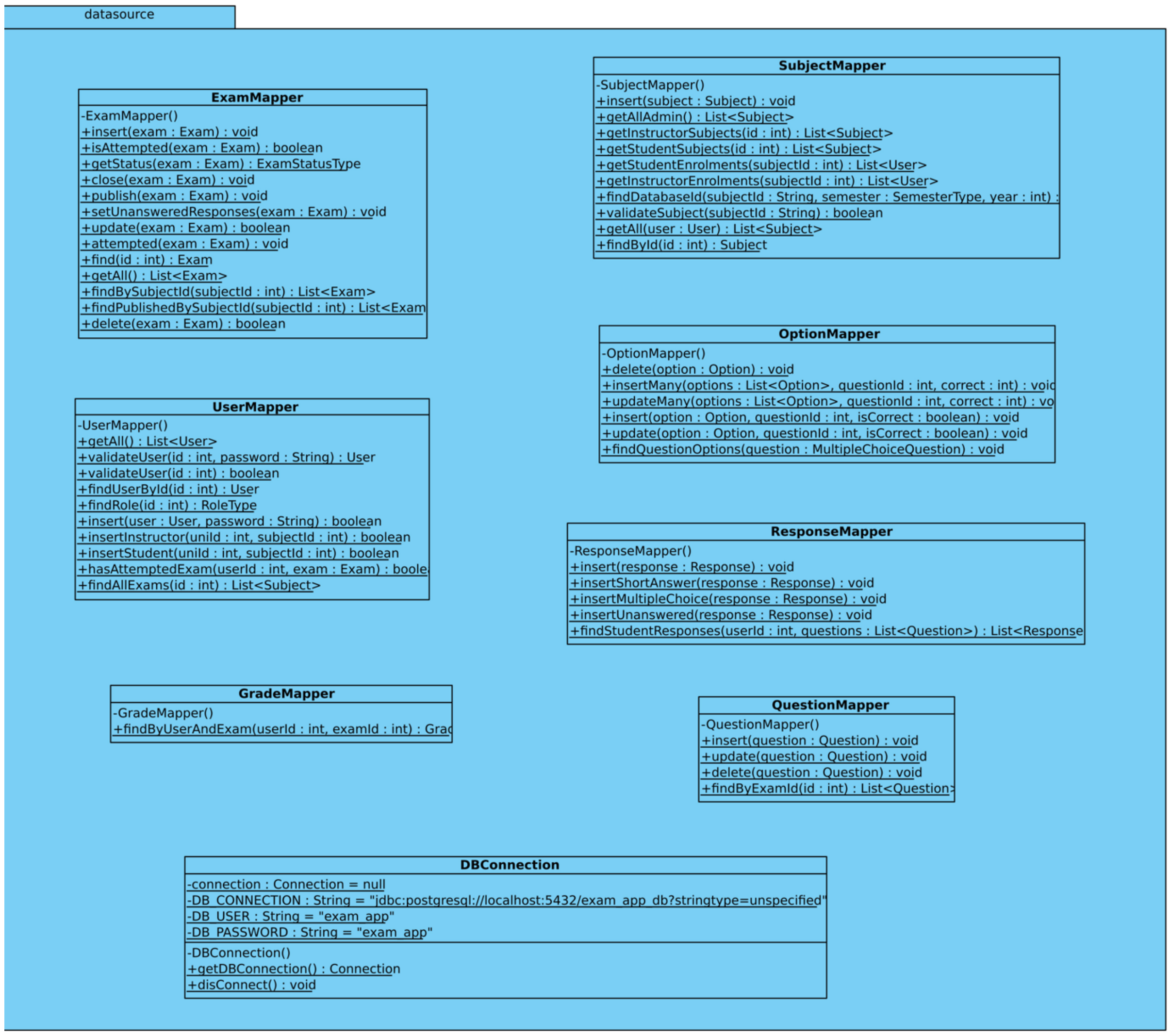


Database schema

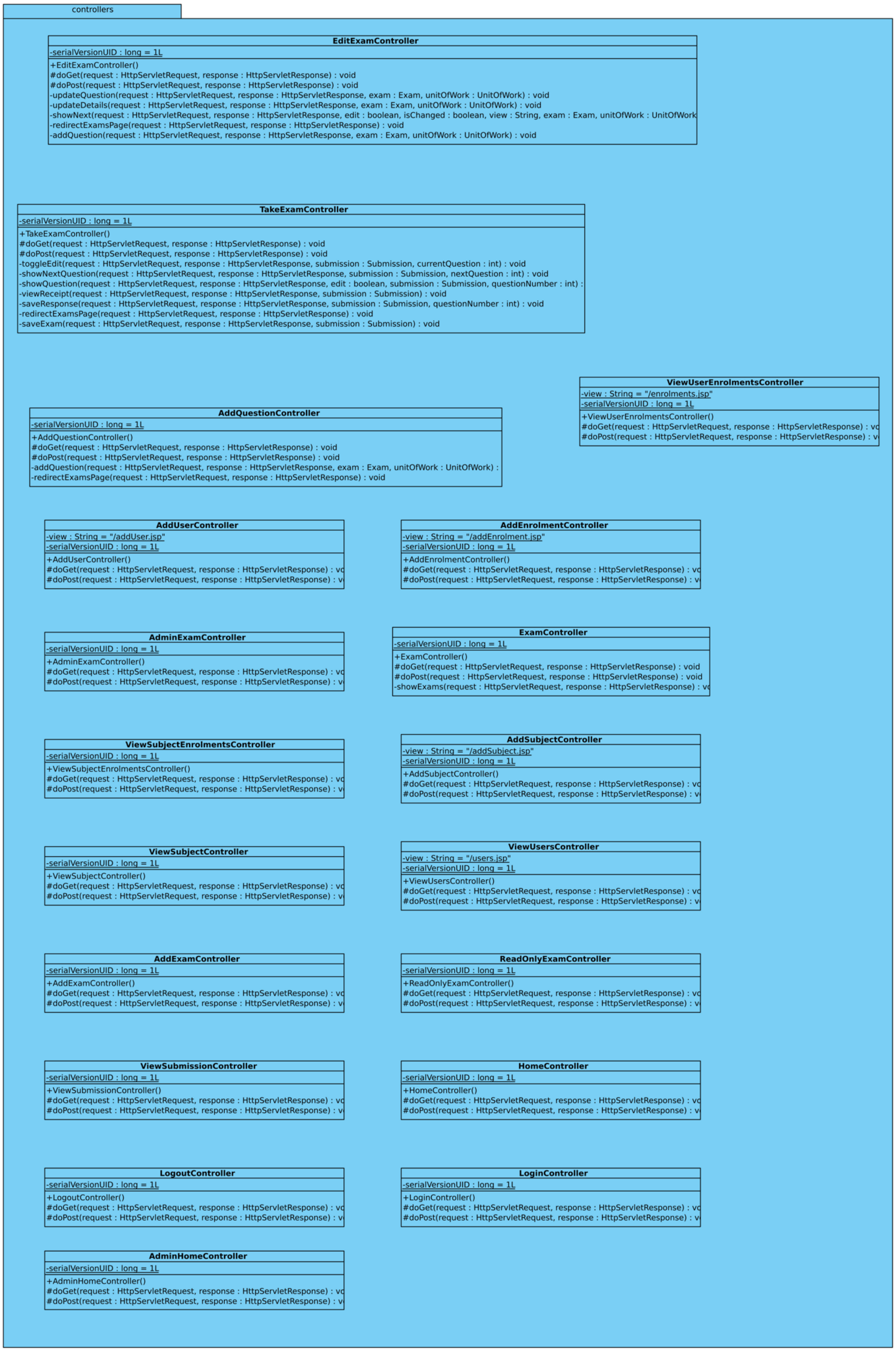
```
CREATE EXTENSION IF NOT EXISTS pgcrypto;

CREATE TYPE ROLE_TYPE AS ENUM('ADMIN', 'STUDENT', 'INSTRUCTOR');

CREATE TABLE IF NOT EXISTS users (
	uni_id INT,
	first_name TEXT NOT NULL,
	last_name TEXT NOT NULL,
	email TEXT NOT NULL UNIQUE,
	password TEXT NOT NULL,
	role ROLE_TYPE NOT NULL,
	PRIMARY KEY (uni_id)
);

CREATE TYPE SEMESTER_TYPE AS ENUM ('ONE', 'TWO', 'SUMMER',
'WINTER');

CREATE TABLE subjects (
	id BIGSERIAL,
	title TEXT NOT NULL,
	code TEXT NOT NULL,
	semester SEMESTER_TYPE NOT NULL,
	year INT NOT NULL,
	PRIMARY KEY(id),
	UNIQUE (code, semester, year)
);

CREATE TABLE IF NOT EXISTS instructor_subject (
	user_id INT NOT NULL
```

```
          REFERENCES users(uni_id)
          ON UPDATE CASCADE
          ON DELETE CASCADE,
     subject_id INT NOT NULL
          REFERENCES subjects(id)
          ON UPDATE CASCADE
          ON DELETE CASCADE,
     PRIMARY KEY (user_id, subject_id)
);

CREATE TABLE IF NOT EXISTS student_subject (
     user_id INT NOT NULL
          REFERENCES users(uni_id)
          ON UPDATE CASCADE
          ON DELETE CASCADE,
     subject_id INT NOT NULL
          REFERENCES subjects(id)
          ON UPDATE CASCADE
          ON DELETE CASCADE,
  PRIMARY KEY (user_id, subject_id)
);

CREATE TYPE EXAM_STATUS_TYPE AS ENUM('PUBLISHED', 'UNPUBLISHED',
'CLOSED');

CREATE TABLE IF NOT EXISTS exams (
  id BIGSERIAL,
  subject_id INT NOT NULL
     REFERENCES subjects(id)
     ON DELETE CASCADE
    ON UPDATE CASCADE,
  status EXAM_STATUS_TYPE NOT NULL
     DEFAULT 'UNPUBLISHED',
  title TEXT NOT NULL,
  attempted BOOLEAN DEFAULT 'false',
  PRIMARY KEY (id)
);

CREATE TYPE QUESTION_TYPE AS ENUM('MULTIPLE_CHOICE',
'SHORT_ANSWER');

CREATE TABLE IF NOT EXISTS questions (
  id BIGSERIAL,
  number INT NOT NULL,
  text TEXT NOT NULL,
  marks INT NOT NULL,
  question_type question_type NOT NULL,
  exam_id INT NOT NULL
     REFERENCES exams(id)
     ON DELETE CASCADE
    ON UPDATE CASCADE,
  PRIMARY KEY (id),
  UNIQUE(id, number)
```

```
);

CREATE TABLE IF NOT EXISTS multiple_choice_options (
  id BIGSERIAL,
  question_id INT NOT NULL
      REFERENCES questions(id)
      ON DELETE CASCADE
    ON UPDATE CASCADE,
  text TEXT NOT NULL,
  is_correct BOOLEAN NOT NULL,
  PRIMARY KEY (id)
);

CREATE TABLE IF NOT EXISTS responses (
  user_id INT NOT NULL
      REFERENCES users(uni_id)
      ON DELETE CASCADE
    ON UPDATE CASCADE,
  question_id INT NOT NULL
      REFERENCES questions(id)
      ON DELETE CASCADE
    ON UPDATE CASCADE,
  marks_awarded INT,
  response TEXT,
  multiple_choice_option_id INT
      REFERENCES multiple_choice_options(id)
      ON DELETE CASCADE
    ON UPDATE CASCADE,
  PRIMARY KEY (user_id, question_id)
);

CREATE TABLE IF NOT EXISTS grades (
  user_id INT NOT NULL
      REFERENCES users(uni_id)
      ON DELETE CASCADE
    ON UPDATE CASCADE,
  exam_id INT NOT NULL
      REFERENCES exams(id)
      ON DELETE CASCADE
    ON UPDATE CASCADE,
  marks_awarded INT,
  PRIMARY KEY (user_id, exam_id)
);
```


datasource
ExamMapper
-ExamMapper()
+insert(exam : Exam) : void
+isAttempted(exam : Exam) : boolean
+getStatus(exam : Exam) : ExamStatusType
+close(exam : Exam) : void
+publish(exam : Exam) : void
+setUnansweredResponses(exam : Exam) : void
+update(exam : Exam) : boolean
+attempted(exam : Exam) : void
+find(id : int) : Exam
+getAll() : List<Exam>
+findBySubjectId(subjectId : int) : List<Exam>
+findPublishedBySubjectId(subjectId : int) : List<Exam
+delete(exam : Exam) : boolean
SubjectMapper
-SubjectMapper()
+insert(subject : Subject) : void
+getAllAdmin() : List<Subject>
+getInstructorSubjects(id : int) : List<Subject>
+getStudentSubjects(id : int) : List<Subject>
+getStudentEnrolments(subjectId : int) : List<User>
+getInstructorEnrolments(subjectId : int) : List<User>
+findDatabaseId(subjectId : String, semester : SemesterType, year : int) :
+validateSubject(subjectId : String) : boolean
+getAll(user : User) : List<Subject>
+findById(id : int) : Subject
OptionMapper
-OptionMapper()
+delete(option : Option) : void
+insertMany(options : List<Option>, questionId : int, correct : int) : voic
+updateMany(options : List<Option>, questionId : int, correct : int) : vo
+insert(option : Option, questionId : int, isCorrect : boolean) : void
+update(option : Option, questionId : int, isCorrect : boolean) : void
+findQuestionOptions(question : MultipleChoiceQuestion) : void
UserMapper
-UserMapper()
+getAll() : List<User>
+validateUser(id : int, password : String) : User
+validateUser(id : int) : boolean
+findUserById(id : int) : User
+findRole(id : int) : RoleType
+insert(user : User, password : String) : boolean
+insertInstructor(unild : int, subjectId : int) : boolean
+insertStudent(unild : int, subjectId : int) : boolean
+hasAttemptedExam(userId : int, exam : Exam) : boole
+findAllExams(id : int) : List<Subject>
ResponseMapper
-ResponseMapper()
+insert(response : Response) : void
+insertShortAnswer(response : Response) : void
+insertMultipleChoice(response : Response) : void
+insertUnanswered(response : Response) : void
+findStudentResponses(userId : int, questions : List<Question>) : List<Response
GradeMapper
-GradeMapper()
+findByUserAndExam(userId : int, examId : int) : Grad
QuestionMapper
-QuestionMapper()
+insert(question : Question) : void
+update(question : Question) : void
+delete(question : Question) : void
+findByExamId(id : int) : List<Question>
DBConnection
-connection : Connection = null
-DB_CONNECTION : String = "jdbc:postgresql://localhost:5432/exam_app_db?stringtype=unspecified"
-DB_USER : String = "exam_app"
-DB_PASSWORD : String = "exam_app"
-DBConnection()
+getDBConnection() : Connection
+disConnect() : void

Class Diagram for the ***datasource*** *package*


controllers
EditExamController
-serialVersionUID : long = 1L
+EditExamController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : void
#doPost(request : HttpServletRequest, response : HttpServletResponse) : void
-updateQuestion(request : HttpServletRequest, response : HttpServletResponse, exam : Exam, unitOfWork : UnitOfWork) : void
-updateDetails(request : HttpServletRequest, response : HttpServletResponse, exam : Exam, unitOfWork : UnitOfWork) : void
-showNext(request : HttpServletRequest, response : HttpServletResponse, edit : boolean, isChanged : boolean, view : String, exam : Exam, unitOfWork : UnitOfWork
-redirectExamsPage(request : HttpServletRequest, response : HttpServletResponse) : void
-addQuestion(request : HttpServletRequest, response : HttpServletResponse, exam : Exam, unitOfWork : UnitOfWork) : void
TakeExamController
-serialVersionUID : long = 1L
+TakeExamController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : void
#doPost(request : HttpServletRequest, response : HttpServletResponse) : void
-toggleEdit(request : HttpServletRequest, response : HttpServletResponse, submission : Submission, currentQuestion : int) : void
-showNextQuestion(request : HttpServletRequest, response : HttpServletResponse, submission : Submission, nextQuestion : int) : void
-showQuestion(request : HttpServletRequest, response : HttpServletResponse, edit : boolean, submission : Submission, questionNumber : int) :
-viewReceipt(request : HttpServletRequest, response : HttpServletResponse, submission : Submission) : void
-saveResponse(request : HttpServletRequest, response : HttpServletResponse, submission : Submission, questionNumber : int) : void
-redirectExamsPage(request : HttpServletRequest, response : HttpServletResponse) : void
-saveExam(request : HttpServletRequest, response : HttpServletResponse, submission : Submission) : void
ViewUserEnrolmentsController
-view : String = "/enrolments.jsp"
-serialVersionUID : long = 1L
+ViewUserEnrolmentsController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : vo
#doPost(request : HttpServletRequest, response : HttpServletResponse) : v
AddQuestionController
-serialVersionUID : long = 1L
+AddQuestionController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : void
#doPost(request : HttpServletRequest, response : HttpServletResponse) : void
-addQuestion(request : HttpServletRequest, response : HttpServletResponse, exam : Exam, unitOfWork : UnitOfWork) :
-redirectExamsPage(request : HttpServletRequest, response : HttpServletResponse) : void
AddUserController
-view : String = "/addUser.jsp"
-serialVersionUID : long = 1L
+AddUserController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : vo
#doPost(request : HttpServletRequest, response : HttpServletResponse) : v
AddEnrolmentController
-view : String = "/addEnrolment.jsp"
-serialVersionUID : long = 1L
+AddEnrolmentController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : vo
#doPost(request : HttpServletRequest, response : HttpServletResponse) : v
AdminExamController
-serialVersionUID : long = 1L
+AdminExamController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : vo
#doPost(request : HttpServletRequest, response : HttpServletResponse) : v
ExamController
-serialVersionUID : long = 1L
+ExamController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : void
#doPost(request : HttpServletRequest, response : HttpServletResponse) : void
-showExams(request : HttpServletRequest, response : HttpServletResponse) : vo
ViewSubjectEnrolmentsController
-serialVersionUID : long = 1L
+ViewSubjectEnrolmentsController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : vo
#doPost(request : HttpServletRequest, response : HttpServletResponse) : v
AddSubjectController
-view : String = "/addSubject.jsp"
-serialVersionUID : long = 1L
+AddSubjectController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : vo
#doPost(request : HttpServletRequest, response : HttpServletResponse) : v
ViewSubjectController
-serialVersionUID : long = 1L
+ViewSubjectController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : vo
#doPost(request : HttpServletRequest, response : HttpServletResponse) : v
ViewUsersController
-view : String = "/users.jsp"
-serialVersionUID : long = 1L
+ViewUsersController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : vo
#doPost(request : HttpServletRequest, response : HttpServletResponse) : v
AddExamController
-serialVersionUID : long = 1L
+AddExamController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : vo
#doPost(request : HttpServletRequest, response : HttpServletResponse) : v
ReadOnlyExamController
-serialVersionUID : long = 1L
+ReadOnlyExamController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : vo
#doPost(request : HttpServletRequest, response : HttpServletResponse) : v
ViewSubmissionController
-serialVersionUID : long = 1L
+ViewSubmissionController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : vo
#doPost(request : HttpServletRequest, response : HttpServletResponse) : v
HomeController
-serialVersionUID : long = 1L
+HomeController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : vo
#doPost(request : HttpServletRequest, response : HttpServletResponse) : v
LogoutController
-serialVersionUID : long = 1L
+LogoutController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : vo
#doPost(request : HttpServletRequest, response : HttpServletResponse) : v
LoginController
-serialVersionUID : long = 1L
+LoginController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : vo
#doPost(request : HttpServletRequest, response : HttpServletResponse) : v
AdminHomeController
-serialVersionUID : long = 1L
+AdminHomeController()
#doGet(request : HttpServletRequest, response : HttpServletResponse) : vo
#doPost(request : HttpServletRequest, response : HttpServletResponse) : v

*Class Diagram for the **controllers** package*