# Software Architecture Report for Hogwarts O.W.L.s Exam System

SWEN 90007 Software Design and Architecture Report (Part 2)

## Git release tag

SWEN90007_2020_Part2_SDA4

## Deployment link

https://swen90007-2020-sda4.herokuapp.com/

# Contributors

Akhmetzhan Kussainov - 1026301

Robert Sharp - 186477

Fanyu Meng - 1026364

Paritosh Wadhavane - 1004023

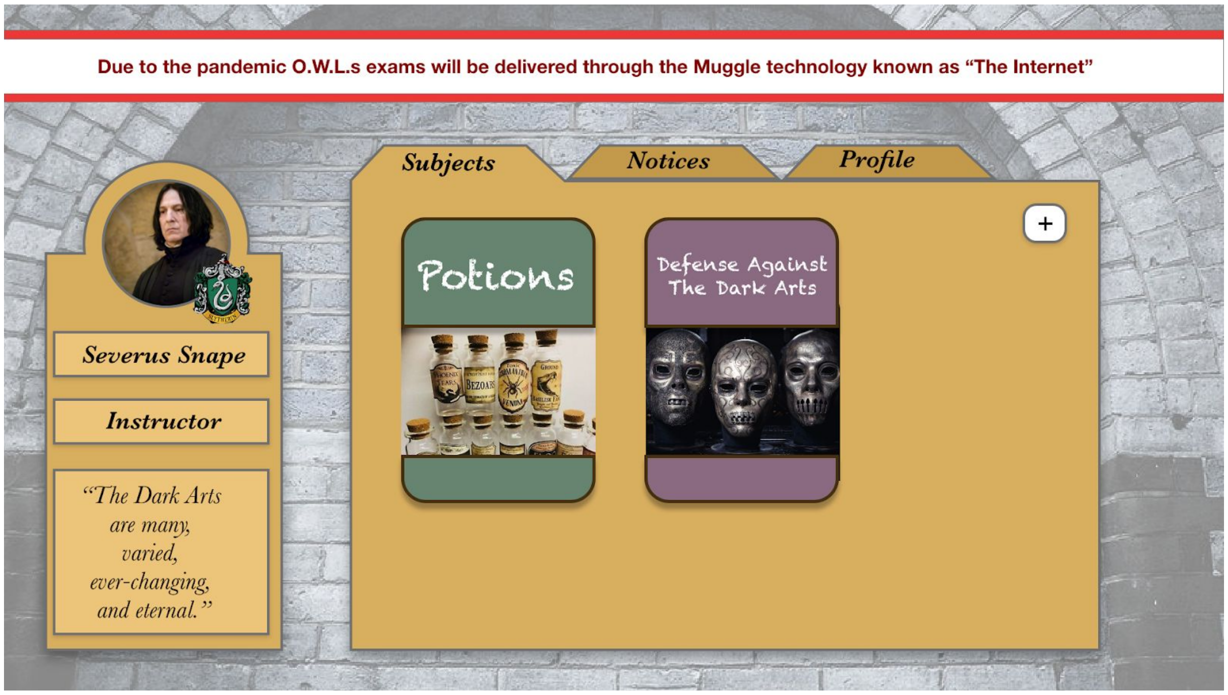
# Revision History

| Date | Version | Description | Author |
|---|---|---|---|
| 4th Oct 11:30 am | 1.0 | Done structure, state diagram for exams, domain diagram, description of patterns, pasted personas in, pasted ui mockup in, pasted Fanyu's use case diagram, wrote three user | Robert |

| | | | |
|---|---|---|---|
| | | scenarios, wrote two test cases, wrote business rules | |
| 4th Oct 12:30 am | 1.1 | Sequence diagram for Unit of Work, Embedded Value | Fanyu |
| 4 th Oct 9:00 pm | 1.2 | Identity Map sequence diagram, lazy load sequence diagram | Robert |
| 4th Oct 9:30 pm | 1.3 | Domain Model Pattern sequence diagram | Fanyu |
| | | | |

# Introduction

This is the Software Design Document for an Enterprise System for the Hogwartz School of Witchcraft and Wizardry. Due to the Covid pandemic, the school has moved online and uses the internet to undertake student assessment.


Due to the pandemic O.W.L.s exams will be delivered through the Muggle technology known as "The Internet"
Subjects
Notices
Profile
+
Potions
Defense Against The Dark Arts
Severus Snape
Instructor
"The Dark Arts are many, varied, ever-changing, and eternal."

A mockup for the UI design made by Robert. We did not have time to implement a complex

| |
|---|
| user interface. |

The Exam system uses persistent data held in databases.

The system supports multiple users. Teachers (or instructors) create exams and mark them. Students take exams. Admin can create subjects.

## Personas

1. Dobby - Administrator

| | |
|---|---|
| | Description: House elf. Has high technical abilities. Should not be able to create exams as is not a qualified wizard.<br><br>Problems: Some Hogwarts staff may be death eaters and Dobby needs to keep his eye on them. He should be able to view all details of subjects and their associated staff, students and exams.<br><br>Goals/Needs: Be able to register subjects and their instructors. |

2. Snape - Instructor

| | |
|---|---|
| | Description: Teacher at Hogwarts (Slytherin house). He is a Potions Professor. May be a death eater. Teaches multiple subjects. Teaches “Defense against the Dark Arts” and “Potions“ by himself. He also co-teaches “Herbology“ with Professor Pomona Sprout. Low technical ability.<br><br>Problems: Harry Potter. |

| | |
|---|---|
| | Dislikes marking multiple choice exams.<br><br>Goals/Needs: To create exams. To mark exams. |

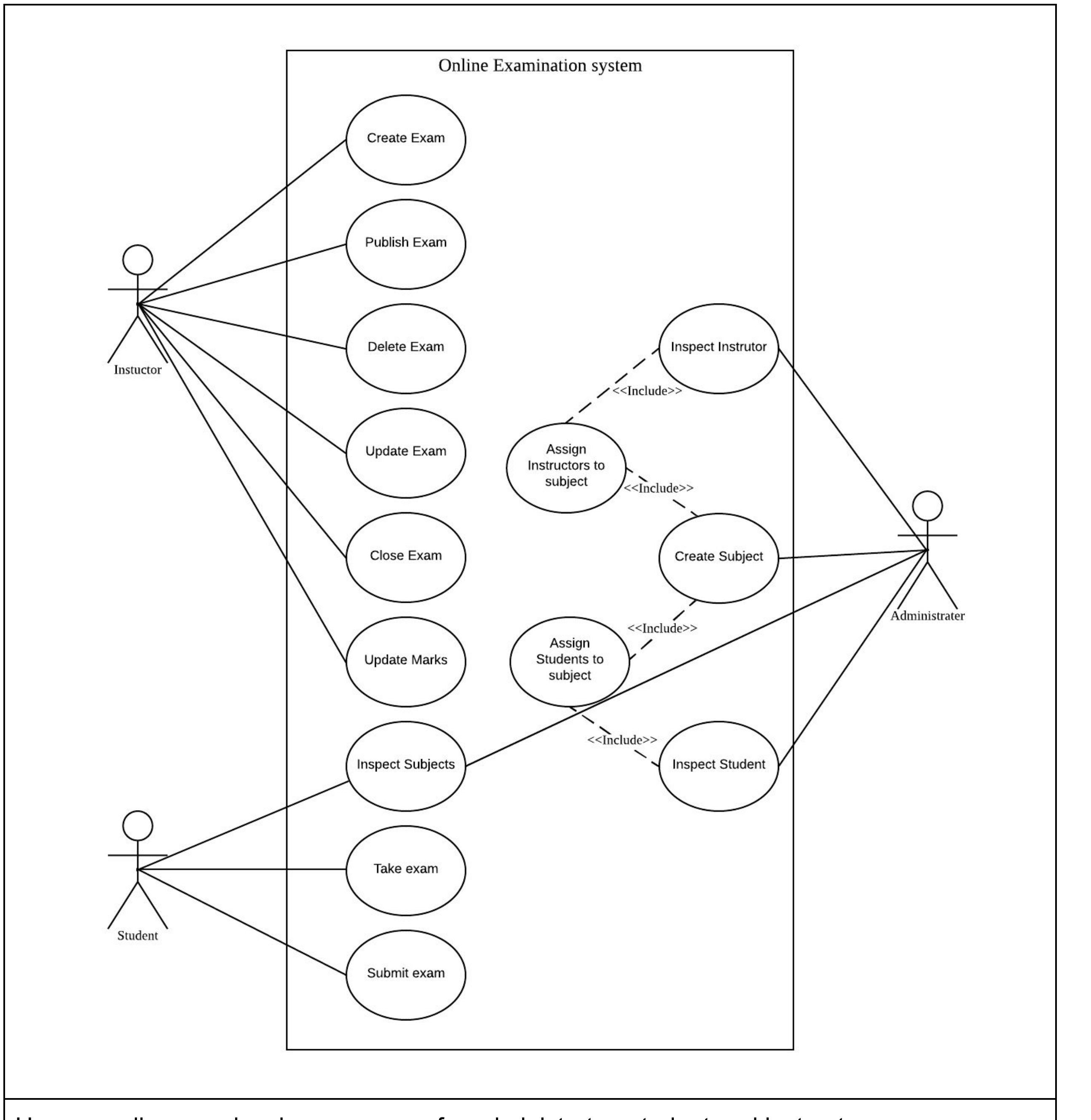
3. Harry Potter - Student

| | |
|---|---|
| | Description: Muggle student. Belongs to Gryffindor House. Moderate technical ability. Will be enrolled in multiple exams for multiple subjects.<br><br>Problems: Dementors.<br><br>Goals/Needs: Needs to know when exams will be. Needs to be able to take his exam and know his answers will be reliably recorded by the system. |

## Use Cases

Please see the initial submission for use cases.

## Use Case diagram



Use case diagram showing use cases for administrator, student and instructor.

## User Scenarios

1. Dobby creates a new subject. It is called “How to breath underwater”. He enrols Harry Potter in this subject at the time of creation, since he is the only student who needs to learn this subject.
2. Snape creates a new midsemester exam for his subject “Defense against the Dark arts”. He publishes it. The time to begin the exam is 5:00 PM (United Kingdom time zone). The duration is 30 minutes. However, he finds out that some students had stolen the answers from his desk using an invisibility cloak. So Snape closes the Exam before the 5:00 PM start time.
3. Harry takes an final exam for Potions class. However he doesn’t submit his exam before the deadline. He asks the instructor to tell him what mark he got. The instructor views the scriptbooks for the exam, and tells Harry he received 0.

## Business Rules

Here are the business rules specified in the application domain.

1. One administrator
2. A subject must have one or more instructors and one or more students

once an exam is published it cannot be unpublished

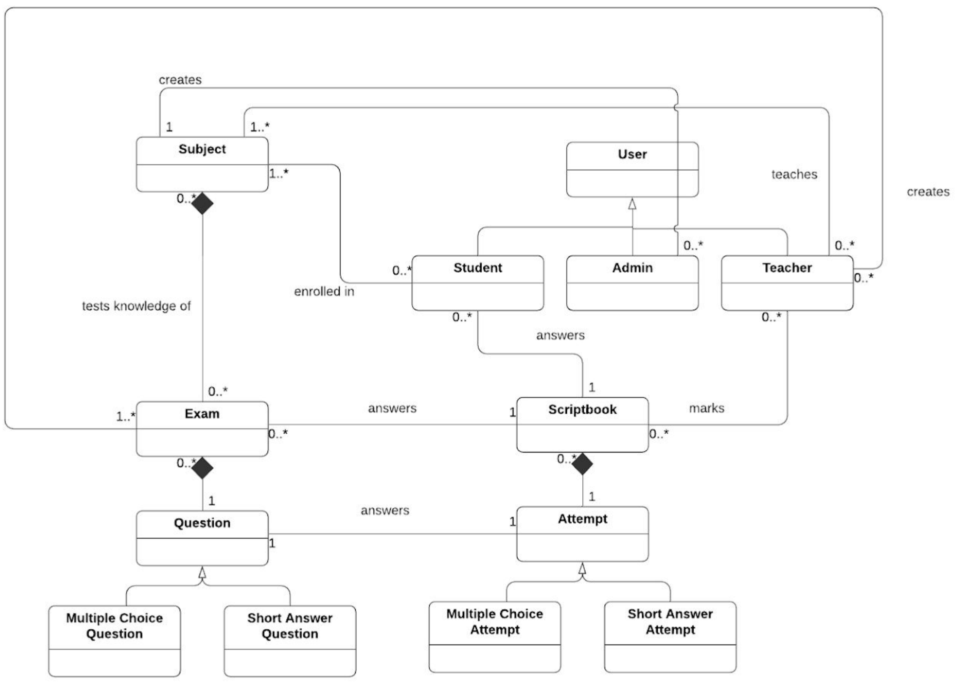
3. Exams have one or more questions

Here are some business rules we invented because they were necessary or seemed plausible:

1. Student can see every exam in subject, not just their year/semester (might help with revision).
2. exams for a subject can’t have the same name (we created this rule to make it easier for students to read)
3. A subject can’t have two exams with the same type for a year and semester (eg. can only have one mid semester exam).
4. Students can see all exams for a subject from previous years (to help with revision - our rule)
5. Teachers have a teacher number that starts with T, students have a student number that starts with S, and admin have an admin number that starts with A. This was a leftover from when Robert created the database and didn’t know how to create composite keys and is no longer necessary.
6. Students can’t also be teachers, teachers can’t also be admin etc unless they use separate accounts. This simplifies things (otherwise we would need a list for user type).
7. Exams consist of multiple choice questions shown as a seperate section to short answer questions and the numbering of questions for each section starts from 1.

## Domain model

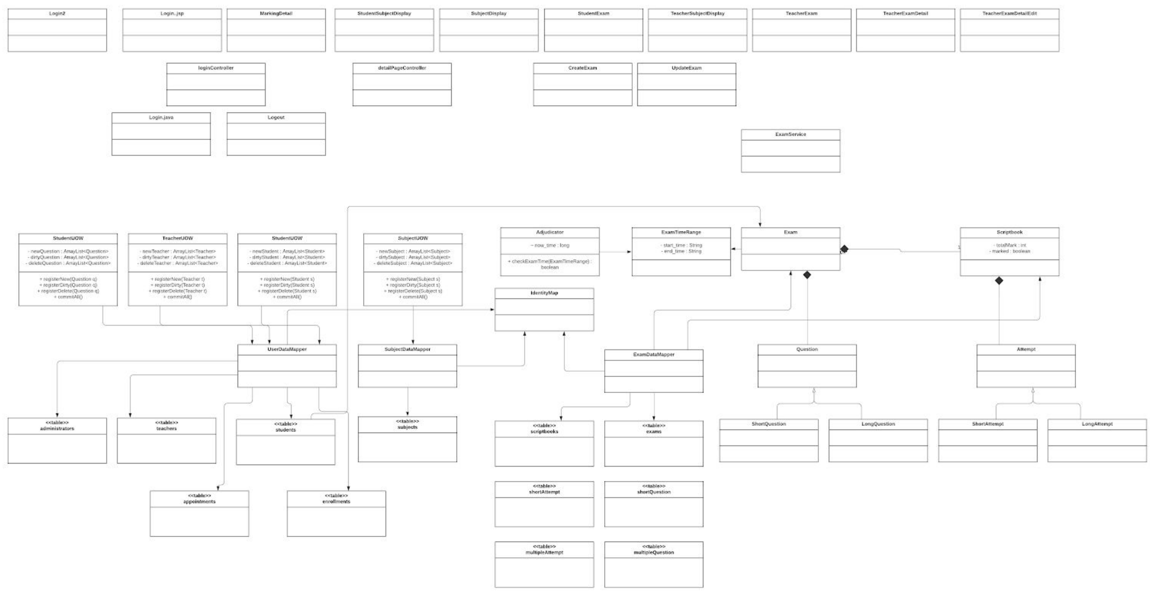
The domain model:



The Domain Model. We assumed that when admin deletes a subject, the associated exams would also be deleted. If that was not the case, then there would be an aggregation relationship rather than composition relationship seen in the diagram.

## Class diagram

The class diagram:



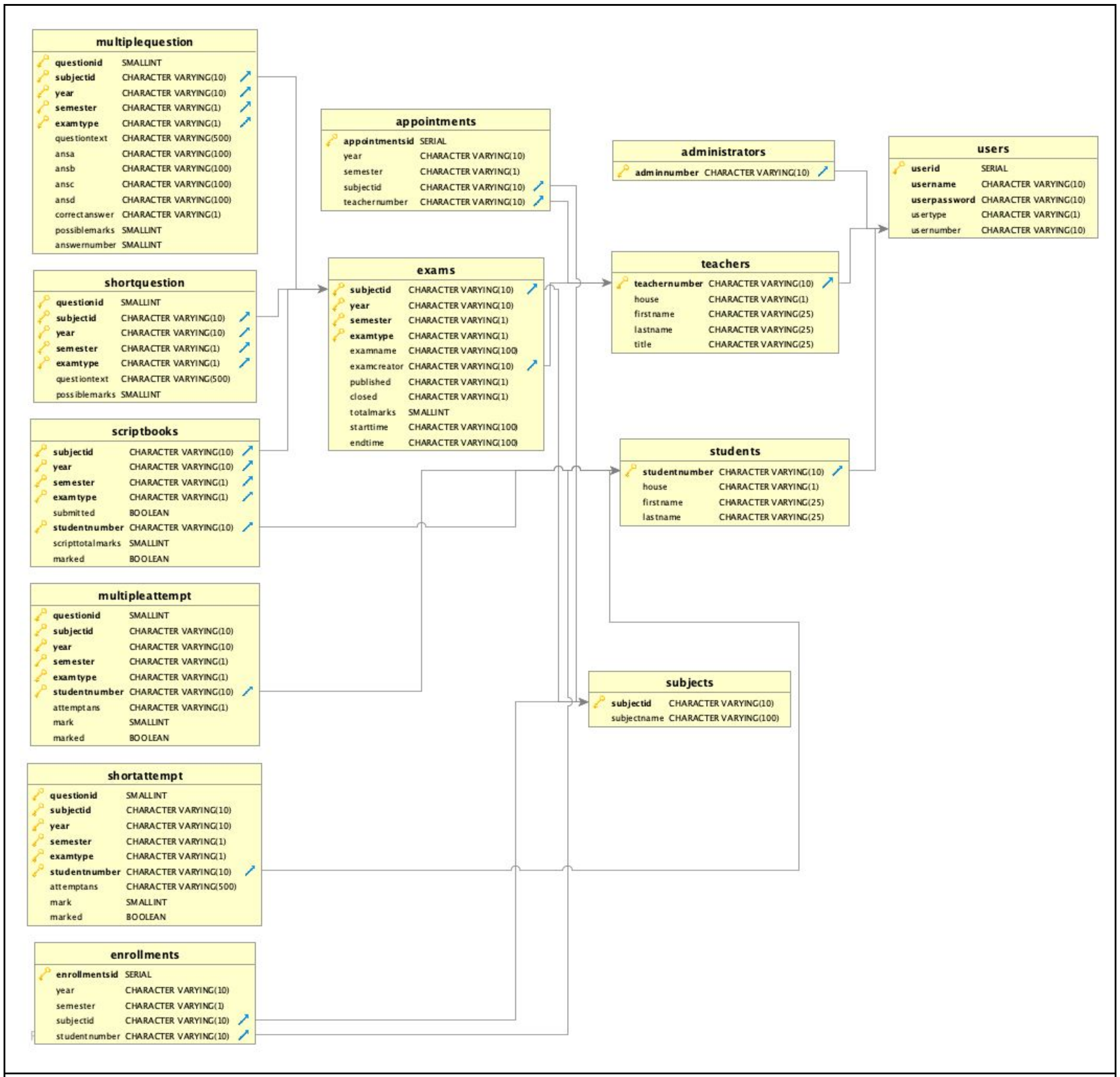
UML Class Diagram

## Architecture

Our application had three layers: Presentation, Domain and Data. The Presentation layer dealt with information presented to users. The Data layer is the SQL database holding the data. The Domain layer includes the data mapper classes that take the data from the database and transform it into domain objects.

## Database Design

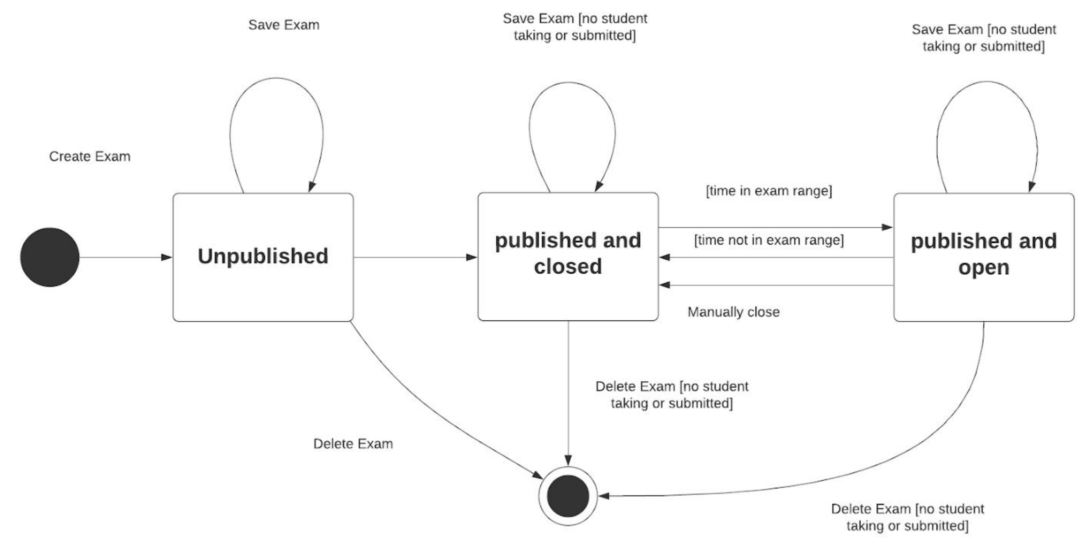
This is the design for the database:


multiplequestion
questionid SMALLINT
subjectid CHARACTER VARYING(10)
year CHARACTER VARYING(10)
semester CHARACTER VARYING(1)
examtype CHARACTER VARYING(1)
questiontext CHARACTER VARYING(500)
ansa CHARACTER VARYING(100)
ansb CHARACTER VARYING(100)
ansc CHARACTER VARYING(100)
ansd CHARACTER VARYING(100)
correctanswer CHARACTER VARYING(1)
possiblemarks SMALLINT
answernumber SMALLINT
shortquestion
questionid SMALLINT
subjectid CHARACTER VARYING(10)
year CHARACTER VARYING(10)
semester CHARACTER VARYING(1)
examtype CHARACTER VARYING(1)
questiontext CHARACTER VARYING(500)
possiblemarks SMALLINT
scriptbooks
subjectid CHARACTER VARYING(10)
year CHARACTER VARYING(10)
semester CHARACTER VARYING(1)
examtype CHARACTER VARYING(1)
submitted BOOLEAN
studentnumber CHARACTER VARYING(10)
scripttotalmarks SMALLINT
marked BOOLEAN
multipleattempt
questionid SMALLINT
subjectid CHARACTER VARYING(10)
year CHARACTER VARYING(10)
semester CHARACTER VARYING(1)
examtype CHARACTER VARYING(1)
studentnumber CHARACTER VARYING(10)
attemptans CHARACTER VARYING(1)
mark SMALLINT
marked BOOLEAN
shortattempt
questionid SMALLINT
subjectid CHARACTER VARYING(10)
year CHARACTER VARYING(10)
semester CHARACTER VARYING(1)
examtype CHARACTER VARYING(1)
studentnumber CHARACTER VARYING(10)
attemptans CHARACTER VARYING(500)
mark SMALLINT
marked BOOLEAN
enrollments
enrollmentsid SERIAL
year CHARACTER VARYING(10)
semester CHARACTER VARYING(1)
subjectid CHARACTER VARYING(10)
studentnumber CHARACTER VARYING(10)
appointments
appointmentsid SERIAL
year CHARACTER VARYING(10)
semester CHARACTER VARYING(1)
subjectid CHARACTER VARYING(10)
teachernumber CHARACTER VARYING(10)
exams
subjectid CHARACTER VARYING(10)
year CHARACTER VARYING(10)
semester CHARACTER VARYING(1)
examtype CHARACTER VARYING(1)
examname CHARACTER VARYING(100)
examcreator CHARACTER VARYING(10)
published CHARACTER VARYING(1)
closed CHARACTER VARYING(1)
totalmarks SMALLINT
starttime CHARACTER VARYING(100)
endtime CHARACTER VARYING(100)
administrators
adminnumber CHARACTER VARYING(10)
users
userid SERIAL
username CHARACTER VARYING(10)
userpassword CHARACTER VARYING(10)
usertype CHARACTER VARYING(1)
usernumber CHARACTER VARYING(10)
teachers
teachernumber CHARACTER VARYING(10)
house CHARACTER VARYING(1)
firstname CHARACTER VARYING(25)
lastname CHARACTER VARYING(25)
title CHARACTER VARYING(25)
students
studentnumber CHARACTER VARYING(10)
house CHARACTER VARYING(1)
firstname CHARACTER VARYING(25)
lastname CHARACTER VARYING(25)
subjects
subjectid CHARACTER VARYING(10)
subjectname CHARACTER VARYING(100)

This is the design for the database. The blue icons indicate foreign key. The gold icons indicate a primary key (multiple keys in one table indicate a composite key).

## Exam State Diagram

The behaviour of the Exam is complex so we include a State diagram to understand it:


Save Exam
Save Exam [no student taking or submitted]
Save Exam [no student taking or submitted]
Create Exam
[time in exam range]
Unpublished
published and closed
[time not in exam range]
published and open
Manually close
Delete Exam [no student taking or submitted]
Delete Exam
Delete Exam [no student taking or submitted]

Exam State Diagram. The condition “time in exam range” means the server time is within the range of the exam’s start and end time.

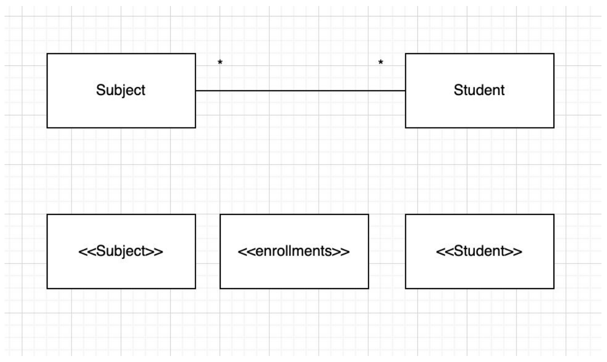
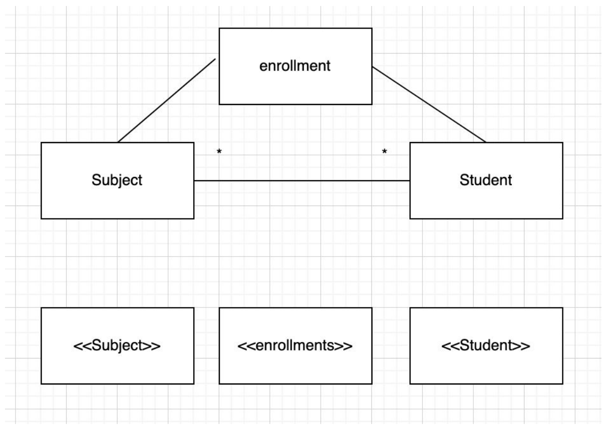
# Patterns

In this section we will describe each pattern used, why it was used, how it was implemented, and a sequence diagram.

## Association Table Mapping

We used the association table mapping pattern. This pattern is necessary when you have a many to many relationship between objects. This occurs with students and subjects: a student has many subjects, and a subject has many student. Also with teachers: a teacher teaches many subjects, and subjects can have more than one teacher. So in the database we create an association table for each. For the relationship between students and subjects, it was called enrollments. This table has the student ID and the subject ID as well as year and semester of enrollment. Likewise the relationship between teacher and subject has a table called appointments, that includes the teacher's ID and the subject ID as well as year and semester. By including year and semester, we give the system the possibility of showing only subjects a student is currently enrolled in.

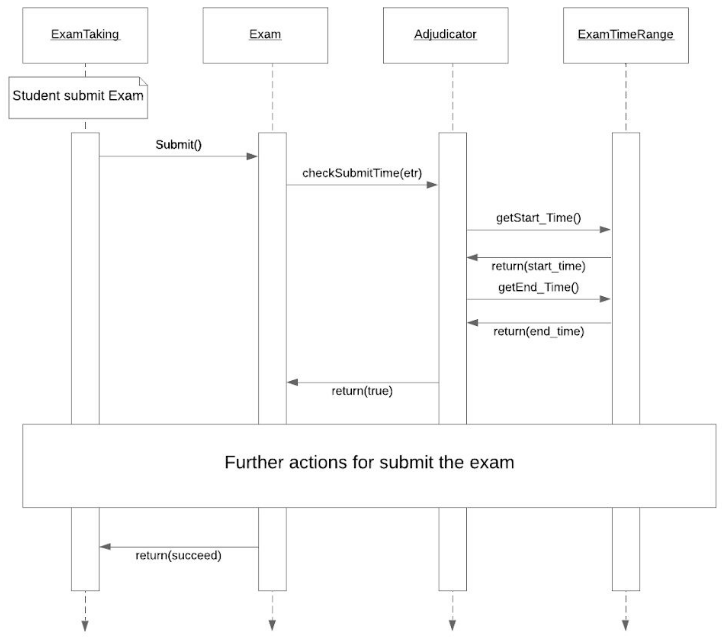
While a relational database cannot handle a many to many relationship, the domain model can. We can just have for say one subject, a linked list of student objects. We don't need an object for each association. However, this is only true in isolation. In practice, Unit of Work requires objects. So if the use cases included say the Admin changing the enrolment of a student from one subject to another, for the unit of work object to handle this, we'd need an object for each association which we can mark as created or deleted. Likewise if the association table appointments contained information such as salary, then we would need an object to mark as dirty if that changed.

| No unit of work | With unit of work |
|---|---|
|  Subject<br>Student<br>*<br>*<br><<Subject>><br><<enrollments>><br><<Student>> |  enrollment<br>Subject<br>Student<br>*<br>*<br><<Subject>><br><<enrollments>><br><<Student>> |

## Embedded Value

An embedded value refers to the situation where something exists as a separate object in the domain but is represented in the database as part of an owner table. The separate object is loaded and saved at the same time as the owner table. In this application, this occurs for the start and end time of the exam. We created a separate object for this range that the adjudicator object accepts when the function canStartExam is called.

Rather than record start and end time for each exam as a seperate table, it is more efficient to record it as two values of the owner table Exams. We create separate objects for the start and end time due to there being other information that might need to be recorded, such as how many minutes reading time, time zone etc. It’s cleaner to pass an object rather than a large number of values to the adjudicator object, and more decoupled too.

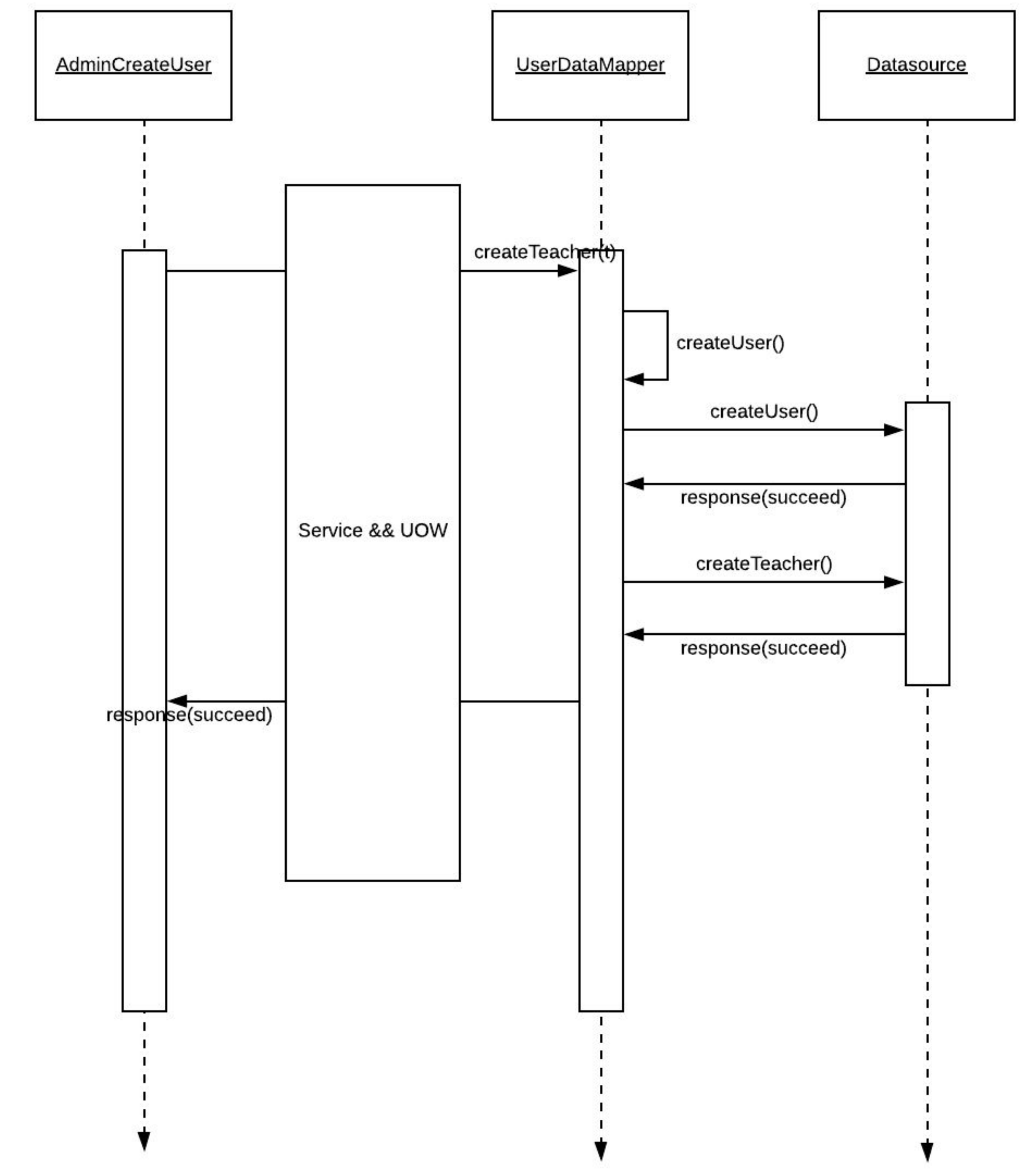


Sequence Diagram for Adjudicator and ExamTimeRange handle students who submit an exam.

## Domain Model Pattern

Domain model refers to the pattern of creating an object orientated model. This means we create objects for each student, teacher, exam etc. The alternative patterns are the Transaction script or Table module. Why use the Domain model for this application? If the system was simple, such as simply recording the enrolments for students (and changes to that), then it would make sense to use the Transaction Script pattern. However, the domain for the Exam application is quite complex. We can take advantage of polymorphism, such as having

Students, Teachers and Admins be types of Users. This means we can reuse methods. A second advantage is the flexibility it provides dealing with new behaviour.



Sequence Diagram for Create a teacher by admin

## Data Mapper

How do we get the data out of the database and into the domain? An important feature is the complexity of the domain. For example, the exam object links to information from both the exam table, the questions tables, the scriptbook table, the attempt tables. There isn't a simple one to one match of domain objects to tables. Therefore the appropriate pattern to choose is the Data Mapper pattern. Our application has SubjectDataMapper, the UserDataMapper and the ExamDataMapper.

The job of these mappers is not just to load data, but to update tables, delete records and create new records.

The datamapper interacts with the service objects, unit of work, identity map and even objects that use lazy loading.

Having a datamapper allows us to decouple the database and domain objects, which makes changing behaviour of the system easier to implement.

The sequence diagram of Unit of Work shows how the system uses the Data Mapper.

## Identity Field

How do we say update a student's record? We need a unique key in the student database, otherwise how would the data mapper object know which row to update. In the case of students, we use the studentNumber as the primary key. This is an example of a meaningful key. An example of a table with a meaningless key is the user. We used a serial key (a made up number) for this table because it was possible a system would need teachers that could also be students. For example, say Dumbledore had an Admin account and a Teacher account, but the school only gave him one ID. In that case, we couldn't use a single ID as a key, we would need a compound key of say ID and Type.

We use a lot of compound keys in the database. For example, for a student's attempt at a multiple choice question, the key is made from the student's ID, the year, type and semester of the Exam, the subject ID and the question number. Why use such a messy composite key? Well we can't really trust say teachers to create a unique Exam id themselves, so it is better to make one out of information that when combined should be unique, such as there should be one exam of the same type per subject per semester.

## Foreign Key Mapping

Objects relate to other objects: an exam has a number of scriptbooks associated with it, and each scriptbook has a number of attempts, which are themselves related to questions, which are related to exams. How do we tie all this information together in the database? Using foreign key mapping.

See the database diagram for the tables that use foreign key mapping.

## Class Table Inheritance (Inheritance Pattern)

Objects implement polymorphism using declarations like extends. This relationship is represented in the database using inheritance patterns. In the case of users, we use class table inheritance. That is, part of a student's information will be stored in the student table, and part of it (username, password etc) will be stored in the user table. Why did we use this pattern instead of Concrete Table Inheritance? We would want the task of users to login to be as quick as possible. So creating a table that just has username, type and password (and necessary foreign keys) is more efficient to load.

## Concrete Table Inheritance (Inheritance Pattern)

In the case of questions, there is no need to store information about a single question over separate tables. So in this situation we have a table for multiple choice questions, table for short answer questions, but no table for the generic object question.

## Template View (Presentation Pattern)

If we wanted to implement a fancy looking user interface, it might be necessary to use a Transform View pattern. Instead we chose to create a bare bones interface, and this the suitable pattern for this is the Template View.

The disadvantage of the Template View is the code is quite messy. There is an interleaving of static data (HTML) and Java that generates dynamic content.

## Page Controller (Inheritance Pattern)

We used the Page Controller pattern. That is, for every view there is a controller that handles the input behaviour.

The reason for using this Page Controller pattern over say the Front Controller is we weren’t sure we could implement the ability to store the type of user in a session.

We were going to use a Service layer, that is, a facade that decouples the Presentation layer from the Domain. However, this was not fully implemented.

## Unit of Work

How often should data be written to the database? The more frequent, the less efficient the system. Should the whole table be updated, or just the data that has changed? How do we keep track of what has changed?

The Unit of Work object will keep track of what objects are new, what have been deleted, and what are modified. Changes are made in the domain objects only, and then after a trigger (commit) then they are written to the database.

When a commit is made, the unit of work will go through the lists of new, dirty and deleted objects and make the appropriate changes to the database.

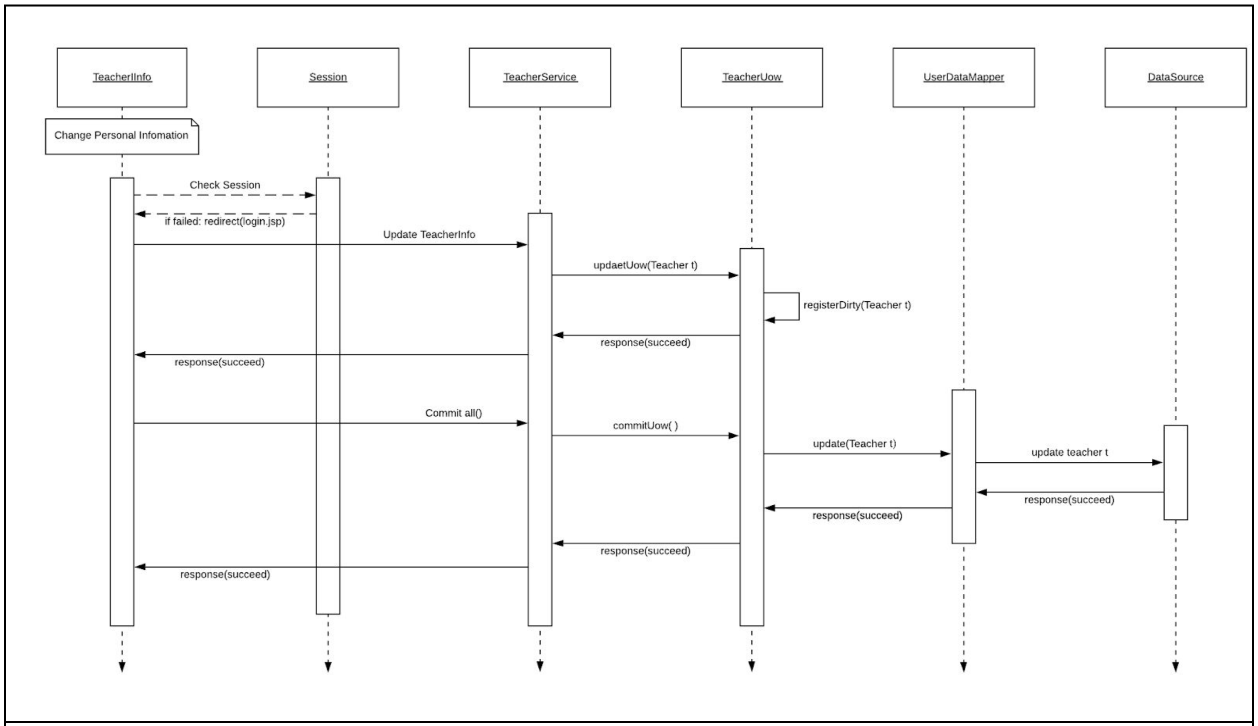
We did not make the unit of work a static class. There is a new unit of work for each session. The reason for this is due to issues stemming from concurrency.

We didn’t keep a list of clean objects instead that can be inferred from the other lists.

We used the caller registration patter, but since the system does not handle requests from multiple users in this stage, the caller registration patten has not been implemented. Lists are going to be saved as a static variable in UOW class.

The pattern is used for teachers, students, subjects, questions and exams.

Commit is called after each transaction or the user presses the button to commit changes. This was not fully implemented.

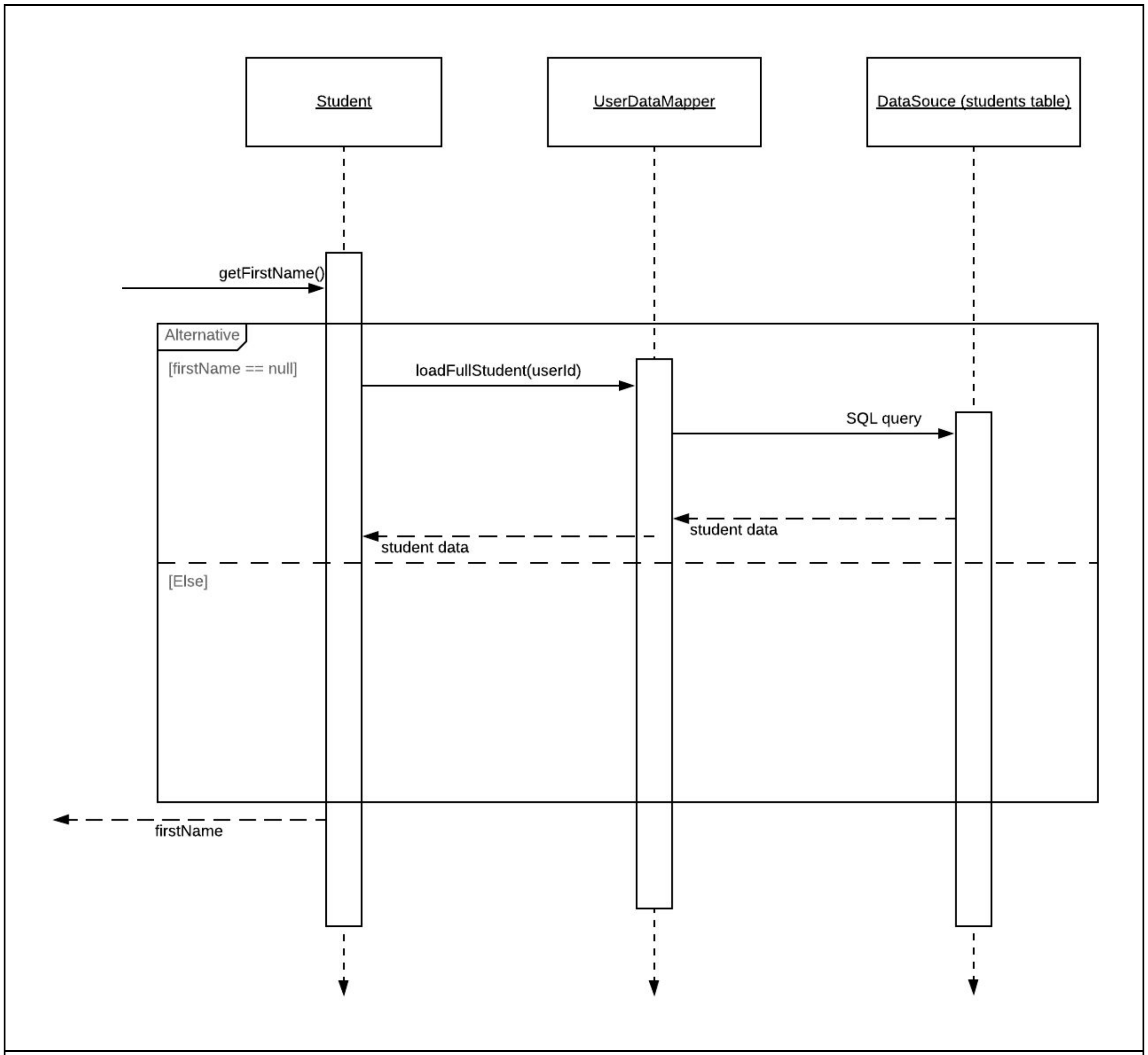


UoW for the Teacher updates personal information

## LazyLoad

We want the login page to load fast. Would it make sense then to load all the student's information, such as their contact information in case of emergency, or their results history etc? No so we use lazy loading. The specific pattern we use is Lazy Initialisation. This means we set the fields like first name, last name etc. to null. Only when say the getFirstName() method is called will that method check if the first name is null, and in that case, call a getInfo() method that will call a method in the Data Mapper to retrieve that information from the database.

We were going to use lazy loading for the exams.



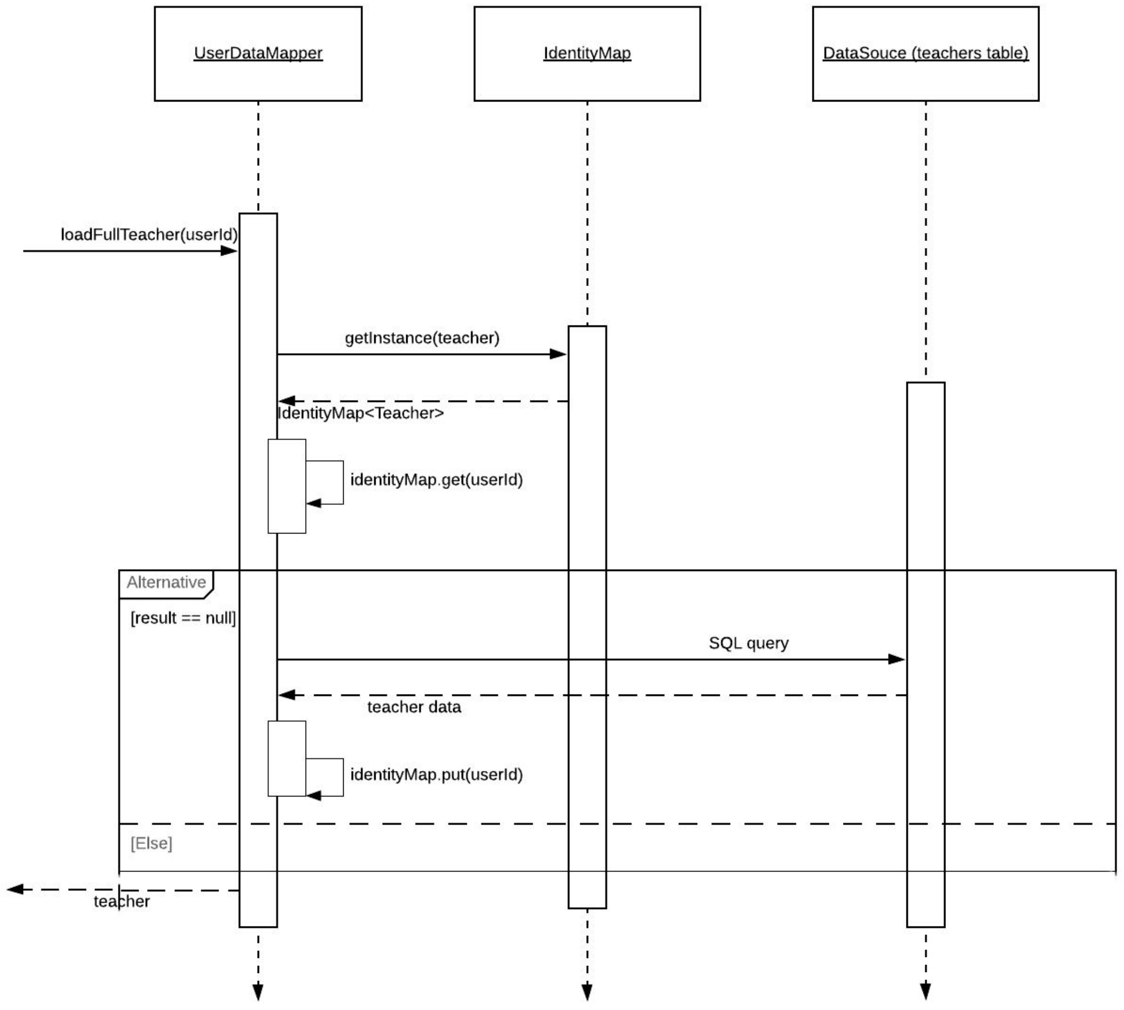
A sequence diagram showing the use of the lazy initialisation. When the method getFirstName is called, it checks if the variable is null, in which case, it calls the User Data Mapper object to fetch the data from the database.

## Identity Map

This pattern is used to make sure we don't load the data more than once. This ensures a single source of truth. It is also more efficient.

The identity map gets called by the Data Mapper objects. So for example, let's say the system allows a teacher to see the first and last name of any other instructors teaching the same subject (for example, when viewing who created an exam). Let's say in a session that teacher A examines a subject B taught by teacher C, and then A later examines subject D which is again taught by C. The UserDataMapper will be able to get from a HashMap using a key of the teacherId the teacher's object, without reloading from the datasource.

We did not implement this fully yet. While putting information to the identity map is done, checking and getting is not done.



A sequence diagram showing the use of the identity map pattern. Only if the identityMap for teachers does not contain the teacher should the UserDataMapper access the database. This has not been fully implemented.

# Testing

We used manual tests. Tests are included in the Appendix. We only tested functional requirements, not performance of the system.

## Features not fully implemented

The Identity Map pattern was not fully implemented, only putting information on the identity map and not checking before loading.

The use cases related to creating questions and marking questions were not fully implemented.

The unit of work was not fully implemented.

Lazy loading was only used for users and not fully implemented for exams.

Not immigrate all the newest changes and make them work.

## Known Bugs

Can’t login to administrator account even with correct details.

## Login

https://swen90007-2020-sda4.herokuapp.com/

Login information so that marker can test our system themselves:

| Username | Password | Account Type |
|---|---|---|
| HarryP | Abra | Student |
| HerG | Dobb | Student |
| RonW | Broken | Student |

| | | |
|---|---|---|
| SSnape | ih8harry | Teacher |
| DracoM | ih8harry2 | Student |
| Dobby | iluvharry | Admin |

## Appendix

| | |
|---|---|
| Title: | Lazy Initialisation 1 |
| Test Case Description: | To test Lazy Initialisation is working, verify the some of student information is set to null. |
| Test Case Instructions: | Disable lazy loading. Load the page that displays the user information that is specific to the student (first name, last name etc). |
| Test Case Data: | |
| Expected result: | See null for fields first name, given name. |
| Actual result: | See null for fields first name, given name. |
| Pass/Fail: | Pass |

| | |
|---|---|
| Title: | Lazy Load Test 1 |
| Test Case Description: | To test Lazy loading is working, |
| Test Case Instructions: | load the page that displays the user information that is specific to the student (first name, last name etc). |
| Test Case Data: | |

| | |
|---|---|
| Expected result: | See all information |
| Actual result: | See all information |
| Pass/Fail: | Pass |

| | |
|---|---|
| Title: | Data Mapper Test 1 (On fmme branch) |
| Test Case Description: | To test User Data Mapper loadAllUsers(), um.getAllStudents(), getAllTeachers(), loadFullStudent(String userNumber), loadFullTeacher(String userNumber), loadAllSubject() works |
| Test Case Instructions: | load the Admin page, which displays all the students, teachers and subjects information. |
| Test Case Data: | |
| Expected result: | See all information |
| Actual result: | See all information |
| Pass/Fail: | Pass |

| | |
|---|---|
| Title: | Data Mapper Test 2 |
| Test Case Description: | To test User Data Mapper loadSubjectByStudent(Student s) is working |
| Test Case Instructions: | Login student account, subjects that the student enrolled should be shown. |
| Test Case Data: | |
| Expected result: | See all information |

| | |
|---|---|
| Actual result: | See all information |
| Pass/Fail: | Pass |