

Project Report
Data Storage Paradigms, IV1351

Lucas Larsson
lulars@kth.se
09-01-2022

1. Introduction

The following project compose of one task namely building a database for a music school to be able to store all needed data for their operations, the task is divided into 4 sub-tasks below that starts with designing a conceptual model then a logical/physical model that follows with building a database to contain the information needed and finally building a program that can perform operation such as renting instruments or terminating already rented instrument to name a few.
I worked on all project tasks with Dennis Hadzialic.

2. Literature Study

Literature study is included as one section for all 4 tasks, first I am going to list general literature and then task specific.

General:

- Canvas webpage. [link](#)
- Fundamentals of database systems 7th edition, Elmasri and Navathe.
- PostgreSQL documentation. [link](#)

Task specific resources:

- Task 1 didn't require anything specific besides the general resources listed above, a very helpful resource was the lectures given by Leif, and the course literature for the inheritance part.
- Task 2 same here as well the best help here was from the lecture given by our awesome teacher, it is really easy to follow and divided into small information dense videos.
- Task 3 was mostly the same, I watched the lecture given on SQL but didn't find it especially helpful, I read more about the built in functions for the specific DBMS used, in my case it was PostgreSQL, and started looking for similar queries online mainly [Stack Overflow](#), another page that I read a lot on is the postgres documentation on inheritance [here](#).
- Task 4 was the one without many resources, it was building a program and I have done that before so it was not a lot of research, the only thing we checked out was the lecture given by Leif just to see how a database is connected to a java program.

3.1 Method

After doing the necessary research and reading the requirements I with my project partner started designing the conceptual model, the program used for the design is Astah since we have used it before and is the program used in the lectures by the teacher, we first wrote a list with all the requirements needed from the requirement text on the project site on canvas.

We started with noun identification from the provided requirements, where we go through the text and mark each noun and add it to the design as an entity, after getting all the nouns from the text we started thinking of entities that may be needed but not existing as nouns in the text requirements, this step is also known as category list search, and is basically exactly how it sound, we think of different situations or categories and if we have all the necessary entities we continue otherwise we create new entities for them. Next step was to remove unnecessary entities that has no purpose in the design, following step is to start considering attributes, which entities has attributes and which entities can be replaced as attributes in other entities.

Once we had found all entities needed and their attributes, we start looking for the relations between them, associations between entities and how they connect we label each association to show the reasoning behind the connection.

Lastly, we remove all entities without attributes since there is no reason for having a table that does not contain data columns, we define cardinality for the attributes, weather they are allowed to be without value, or if they are unique or can be duplicated and the data types specified for them.

A method that we thought was helpful is to work together on different designs from the same requirements and then compare the designs to see what one has missed, and how was the interpretation of the text to prevent mistakes or misinterpretation, after compering designs we then created one design as a final design.

3.2 Result

Figure 1 is the result of flowing the method above and it shows the conceptual model build by us, it is a small model compared to other models build by other students from the same class following the same requirements, but we feel that it does contain all the needed information.

The model shows where information of a new applicant will be stored and how will be able to handle them, to process a new applicant, staff need to be able to see whether there are vaccines, which they can see in the schedule entity.

All information regarding a lesson can be viewed i.e number of places, instrument used for the lesson, genre, student level, time for group and ensemble and appointment for individual lessons.

Staff can make bookings that get registered then on the schedule and can look up vaccines in case of new applicants.

Payment is modeled as two separate entities; instructor payment is based on the number of lessons they give, and student payment is based on the number of the lessons they take, if they have rented instrument from the school and if they have sibling attending the school which can result in a discount.

Renting instruments is shown as a single entity to demonstrate that it is a possibility for the students to rent instrument, this entity is changed in the next task for more clarity, for now it is only added a note to preserve the business rule that a student is allowed to have a max of 2 instrument a time.

Student, applicant and instructor are inherited from the entity person, which has all the information that need to be stored of a person.

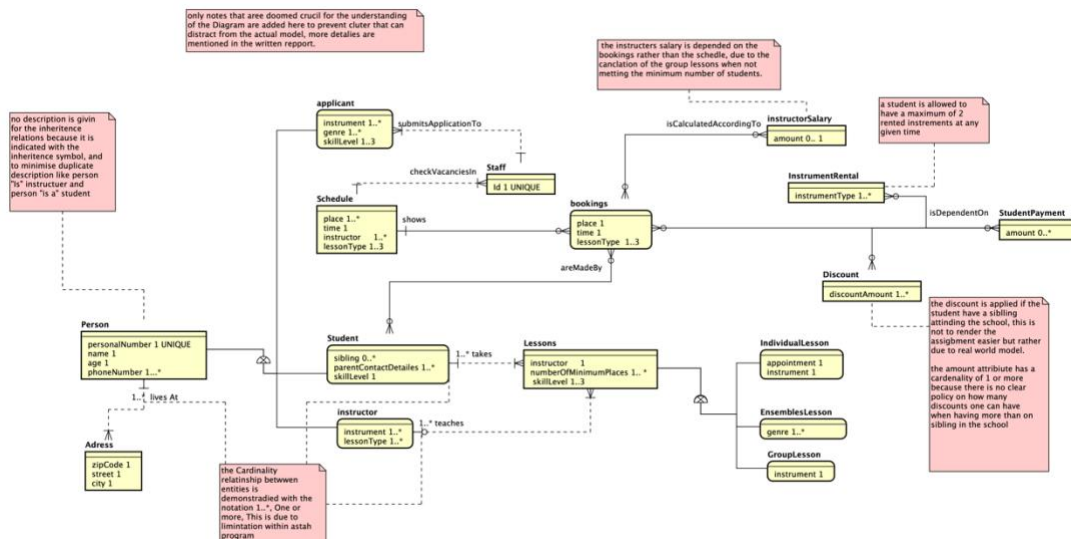


Figure 1: Conceptual model representing task-1 of the project.

3.3 Discussion

“Soundgood sells music lessons to students who want to learn to play an instrument. When someone wants to attend the school, they apply by submitting contact details, which instrument they want to learn, and their present skill.”

The above quoted phrase is the first requirement which we fulfil by having the schedule entity where staff can check for vacancies and the applicant entity where an applicant data will be stored.

There are 3 type of lessons which are represented with three entities that inherit the mutual attributes to minimize repatriation and maximize normalization in the database. A lesson is not given unless minimum number of participant attend the lesson which is modeled by the min/max number of student' s attribute. This means that the higher-grade points are also fulfilled by using inherits in at least one place.

All information listed in sections 1.1 and 1.2 on the project website on canvas [here](#) are modeled in the conceptual model in figure 1.

The notation used in the conceptual model is the crow foot notation, at some places notes are placed indicating a different cardinality that is due to limitation in the astah soft wear, where it is not possible to specify cardinality such as 1...*. In total the model has 16 entities which is sufficient to accommodate for all the data that need to be stored in the database, this is controlled by checking that all the data needed to be stored in the database application is accounted for.

As mentioned above we used inheritance to avoid repetition of the same attributes in the different lesson entities, for an example: all lessons have an instructor, number of participance and a skill level, so those are the attributes placed in the lessons entity to be inherited to the other three entities. Individual_lesson entity has 1 attribute that is unique to it appointment and instrument, ensembles_lesson entity has one unique attribute “genre” , and group_lesson has the attribute instrument. This design is to avoid null values in the entities. There are multiple different ways to model the entities one way without inheritance would be, to have the same three lesson entities with the three attributes in the entity lessons in them and remove the lessons entity, which can work as well but is not preferable because it does not bring any more value or help the understanding of the diagram in whole, it will only serve to clutter it more and would result in a database application that is hard to maintain, that is if one would like to add an attribute to all lessons it would need to be added to all three entities, where in the case of inheritance if a attribute is needed in all lesson types it would need to be added at only one place in the lessons entity that is inherited, the same is true for removal as well.

4.1 Method

The same diagram editor is used here as in the previous task Astah, no other software is used. The DBMS is PostgreSQL.

I started by creating a table for each entity present in the diagram from figure 1, after creating the tables, I created a column for each attribute that has cardinality of maximum 1 in the tables, all attributes with cardinality higher than 1 are replaced with a table in the logical/ physical model, after creating the necessary tables I started considering the data type to be used for the columns, such a choice is mainly the client responsibility to specify the data they want to store in the columns, since this is a school project I took the freedom to decide myself the datatypes after a discussion with my project partner.

Next step is to consider column constraints, mainly 2 choices are important, whether the column can be without a value and if it is unique, after that I started assigning Primary Keys to strong entities, all primary keys are surrogate keys, and then foreign keys.

As in the previous task after finishing work on the separate tables I started considering relations between them, the difference here is that one table cannot have many to many relationships with another table, (notice that I am using the word table to describe the actual table and relationship to describe connection between two tables).

The relationship creation in the logical/physical table is in multilabel steps, first is creating the relation, second assigning the PK of the strong entity as a FK in the weak entity, if the weak entity has a meaning in existing by itself it is assigned its own PK otherwise the foreign key assigned to it from the strong entity acts as its PK, after finishing the connection we need to consider the FK constraints, as in what should happen in case of updating or deletion on the table rows both the strong and weak entity.

Last step is the many to many relations from the table in figure 1, if now other work around can be made to change the many to many relation, a cross-reference table is needed, a cross reference table act as a solution to be able to map the multiple rows of a tables to each other, the PK assigned to the cross-reference table is either its own surrogate PK or a two combined FKs from the two tables, the method used in this design is the latter, and again as previously we consider FK constraints, finally we can add more columns to the newly created table if needed.

4.2 Result

The diagram present in figure 2 is the result of task 2, as shown below some design changes were made compared to the diagram from task 1, for an example the address entity was removed and added inside the person entity, which is a denormalization in a way, but it is seen as a better fit from the designer's perspective.

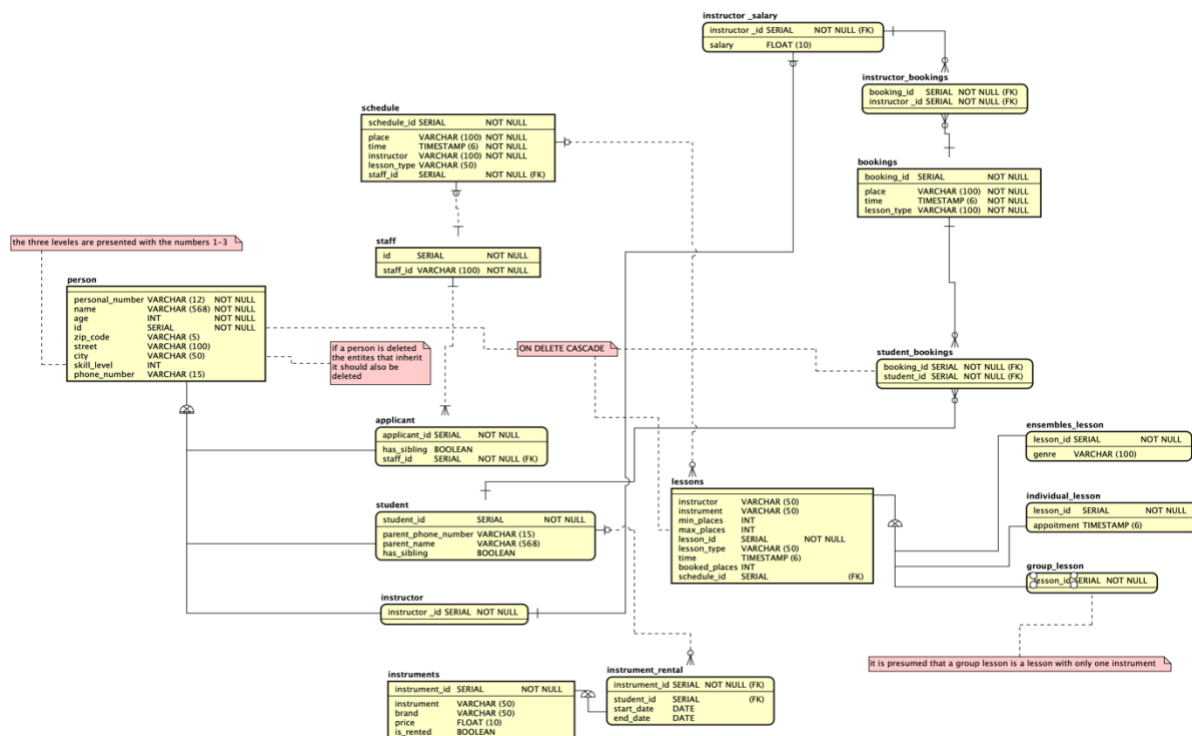


Figure 2: logical / physical diagram.

Two new cross-reference tables are created to solve the many to many relations from task 1, student bookings and instructor bookings, they both have combination of FKs as their primary key to solve the rows between the entities booking—student and booking—instructor_salary respectively.

The data type SERIAL is used for all ids as I am using postgres and it is a postgres native data type, all ids are NOT NULL since they are used as PK for the entities. The name attribute has a maximum of 568 since it is the longest recorded name, and there is no client to consult on the data type, no PK is given for entities person or lesson, the reasoning behind that is that it is not interesting to search for a person in the data base or for lessons entity, it is however relevant to search for student or instructor and a specific type of lesson.

The has_sibling attribute contains a integer value representing sibling count, where a student get discount percentage depending on the sibling count, this modeling not render

the task easier, it is rather build on real world application, if you are a member of a school/ gym it is often that you can get a discount if two persons from the same house-hold are members, so it is reasonable to assume the same for this situation.

The skill_level attribute is represented with a string which can hold the three levels a student/ applicant can have beginner, intermediate and advanced.

4.3 Discussion

Task 2 is about building the data base that is going to be used in task 3, but after working on the project the I think they should be combined into one bigger part, since after creating the model shown in figure 2 and starting on the third task a lot of changes were made that impacted the design to the point that it is not the same design any more, where I needed to go back to Task 2 and change it.

As mentioned before datatypes are the clients responsibility, it is their database and choice, but some choices are the programmer/designer, simply due to the client often not having experience in the field of computer science, attributes such as personal number and identifications are better stored as strings even though they are numbers, this is to avoid mistakes later, such as someone born in year 2000, would have their personal number stating with 2 zeros that has no value and won't be represented in a integer datatype where on the other side string will present each character with no regard to their value, and in doing so preserving the data.

The skill_level attribute is represented with a string which can hold the three levels a student/ applicant can have beginner, intermediate and advanced. It is true that when having the skill_level attribute in the person entity it means it is inherited to the instructor where it is a redundant attribute, but the other choice was to have it in two separate places, and after considering both solutions we decided to have it in the person entity.

5.1 Method

The DBMS used in this task is the same used to model the previous one namely PostgreSQL.

The tools used to develop the queries are Jet-Brains IntelliJ, DataGrip ideas and the Mac OS terminal.

There is no specific reasoning behind the choice of the tools and the DBMS, it simply because the lecturer uses PostgreSQL in the lectures giving on the project and my own experience of IntelliJ and the good integration between all Jet-Brains tools, the MacOS terminal is an extra edition just for quick access and to get familiar with the SQL queries from another interface to maximize learning outcome, and since we are building a application that would have a CLI.

The SQL script that creates the database application is generated at first from the task 2 diagram using Astah soft-wear and then tweaked furthermore by adding moreSQL statements to satisfy all the requirements, an example of the added SQL statements is all script regarding inheritance since Astah does not support it and does not export the inherited entities as intended.

The next couple of steps are straight forward, I started reading on the queries that need to be performed on the database and started doing research accordingly, in this step is basically all internet research on the specific DBMS and what functions are built in and which functions need to be created to be able to extract the data requested.

At sometimes more columns needed to be added, which showed flowed design, example of that is the price column for the table lessons, and at other times whole tables needed to be added in order to satisfy the conditions for the higher-grade tasks, more about the specific tables in the result section.

Finally, I needed to generate a script to insert data to be able to evaluate that the sql queries works.

5.2 Result

The result is in the following [link](#) to the git-hub repository, read the README.md file for instructions on running the database and the scripts for inserting mock data and querying the database.

```
sg=# SELECT DATE_TRUNC('month',time) AS time,
sg=# COUNT (id) AS count FROM ensembles_lesson
sg=# GROUP BY DATE_TRUNC('month', time) ORDER BY time;
      time           | count
-----+-----
2022-01-01 00:00:00 |    16
2022-02-01 00:00:00 |    14
(2 rows)

sg=#
```

Figure 3.1 a snippet showing the first query, representing a table with a count of how many ensembles lessons are given per month.

```
sg=# SELECT DATE_TRUNC('month',time) AS time,
sg=# COUNT (id) AS count FROM individual_lesson
sg=# GROUP BY DATE_TRUNC('month', time) ORDER BY time;
      time           | count
-----+-----
2022-01-01 00:00:00 |    18
2022-02-01 00:00:00 |    12
(2 rows)
```

Figure 3.2 a snippet showing the second query, representing a table with a count of how many individual lessons are given per month.

```
sg=# SELECT DATE_TRUNC('month',time) AS time,
sg=# COUNT (id) AS count FROM group_lesson
sg=# GROUP BY DATE_TRUNC('month', time) ORDER BY time;
      time           | count
-----+-----
2022-01-01 00:00:00 |    17
2022-02-01 00:00:00 |    13
(2 rows)
```

Figure 3.3 a snippet showing the third query, representing a table with a count of how many group lessons are given per month.

```
sg=# SELECT DATE_TRUNC('month',time) AS time,
sg=# COUNT (id) AS count FROM lessons
sg=# GROUP BY DATE_TRUNC('month', time) ORDER BY time;
      time           | count
-----+-----
2022-01-01 00:00:00 |    51
2022-02-01 00:00:00 |    39
(2 rows)
```

Figure 3.4 a snippet showing the fourth query, representing a table with a count of how many lessons are given per month, regardless of the type.

Figures 3.1-3.4 are the solution for first query on the third task, the tables show one number for how many lessons are given of a specific type per month and the last one shows one number regardless of the lesson type, it is shown as only two rows in this example, but the query work for any number, it is showing only 2 rows due to the data

only covering 2 months forward from today's date.

The queries have the same structure where one select first the time column and with the postgres built in function `DATE_TRUNC` that returns the result over an entire month, and then the count column where the `COUNT` function count the id column to return a number of the lessons per month, the id column is used for the count because it is always guaranteed to have distinct values for each lesson, then the `FROM` statement says from which table should this function run and finally the whole statement is grouped and ordered by time, using the respective function.

```
sg=# SELECT DATE_TRUNC('year',time) AS time,
sg=# ROUND((CAST (COUNT(id)AS DECIMAL)/12)::DECIMAL,2)
sg=# AS AVG FROM ensembles_lesson
sg=# GROUP BY DATE_TRUNC('year', time) ORDER BY time;
      time      | avg
-----+-----
2022-01-01 00:00:00 | 2.50
(1 row)
```

Figure 3.5 a snippet showing query number 5, representing a table with an average of how many ensembles lessons are given per month.

```
sg=# SELECT DATE_TRUNC('year',time) AS time,
sg=# ROUND((CAST (COUNT(id)AS DECIMAL)/12)::DECIMAL,2)
sg=# AS AVG FROM individual_lesson
sg=# GROUP BY DATE_TRUNC('year', time) ORDER BY time;
      time      | avg
-----+-----
2022-01-01 00:00:00 | 2.50
(1 row)
```

Figure 3.6 a snippet showing query number 6, representing a table with an average of how many individual lessons are given per month.

```
sg=# SELECT DATE_TRUNC('year',time) AS time,
sg=# ROUND((CAST (COUNT(id)AS DECIMAL)/12)::DECIMAL,2)
sg=# AS AVG FROM group_lesson
sg=# GROUP BY DATE_TRUNC('year', time) ORDER BY time;
      time      | avg
-----+-----
2022-01-01 00:00:00 | 2.50
(1 row)
```

Figure 3.7 a snippet showing query number 7, representing a table with an average of how many group lessons are given per month.

```
sg=# SELECT DATE_TRUNC('year',time) AS time,
sg=# ROUND((CAST (COUNT(id)AS DECIMAL)/12)::DECIMAL,2)
sg=# AS AVG FROM lessons
sg=# GROUP BY DATE_TRUNC('year', time) ORDER BY time;
      time      | avg
-----+-----
2022-01-01 00:00:00 | 7.50
(1 row)
```

Figure 3.8 a snippet showing query number 8, representing a table with an average of how many lessons are given per month, regardless of lesson type.

Queries in figures 3.5-3.8 represent the solution for the second query on the third task, they return a table with an average of the number of lessons for a giving month.

Not much is different here from the previously explained query the main difference is that after counting the number of lessons per year, the count result gets divided on 12 to get an average over the entire year, the count function returns number as a string that get casted to a numerical value and get specified to max of 2 decimal precision.

```
sg=# WITH time_report AS (
sg(#      select instructor, count (id)
sg(#      from lessons
sg(#      WHERE date_trunc('month', time)=date_trunc('month',current_timestamp)
sg(#      group by instructor order by count DESC
sg(#      )
sg=#      SELECT * FROM time_report WHERE count > 2;
instructor | count
-----+-----
Carly      |    10
Dudley     |     6
Farlee     |     6
Lilyan     |     6
Lulita     |     5
Modesta    |     5
Jesselyn   |     4
Harper     |     3
Martita    |     3
Niccolo    |     3
(10 rows)
```

Figure 3.9 a snippet showing query number 9, representing a table with two columns with instructors and their lesson count, that is any instructors that have worked over 2 lessons per month.

The query present in figure 3.9 first creates a variable called `time_report` with a count of instructors given lessons over the period of the current month using the postgres function `date_trunc` for the current month as shown in the query and then it selects only instructors that have over the specified number of lessons, the number is 2 in this example, but it can be used with any variable as requested in the requirements.

```
sg=# SELECT * FROM (
SELECT max_places, booked_places, instructor, instrument, genre, time,
CASE
WHEN max_places = booked_places
THEN 'full booked'

WHEN max_places - booked_places <=2
THEN '1-2 seats left'

ELSE 'there is more than 2 places'

End AS place_availability From ensembles_lesson)
ensembles_lesson WHERE
extract ('week' FROM time )=
extract( 'week'from CURRENT_TIMESTAMP+ INTERVAL '1 week')

ORDER BY time ASC, genre;
max_places | booked_places | instructor | instrument | genre | time | place_availability
-----+-----+-----+-----+-----+-----+-----
30 | 15 | Lulita | drums | indie rock | 2022-01-18 18:02:44 | there is more than 2 places
30 | 27 | Modesta | French horn | indie | 2022-01-22 23:19:47 | there is more than 2 places
30 | 26 | Dudley | drums | rock | 2022-01-23 06:32:30 | there is more than 2 places
(3 rows)
```

Figure 3.10 a snippet showing query number 10, representing a table that shows place availability in the `ensembles_lesson` table for the coming week.

The above query in figure 3.10 use a CASE statement as instructed in the project web-

page on canvas.

For this specific query a new column was needed and added which is booked_places, the query is straight forward, if the condition inside the case statement evaluate to true then the THEN statement is executed and returns as a string with the string specified as shown in the figure 3.10, extract statement is used to specify the period that the query should search during. The requested time is next week, and this is implemented with an extract statement that search for intervals of week with a difference of 1 week ahead, which will always return the next week from the current week. ([Postgres documentation](#))

```
sg=# INSERT INTO lessons_archive ( student_id, lesson_type, price)
sg=#     SELECT student_id, lesson_type, price FROM lessons
sg=#     WHERE extract(DAY FROM time) = extract(DAY FROM now());
INSERT 0 9
sg=# select * from lessons_archive;
 price | student_id |   lesson_type
-----+-----+-----
  366  | 15457      | Ensembles_lesson
  468  | 04399      | Ensembles_lesson
  845  | 96292      | Ensembles_lesson
  758  | 43695      | Group Lesson
  638  | 41129      | Group Lesson
  824  | 81645      | Group Lesson
  890  | 13436      | Group Lesson
  571  | 56603      | Individual Lesson
  996  | 42081      | Individual Lesson
(9 rows)
```

Figure 3.11 a snippet showing query number 11, a query to archive given lessons in the database as a part of the high-grade task.

The query in figure 3.11 inserts student_id, lesson_type and price from the lessons table for all the lesson types given today.

This query is for the sound good school to be able to know which lesson was given to which student at what price, for this query a new table was created lessons_archive.

5.3 Discussion

One thing to address right away is the amount of data inserted into the database, there is not a big amount of data in the database, but it has sufficient amount, which the only thing that matters.

In the project webpage it is instructed to discuss the choice of creating views or not, no views are created in this task simply because they are never used or needed.

Not much is there to write in the discussion part, as I mentioned before I feel that task 2 and 3 would be a better fit as one task, since after learning about the queries needed to be performed on the database I needed to change it and redesign the task 2 diagram, some changes were not important to the understanding of the application such as rental archive and lesson archive, so they were not added to the design.

6.1 Method

Same IDEs from the previous tasks are used here, i.e JetBrains IntelliJ and Data Grip, no reason other than I am used to them and have already the whole project in them.

Our project group aimed for the high grades, so we designed our application using the MVC pattern, both project members have had the course IV1350 which helped a lot in regard for the applications design and build.

First step we did was to start the design on a white board and figure out which classes we need and what are the methods that will be used and then what methods are used in more than one place to avoid repetitive code for a cleaner more efficient program. After figuring out the necessary classes we designed the SQL queries that would be sent from the program to the database.

Last step was to go to square one and watch a lecture on database application given by our teacher Lief, this was to have an idea on how a database would connect with a program and to have a general idea of the program designed. After that we just got to work wrote the methods and checked the connection, once the first query worked the rest was easy to fix.

This is the list written by us on what functionality the application would need to have.

1. The user should be able to **list all** the available instruments to rent of a certain kind. The user should be displayed with the following information: **brand** and **price**.
2. The user should be able to **rent an instrument** to a student if the student is eligible to rent an instrument (a student can only rent 2 instruments at a time).
3. The user should be able to **terminate an active rent** without deleting data from the database.

All the above listed command would have their output in the CLI.

6.2 Result

The application can be found the project GitHub repository [here](#).

```
1. li <type> (List instruments with specific type)
2. rent <studentId> <instrumentId> (rent instrument with id to student with id)
3. terminate <instrumentId> (terminate rental for the student and instrument)
4. exit
```

Figure 4.1 a snippet of the CLI.

Figure 4.1 is the first print out presented for a user, it shows the three functions required to be in the application.

```
li drums
{instrument='drums', brand='Greydon', price=303.0 SEK}
```

Figure 4.2 a printout of the output of the function number 1 when called with the augment drums as an instrument type.

After performing the first function the CLI goes back to the list of the functions that can be performed as in figure 4.1, this is achieved using a switch statement.

```
rent 32910 376

The student has 1 rented and is eligible to rent the desired item. Proceed? (y/n)
|
```

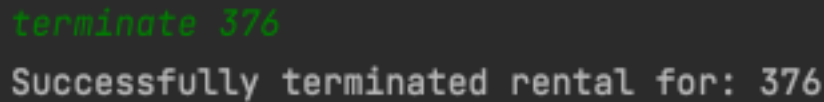
Figure 4.3 is a screen shot showing the how the second function is performed.

The user inputs the command rent and then their student id followed by the id of the instrument they wish to rent, as shown in line number 2 figure 4.1.

They are then presented with a sentence asking them to confirm the rent or deny, if they deny nothing happens and if they confirm the database is updated. Figure 4.4 shows a printout after a successful rent.

```
The student has 1 rented and is eligible to rent the desired item. Proceed? (y/n)
y
student: 32910
is now renting instrument: 376
```

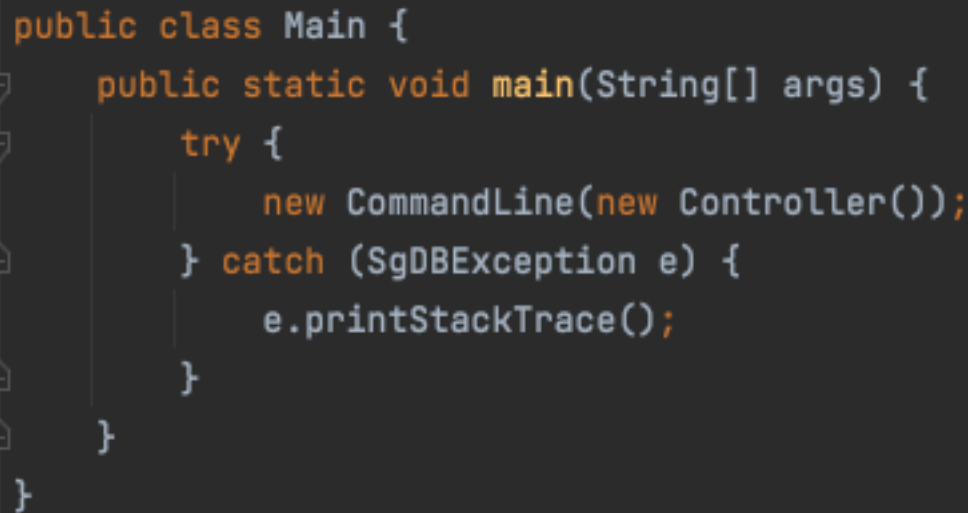
Figure 4.4 a printout after a successful rent operation.



```
terminate 376
Successfully terminated rental for: 376
```

Figure 4.5 a printout of a successful termination.

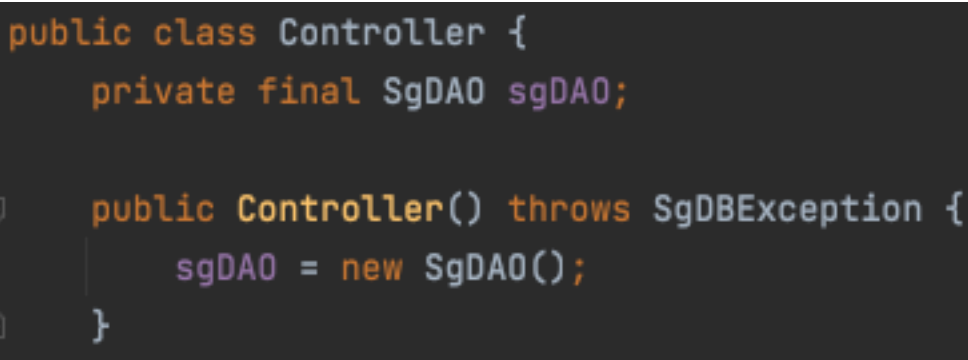
As for the previous command for termination of a rental a student inputs the command terminate followed by the id of the instrument they want to terminate its rental contract.



```
public class Main {
    public static void main(String[] args) {
        try {
            new CommandLine(new Controller());
        } catch (SgDBException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 4.6 a code snippet of the main class.

As shown in figure 4.6 the main method creates an instance of the command line class, which takes parameters to the constructor which is a controller.



```
public class Controller {
    private final SgDAO sgDAO;

    public Controller() throws SgDBException {
        sgDAO = new SgDAO();
    }
}
```

Figure 4.7 a code snippet of the Controller classes constructor.

When a new instance of the controller is initiated, a new instance of the sgDAO Sound-Good-Database-Access-Object is initiated, this object is the object responsible for updating and retrieving data from the database. See figure 4.7


```
getInstruments = connection.prepareStatement(  
    sql: "SELECT * FROM instruments WHERE instrument = ? AND is_rented = FALSE;"  
);
```

Figure 4.8 a code snippet showing the SQL prepareStatement.

The prepared statement in figure 4.8 shows an SQL statement that shows all instrument from a specific type, that are available for rent. The type inserted by the student and is passed in with the java program in place of the question mark.

```
public ArrayList<Instrument> getInstruments(String type) throws SgDBException {  
    ResultSet res = null;  
    try {  
        getInstruments.setString( parameterIndex: 1, type);  
        res = getInstruments.executeQuery();  
        ArrayList<Instrument> i = new ArrayList<>();  
        while (res.next()) {  
            i.add(new Instrument(  
                res.getString( columnLabel: "instrument_id"),  
                res.getString( columnLabel: "instrument"),  
                res.getString( columnLabel: "brand"),  
                res.getDouble( columnLabel: "price"),  
                res.getBoolean( columnLabel: "is_rented"))  
            );  
        }  
        connection.commit();  
        return i;  
    } catch (SQLException throwables) {  
        handleDatabaseException("Could not fetch instrument", throwables);  
    }  
    return null;  
}
```

Figure 4.9 a code snippet of the getInstruments method in the sgDAO.

The method will fetch all instruments that meet the requirements and create a object of the type instruments and then add it to an ArrayList which will be returned to the controller and in its turn to the view to be displayed.

Above is a clear explanation of the flow throughout the program, I do not see any value in adding more pictures of the code that can be viewed in the git repository and explain more of it.

6.3 Discussion

After finishing the application, me and my project partner went over all the criteria needed to satisfy a high-grade project and checked that we have met them, requirements include implementing the MVC pattern of design which we have done, we do not have any duplicate code, the program is easy to understand and so is the code, as mentioned in the requirements “relatively easy” .

As explained in the result section above the application is divided into 5 packages controller, integration, model, startup and view. Naming convention for the whole Java application is followed i.e. camelCase.

One flaw is found by my project partner in regard to the ACID protocol is in the function `terminateRental`, the flaw regards the Atomic part, that is a transaction is either executed entirely or not at all, the problem is that we have two separate SQL statements, first one for archiving the rental in the `rental_archive` table and the other to update the `instrument_rental` table, solutions for this problem can be either combining the two statements in a one or not committing the first change from the java program until the second SQL statement is committed, in such a case if it were to fail in second statement it would just roll back and not commit anything to the database.