Project Report

Data Storage Paradigms, IV1351

Dennis Hadzialic

denhad@kth.se

11/01/2022

1 Introduction

This project was divided into 4 tasks. Each task has several subtasks. I and my partner decided that this is a course where it really benefits from working in a group. There need to be discussions and there is not as much right or wrong, but much is how the person incorporates the requirements. The same was in IV1350-course (Object-oriented design), where we submitted everything individually but discussed our solutions a lot with other students. I and Lucas discussed many solutions to the different problems and when we found something we both agreed on, we chose it as a "final answer". This is also a reason why my and Lucas' s report is similar, sure the text is different, but its content is almost the same.

In the first task, the assignment was to create a conceptual model of Sound Goods music school database, when designing the database, the developers (Dennis and Lucas) had to consider the requirements. When the conceptual model was finished the second task was to design the logical/physical model. This is to be able to realize the database we designed from the first task. Make all the attributes columns with a specific data type. The third task was to query information from the database that the Sound Good music school will use to see information/statistics from the data gathered in their database. An example is to see how many lessons/classes there has been on average per month during the last year. The last task (task 4) wraps everything up. It adds a semi-friendly user interface (a command line) in which the user can interact, list available instruments, rent them and terminate active rentals to the database. This last step brings the whole database to life and makes the user interact with it.

2 Literature Study

The literature study for all tasks will be under this section and will not be divided into subsections. I have divided the literature into two parts. One "general" and "specific" could be specific for a task or similar.

The general literature would be the following:

- Canvas, webpage
- "*Fundamentals of database systems 7th edition*", Elmasri and Navathe.
- PostgreSQL documentation

The task-specific resources are the following:

- Task 1 only used the general literature and Leif' s lecture on inheritance IE-symbols (where he is an IE-symbol and the UML equivalent).
- Task 2 takes the most information from the general literature I would also say.
- Task 3 also used general literature mostly but specifically, there was a good lecture to watch. The one about SQL. I did not find this necessary but was quite nice to watch so one can once again become familiar with it. Otherwise, we checked a lot of PostgreSQL docs and on Stack Overflow to see how two queries can be written into one.
- Task 4 was the one with only one resource, the Transaction ACID lecture and the one with how to set up JDBC. Otherwise, it was muscle memory from IV1350.

3 Method

3.1 Task 1

After conducting the necessary research and reading the requirements, I and my project partner began designing the conceptual model. We used Astah for the design because we had previously used it and because it is the program used in the teacher's lectures. First, we created a list of all the requirements needed from the requirement text on the project site on canvas.

We began by identifying nouns from the provided requirements, which entails going through the text and marking each noun and adding it to the design as an entity. Once we had all the nouns from the text, we moved on to thinking of entities that may be required but do not appear as nouns in the text requirements. This step is also known as category list search, and it entails thinking of various situations or categories and determining if we have all the necessary entities. The next stage was to eliminate any extraneous entities from the design, and then start thinking about attributes, such as which entities have attributes and which entities may be used as attributes in other entities. We start looking for relations between entities, associations between entities, and how they link once we've located all the entities we require and their qualities. We label each association to indicate the logic behind the relationship.

Finally, because there is no need to have a table without data columns, we delete any entities without attributes. We define cardinality for the attributes, whether they are permitted to be without value if they are unique or may be duplicated, and the data types for them.

Working together on different designs from the same requirements and then comparing the designs to see what one has missed and how the text was interpreted to avoid mistakes or misinterpretation was a method that we thought was helpful. After comparing designs, we then created one design as a final design.

3.2 Task 2

The same graphical editor as in task 1 is utilized here. No additional software is utilized, Astah. PostgreSQL is the database management system. I began by constructing a table for each entity in the figure 1.1 diagram. After establishing the tables, I added a column for each attribute in the tables that have a cardinality of no more than one. In the logical/physical model, any attributes with a cardinality of more than one is replaced with a table. Following the creation of the necessary tables, I began to consider the data type to be used for the columns. While it is typically the responsibility of the client to specify the data they want to store in the columns, since this is a school project, I decided to choose the data types after consulting with my project partner. Following that, I started assigning Primary Keys to strong entities, with surrogate keys for all primary keys, and then foreign keys.

After finishing work on the individual tables, I started thinking about relationships between them, much like in the prior challenge. The difference here is that one table cannot have many-too-many links with another table (notice that I am using the word

table to describe the actual table and relationship to describe the connection between two tables).

The logical/physical table relationship is created in multilabel steps: first, create the relation; second, assign the PK of the strong entity as an 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; and finally, consider the FK constraints, such as what should happen in case of updating or deletion of the weak entity. The last step is to change the many to many relations from the table in figure 1. If no other workarounds can be found, a cross-reference table is required. A cross-reference table acts as a solution to be able to map the multiple rows of a table to each other. The PK assigned to the cross-reference table is either its own surrogate PK or two combined FKs from the two tables, which is the method used in this design.

3.3 Task 3

PostgreSQL was once again used as the DBMS in this task too.

The tools used to develop the queries was DataGrip, from Jet Brains and lastly, the terminal found in macOS Monterey. The reasoning for choosing PostgreSQL over any other DBMS has to do with the course IV1351 using it through its lectures. This gives us some similarity and therefore chose it. The reasons for using DataGrip are many. Very easy to display data from the different tables. Easy building of SQL queries. It gives you quick options to alter results and statements, and the programmer (us) can very easily see relationships between tables while changing the database design.

The DDL script for generating the database was created by generating it through Astah. Many programs have these types of features. IntelliJ from Jet Brains can generate UML diagrams from a Java project and Data Grip has plugins that can generate a diagram from a DDL script. This had to be tweaked a little to satisfy all the requirements from the third task. An example of problems Astah had that did not meet the requirements was that it did not manage to generate a script for tables that had inheritance as intended.

The next few steps are straightforward; I began reading about the database queries that must be performed and began conducting research accordingly. This step consists primarily of internet research on the specific DBMS and what functions are built-in and which functions must be created to extract the data requested. More columns were occasionally needed to exhibit fluid design, such as the price column for the table lessons, and other times full tables were needed to meet the criteria for the higher-grade jobs; more on the individual tables can be found in the result section.

Finally, I needed to write a script to enter data so that I could test whether the SQL queries were working.

3.4 Task 4

I used IntelliJ and DataGrip which are both IDEs from Jet Brains. The reason I used IntelliJ instead of NetBeans or something similar is mostly that I am familiar with the

IDE and the perks of auto-generating a diagram of the Database/Classes on the fly while modifying it and debugging. DataGrip is also a tool that made creating SQL queries a breeze, viewing results and modifying tables/columns very quickly. As for DBMS, we used PostgreSQL, which is recommended to use throughout the whole course.

When solving the following problem/assignment I started by reading thoroughly the requirements one by one. Because our group wanted to complete the higher-grade task, we needed to implement MVC and Layer patterns and implement them correctly. There was a lecture on Database Applications which made this part much easier because I thought that our application would be similar from the architectural standpoint. The last part for making this a higher-grade rated assignment was to make the code easy to understand and not have repeating code. This is achieved by analyzing a task before starting to write the code. I started by illustrating with pen and paper (iPad in my case) how the different classes will interact with each other and try to see if there is a task that is repeating itself then I know I must make it a function that can be called upon multiple times. And to make the code easy to understand I try always to avoid making a function too long. Course IV1350 helped me understand how to divide code into smaller chunks. This is to make it easier to edit later, make it reusable and make it easier for a non-author to understand. The method should only do what the inspector of the code would think the method-name does. So if a method is called *sortInstruments()* it should sort instruments, if the method does something besides that, then have I failed to make the code understandable and logical. The mandatory part for task 4 is that the program should have 3 major functions:

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.

To solve these problems the user should need an interface to interact with (command-line/console). The console should then call a controller which interacts with the SoundGoodDAO which will retrieve/update data from the database accordingly.

4 Result

4.1 Task 1

Figure 1.1 illustrates the conceptual model that we created using the process above in (3.1); it is a modest model compared to other models created by other students in the same class using the same requirements, but we believe it has all the necessary information. The model depicts where new applicant information will be kept and how staff will be able to manage them. For example, to process a new application, staff must be able to know if immunizations are available, which they may view in the schedule entity. The number of spaces for a lesson, the instrument used for the class, the genre, the student level, the time for group and ensemble lessons, and the appointment for individual lessons may all be viewed. Staff may set appointments that are then entered into the schedule, as well as lookup immunizations for new applicants. Instructor remuneration is dependent on the number of lessons they provide, while student payment is based on the number of lessons

they take, whether they rented an instrument from the school, and if they have a sibling attending the school, which might result in a discount. Renting instruments is represented as a single entity to indicate that students can rent instruments; this entity will be altered in the following assignment for greater clarity; for now, a note has been inserted to retain the business rule that a student can only have two instruments at a time. Student, applicant, and teacher are all descendants of the entity person, which contains all a person's information.

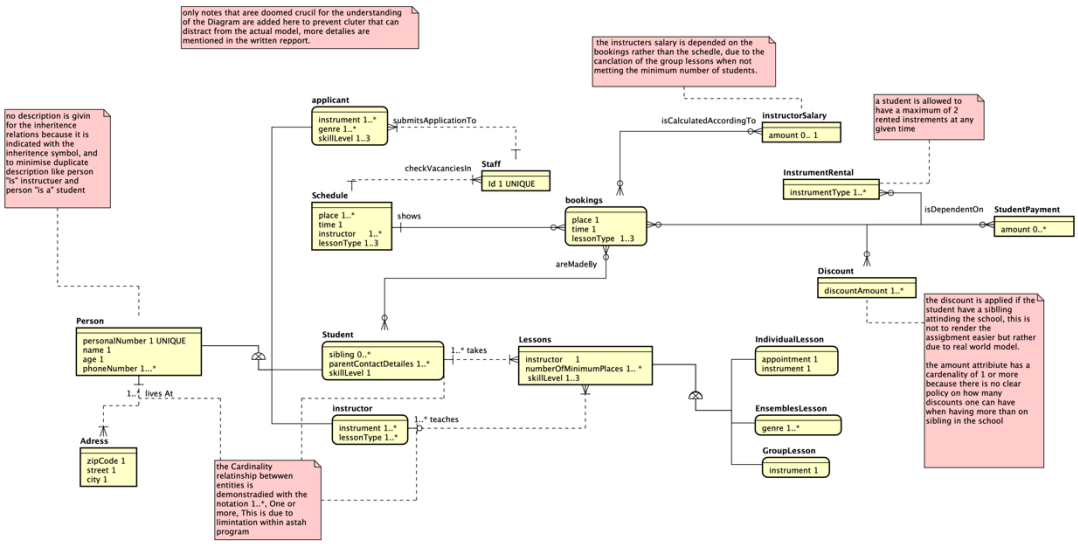

Figure 1.1: The conceptual model from task 1.

4.2 Task 2

The diagram in figure 2.1 is the result of task 2, as shown below. Some design changes were made in comparison to the diagram from task 1, for example, the address entity was removed and added inside the person entity, which is denormalization in some ways, but from the designer's perspective, it is a better fit.

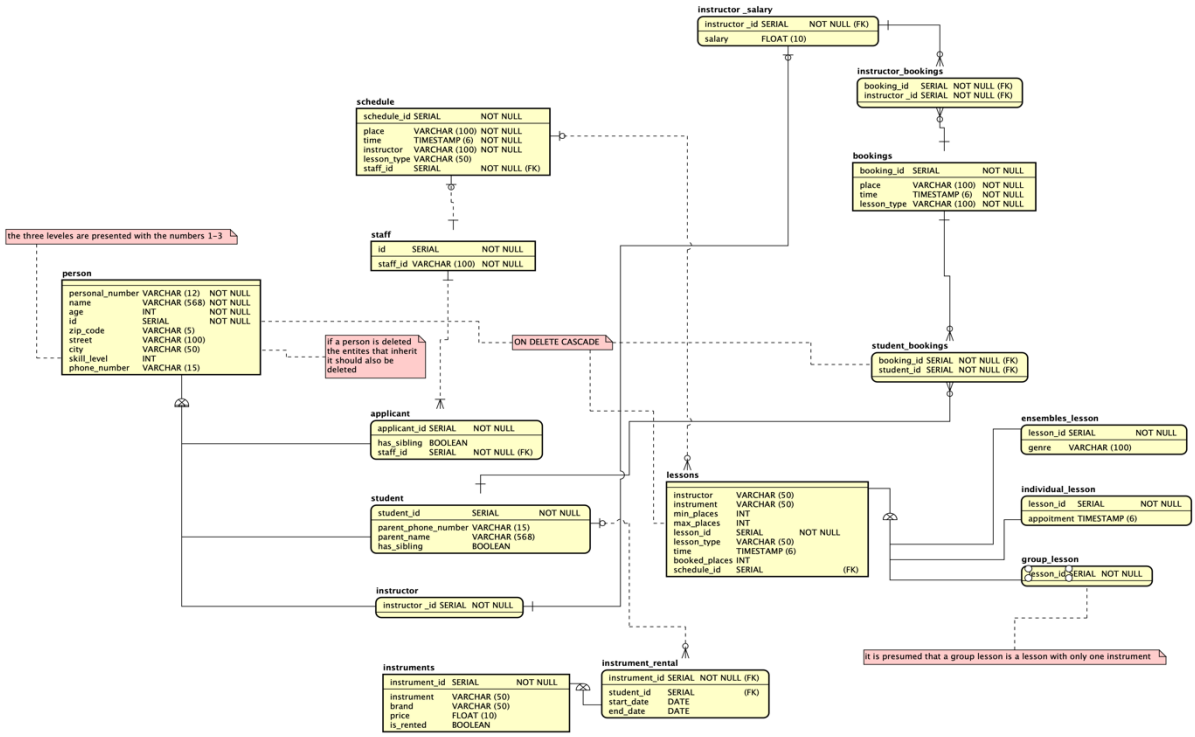

Figure 2.1: logical and physical diagram from task 2

To solve the many-to-many relations from Task 1, two new cross-reference tables are created: one called student bookings and the other instructor bookings. They each use a mixture of FKs as their primary key to solving the rows between the entities booking—student and booking—instructor salary, respectively. All ids` are of the data type SERIAL since we are using Postgres and it is a native data type for the DBMS; all ids´ are NOT NULL because they are used as PK for the entities.
The name attribute has a maximum value of 568 because it is the longest recorded name, and there is no client to consult on the data type. There is no PK for entities person or lesson because it is not interesting to search for a person in the database or for a lesson entity, but it is relevant to search for student or instructor and a specific type of lesson. The skill_ level property is a string that represents the three levels that a student/applicant might have: beginner, intermediate, and advanced. The has sibling attribute contains an integer value representing sibling count, where a student receives a discount percentage based on the number of siblings. This modelling does not make the task easier; rather, it is based on real-world applications. For example, if you are a member of a school/gym, you can frequently receive a discount if two people from the same household are members, so it is reasonable to assume the same for this situation.

4.3 Task 3

The results of task 3 can be found on this GitHub-repository. There is a *README.md*-file that will have instructions on how to set up the database, insert it with the correct data and query the database for the tasks in this part of the project.

The first query is showing the number of lessons given per month during a specified year. We select the month and number of lessons where they have the same interval (month). This is illustrated, see figure 3.1.

```
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 printout of the query that retrieves the result of how many ensemble lessons were given per month.

The second query is like the one in figure 3.1 but instead of counting ensemble lessons it will count individual lessons, see figure 3.2.

```
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 printout of the query that retrieves the result of how many individual lessons are given per month.

The last of these three similar queries is the same but counts group lessons instead. See figure 3.3.

```
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 printout of the query that retrieves the result of how many group lessons are given per month.

There is the last one that does the same as the one above but ignores type (i.e., if it is a group lesson or individual lesson). See figure 3.4.

```
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 printout of the query that retrieves the result of how many lessons is per month generally.

These queries listed above are the solutions for the first mandatory part of task 3. We did

not put much effort into mock data at this point which is why the results look a bit "lack-lustre".

The second part of the task is to do the same as above but give the average per month this is done by instead of selecting the month, we instead selected the year and divided the results by 12 (rounding the number with 2 decimals). See figure 3.5 for the query where we retrieve ensemble lessons. See figure 3.6 for the query where we retrieve individual lessons. See figure 3.7 for the query where we retrieve for group lessons. Lastly, see figure 3.8 for the query that does the same as the queries above except that it ignores lesson-type.

```
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 printout of the query that retrieves the result of an average number of ensembles lessons 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 printout of the query that retrieves the result of an average number of individual lessons 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 printout of the query that retrieves the result of an average number of group lessons 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 printout of the query that retrieves the result of an average number of lessons.

To solve the next task, we needed to create two queries in one. The first part is to select all the instructors and count them from lessons where the date matches the current month. Step to is to filter out all instructors that have more than 2 lessons that month. This can be done with the "WITH" command and making the first query-result a variable that we can later filter on, see figure 3.9.

```
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 printout of the query that retrieves the result of the number of lessons an instructor has given the current month, excluding all that have less than 3 lessons.

The next task should list all ensembles held during the next week, sorted by music genre and weekday. For this query, we need to make a CASE statement, which works similar to a series of if-else statements. If it is fully booked (i.e., max_places = booked_places) then "fully booked" should be returned or something accordingly. We first check if it is fully booked else if it has 2-1 places left else return that there are more than 2 places left. See figure 3.10 for the entire query.

```
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_avaliability 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_avaliability
------------+---------------+------------+-------------+------------+---------------------+----------------------------
         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 printout of the query that retrieves the result of ensembles_lessons for the next upcoming week.

For the next assignment, we needed to be able to archive lessons. So that when a lesson is no longer active it will be sent to an archive table that stores all the information about the lesson. We had to expand the database with a new table called *lessons_archive*. This is quite easily done with the Insert command. See figure 3.11 statement.

```
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 printout of the statement that inserts a lesson to the archive (lessons_archive table), part of the higher-grade task.

4.4 Task 4

The link to the GitHub repository for the following program. When starting the program, the user is prompted with an interface that lists the possible commands, see Figure 4.1.

```
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 printout of how the user is presented with the options of the program.

For the first function which is to list all available instruments of a certain type, the user needs to type “li <type of instrument>”, when the user has pressed enter it will be presented with the results as follows, see Figure 4.2.

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

Figure 4.2: A printout of how the user has presented with data listing available instruments of type drums.

After the user has executed a command, it will return to the prompt of figure 4.1. If the user chooses to rent an instrument for a student, he/she will use the rent-command. The command takes in two parameters first being the students' id (the one who wants to rent the instrument) and the id of the instrument ("rent <studentId> <instrumentId>"). If the student is not eligible to rent an instrument, they should be informed of that and not be able to proceed, else the user will be prompted with a confirmation if the renting process should be proceeded, see figure 4.3.

```
rent 32910 376

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

Figure 4.3: A printout of a user wanting to rent an instrument (confirmation-screen).

If the user wants to proceed, he/she will input "y" and the database will be updated. The program will return a response if the request was successful or not. See figure 4.3.

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

Figure 4.4: A printout of a user proceeds with the renting process.

The last possible command is to terminate a rental. The user will have to input the "terminate" command followed by an instrumentId as an argument. This will terminate the active rental for the student related to the rental and insert the rental information into the archive. See figure 4.5 for a visual representation of how it will look like.

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

Figure 4.5: A printout of a user terminating an active rental

The program consists of 5 packages: controller, integration, model, startup, view. The startup package includes the main file with the main function. It creates a new instance of the CommandLine-class which takes a parameter to the constructor which is a controller. See figure 4.6.

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

Figure 4.6: A printout of the class Main

The class CommandLine is the class with which the user will interact. It has only one method called "runTime()". It has a while-loop running indefinitely until the user types the "exit" command. The class CommandLine uses another class from view-package called Commands. That class has all the available commands and helps the CommandLine-class fetch the arguments that the user inputs after each command. See figure 4.7 for the Commands class.

```java
public class Commands {
    private String[] command;

    public Commands() {
    }

    public String listCommands() {
        return "1. li <type> (List instruments with specific type)\n" +
                "2. rent <studentId> <instrumentId> (rent instrument with id to student with id)\n" +
                "3. terminate <instrumentId> (terminate rental for the student and instrument)\n" +
                "4. exit\n";
    }

    public String input(String in) {
        this.command = in.split( regex: "\\s+");
        return this.command[0];
    }

    public String getArgument(int arg) {
        if (this.command.length < arg) return null;
        return this.command[arg];
    }
}
```

Figure 4.7: A printout of the Commands class.

The listCommands-method returns a String with all the available commands to the CommandLine class. The input method splits the input string into an array, it splits the string by a "blank space", and then returns the first inputed "word" to CommandLine so it can read what command the user is trying to run. The getArgument-method takes an integer (the wanted argument with the index starting at 0) and returns the argument at that place. This is so the program can easily extract instrumentId or studentId if it is necessary.

The runtime-method uses a switch case statement for evaluating which command the user has typed in, it compares with the string input-method returns from Commands class. See figure 4.8.

```
while (running) {
    try {
        System.out.println(commands.listCommands());
        switch (commands.input(scan.nextLine())) {
            case "li":
                type = commands.getArgument(1);
                ArrayList<Instrument> instruments = controller.listInstrumentRental(type);
                if (instruments.size() > 0) {
                    for (Instrument instrument : instruments) {
                        System.out.println(instrument);
                    }
                } else {
                    System.out.println("Could not find any of those instruments");
                }
                break;
```

Figure 4.8: A printout of a switch case statement.

When "li" is run. The view calls the function listInstrumentsRental with the parameter of type (the type of instrument that the user wants to be returned). The method returns an ArrayList containing the type Instruments, and then the view prints all the returned instruments. See figure 4.8 above and see figure 4.9 for the Instruments class.

```java
public class Instrument {
    private String instrumentId;
    private String instrument;
    private String brand;
    private double price;
    private boolean isRented;

    public Instrument(String instrumentId, String instrument, String brand, double price, boolean isRented) {
        this.instrumentId = instrumentId;
        this.instrument = instrument;
        this.brand = brand;
        this.price = price;
        this.isRented = isRented;
    }

    @Override
    public String toString() {
        return "{" +
                "instrument='" + instrument + '\'' +
                ", brand='" + brand + '\'' +
                ", price=" + price +
                " SEK}";
    }
}
```

It has 5 instance variables that correspond with the Instrument-tables columns.

When the Controller class is instantiated, it will create a new instance of the object SgDAO, which will be the SoundGoodDAO (Database-Access-Object). That object will be responsible for retrieving/updating data to the database. It does this in the constructor, see figure 4.10.

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

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

Figure 4.10: A printout of the Controller classes constructor.

If the object sgDAO cannot connect to the database it will throw an SgDBException, which the main method will catch.

The controllers' listInstrumentRental method tries to return the instruments that sgDAO will return or throw an exception if something went wrong. See figure 4.11.

```
public ArrayList<Instrument> listInstrumentRental(String type) throws InstrumentException {
    final String baseError = "Query for listInstrumentRental with type " + type + " is unsuccessful";
    if (type == null)
        throw new InstrumentException(baseError);
    try {
        return sgDAO.getInstruments(type);
    } catch (SgDBException e) {
        throw new InstrumentException("Did not manage to update database");
    }
}
```

Figure 4.11: A printout of the listInstrumentRental method in class Controller

When the SgDAO class is instantiated, the constructor is called and will try to connect the object to the database. See figure 4.12 and 4.13.

```
public SgDAO() throws SgDBException {
    try {
        connect();
    } catch (Exception e) {
        throw new SgDBException("Connection to database failed", e);
    }
}
```

Figure 4.12: A printout of SgDAOs constructor.

```
private void connect() throws SQLException {
    Properties properties = new Properties();
    properties.setProperty("user", "postgres");
    connection = DriverManager.getConnection(DATABASE_URL, properties);
    connection.setAutoCommit(false);
    prepareStatements();
}
```

Figure 4.13: A printout of the connect method.

The connect method is also called the prepareStatements method in the end but I will go through it later. When getInstruments is called from the controller it will fetch all the instruments where "instrument" is the type (instrument-type user wants) and where is_rented is false because the user only wants available instruments to be listed. See figure 4.13 for the query.

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

Figure 4.13: A printout of the query to select all available instruments of a certain kind.

The method will fetch all the instruments that meet the requirements and instantiate a new object of type Instruments which will be added to an ArrayList and later returned to the controller which will return into the view for the data to be displayed. See figure 4.14.

```
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 4.14: A printout of the method getInstruments in the SgDAO class.

When renting an instrument there are two separate methods that are called in the SgDAO class. One for counting the number of rented instruments by a student and another which updates the database with the relevant information. The controller sends the first request to the SgDAO class, to retrieve the number of active rentals that the student has. Method countRentals only takes one parameter (studentId) and returns the number of active rentals

the student has. See Figure 4.15.

```
public int countRentals(String studentId) throws SgDBException {
    ResultSet res = null;
    try {
        getNumberOfActiveRentsbyStudent.setString( parameterIndex: 1, studentId);
        res = getNumberOfActiveRentsbyStudent.executeQuery();
        int count = 0;
        while (res.next()) {
            count = res.getInt( columnLabel: "amount");
        }
        connection.commit();
        return count;
    } catch (SQLException throwables) {
        handleDatabaseException("Could not count rentals", throwables);
    }
    return 0;
}
```

Figure 4.15: A printout of countRentals method in SgDAO class.

The query which is processed in the method selects the number of rows that are returned by the database where the row in table instrument_rental must fulfil the corresponding student_id and that is_rented is true. See figure 4.16 for the query.

```
getNumberOfActiveRentsbyStudent = connection.prepareStatement(
        sql: "SELECT COUNT(*) AS amount FROM instrument_rental WHERE student_id = ? AND is_rented = TRUE;"
);
```

Figure 4.16: A printout of the query for retrieving the number of active rentals a student has.

The controller returns the data it has just recived from the sgDAO object and return it to the view. See figure 4.17.

```
public int checkRentPossibility(String studentId) throws Exception {
    final String baseError = "Could not check availability for student,";
    if (studentId == null) throw new Exception(baseError + " studentId was 'null'");
    try {
        return sgDAO.countRentals(studentId);
    } catch (SgDBException e) {
        throw new Exception("Was not able to fetch data from databse", e);
    }
}
```

Figure 4.17: A printout of the method checkRentPossibility.

If the student has less than 2 active rents the view will prompt the user to decide if it wants to proceed with the rent or not. As shown in figure 4.4. Otherwise, if the student cannot rent an instrument the user will be brought back to the main screen. If the user proceeds with the renting process the controller through the method rentInstrument which takes studentId and instrumentId as parameters. See figure 4.18.

```
public void rentInstrument(String studentId, String instrumentId) throws InstrumentException {
    final String baseError = "Could not rent instrument, ";
    if (studentId == null) throw new InstrumentException(baseError + "studentId was 'null'");
    if (instrumentId == null) throw new InstrumentException(baseError + "instrumentId was 'null'");
    try {
        sgDAO.rentInstrument(studentId, instrumentId);
    } catch (SgDBException e) {
        throw new InstrumentException("Did not manage to update database");
    }
}
```

Figure 4.18: A printout of the method rentInstrument.

When the controller calls sgDAOs method rentInstrument, it should update the database accordingly. This is where ACID properties where even if something fails it will not update the database incompletely. When renting an instrument two datasets from two different tables need to be updated, if the first succeeds and the second one fails, the database will be in an incomplete state which will lead to problems in the future. Instead of having two separate statements be called and risking only one succeeds we needed to rewrite it so that all is in one statement. This was done with the following statement, see figure 4.19 for statement and 4.20 for rentInstrument method.

```
addRentalForStundent = connection.prepareStatement(
        sql: "WITH  rented  AS (" +
                "UPDATE instrument_rental " +
                "SET is_rented = TRUE, student_id = ?, start_date = CURRENT_DATE, end_date = CURRENT_DATE + INTERVAL '1 year' " +
                "WHERE instrument_id = ? RETURNING * ) " +
                "UPDATE instruments SET is_rented = TRUE " +
                "WHERE instrument_id IN (SELECT instrument_id FROM rented);"
);
```

Figure 4.19: A printout of the SQL statement responsible for renting an instrument, credit for the idea goes to thread on StackOverflow.

```
public void rentInstrument(String studentId, String instrumentId) throws SgDBException {
    try {
        addRentalForStundent.setString( parameterIndex: 1, studentId);
        addRentalForStundent.setString( parameterIndex: 2, instrumentId);
        addRentalForStundent.executeUpdate();
        connection.commit();
    } catch (SQLException throwables) {
        handleDatabaseException("Could not rent instrument", throwables);
    }
}
```

Figure 4.20: A printout of the method rentInstrument in the SgDAO class.

When the user wants to terminate a rental, he/she will input the command "terminate <instrumentId>" which will call the controllers' method terminateRental. The method calls to methods in sgDAO, one which adds the rental to the instruments-archive and the second one which terminates the rental. See figure 4.21.

```
public void terminateRental(String instrumentId) throws InstrumentException{
    final String baseError = "Could not terminate rental, ";
    if(instrumentId == null) throw new InstrumentException(baseError + "instrumentId was 'null'");
    try{
        sgDAO.addRentalToArchive(instrumentId);
        sgDAO.terminateRental(instrumentId);
    } catch(SgDBException e){
        throw new InstrumentException("Did not manage to update database");
    }
}
```

Figure 4.21: A printout of the terminateRental method in the controller class

The first step is adding the rental to the archive which is just inserting data with the following query in figure 4.22.

```
insertRentalToArchive = connection.prepareStatement(
        sql: "INSERT INTO rental_archive (instrument_id, instrument, brand , price, student_id, start_date, end_date) " +
             "SELECT instrument_id, instrument, brand , price, student_id , start_date, CURRENT_DATE " +
             "FROM instrument_rental WHERE instrument_id = ?;"
);
```

Figure 4.22: A printout of the statement which inserts rental to archive

The second step is to remove the rental from the student in the database which is illustrated in figure 2.23.

```
removeRentalForStudent = connection.prepareStatement(
        sql: "WITH  rented  AS ( " +
             "UPDATE instrument_rental " +
             "SET is_rented = FALSE, end_date = CURRENT_DATE, student_id = null " +
             "WHERE instrument_id = ? RETURNING * ) " +
             "UPDATE instruments SET is_rented = FALSE " +
             "WHERE instrument_id IN (SELECT instrument_id FROM rented);"
);
```

Figure 4.23: A printout of the statement which removes a student from a rental

The following figure shows how the two methods look like that are called from the controller, see figure 4.24.

```
public void terminateRental(String instrumentId) throws SgDBException {
    try {
        removeRentalForStudent.setString( parameterIndex: 1, instrumentId);
        removeRentalForStudent.executeUpdate();
        connection.commit();
    } catch (SQLException throwables) {
        handleDatabaseException("Could not terminate rental", throwables);
    }
}

public void addRentalToArchive(String instrumentId) throws SgDBException {
    try {
        insertRentalToArchive.setString( parameterIndex: 1, instrumentId);
        insertRentalToArchive.execute();
        connection.commit();
    } catch (SQLException throwables) {
        handleDatabaseException("Could not add rental to archive", throwables);
    }
}
```

Figure 2.24: A printout of the methods terminateRental and addRentalToArchive.

To make sure that the database is not incomplete, one statement fails and the other does not. I needed to handle that in the catch-clause with the method handleDatabaseException. It will roll back the database, so action is not incomplete. See figure 4.25.

```
public void handleDatabaseException(String message, Exception e) throws SgDBException {
    try {
        connection.rollback();
    } catch (SQLException throwables) {
        throwables.printStackTrace();
    }
    if (e != null) throw new SgDBException(message, e);
    else throw new SgDBException(message);
}
```

Figure 2.25: A printout of the handleDatabaseException method.

5 Discussion

5.1 Task 1

” *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-mentioned phrase is the first criterion, which we meet by having the schedule entity, which allows workers to check for openings, and the applicant entity, which stores applicant data.

To reduce repatriation and optimize database normalization, three types of courses are represented by three entities that inherit mutual properties. A lesson is not offered unless a certain number of students attend, as determined by the min/max number of students' attributes. This signifies that inherits are used in at least one location to complete the higher-grade criteria. The conceptual model in figure 1.1 models all the information mentioned in sections 1.1 and 1.2 of the project website on canvas.

The crowfoot notation is used in the conceptual model, with certain notes suggesting a different cardinality owing to a restriction in the Astah-software, which makes it impossible to define cardinality such as 1...*. The model contains 16 entities in total, which is enough to handle all the data that needs to be stored in the database; this is regulated by ensuring that all the data that needs to be recorded in the database is accounted for.

As previously stated, inheritance was used to avoid duplication of attributes in the various

lesson entities. For example, all lessons have an instructor, several participants, and a skill level, so those are the attributes placed in the lessons entity to be inherited by the other three entities. Individual lesson entities have two unique attributes: appointment and instrument, ensembles lesson entities have one unique attribute "genre," and group lesson entities have the property instrument.

5.2 Task 2

Task 2 is about creating the database that will be used in Task 3, but after working on the project, I believe they should be combined into one larger part, because after creating the model shown in Figure 2 and beginning the third task, many changes were made that impacted the design to the point where it was no longer the same design, and we had to go back and forth and change it.

When designing which datatypes should which attribute be, we first thought (as earlier mentioned) that it would be the client' s choice (Sound Good). But in this case, the client isn' t very experienced in this field (I assume). So, it is up to us/me to engineer and design a solid solution. Personal numbers and identifications are better stored as strings, even though they are numbers, to avoid mistakes later. For example, someone born in the year 2000 would have a personal number with two zeros, which has no value and will not be represented in an integer data type, whereas a string will present each character without regard to their value, preserving the data. The skill_ level property is a string that represents the three levels that a student/applicant might have: beginner, intermediate, and advanced. True, having the skill level attribute in the person entity means it is inherited by the instructor, making it a redundant attribute; however, the alternative was to have it in two locations, and after examining both options, we chose to have it in the person entity.

5.3 Task 3

What I would like to have changed/added a bit more energy into was the dataset. There is not much data which made it hard to evaluate the results without needing to manually insert more data to be able to validate the results of the queries. According to the project page, it is instructed to discuss the choice of creating views or not. We managed to create everything (working queries) without the need for them.

There is not much to add to the discussion of this task. Me and my partner both felt like this task and the second one could maybe be one bigger task and maybe remove a requirement or two from that task (to not overwhelm the student). We chose to also not to add the rental archive or the lesson archive to the design of task 1 or 2 because we did not see it ass necessary.

5.4 Task 4

The requirements for this task were to list instruments which we have met as shown in the results-section, rent an instrument for a student which is met and terminate a rental, the whole program should also follow the MVC-layering which I would say we also met. Naming conventions are met. All methods and variables have camel-case and classes start with a capital letter. Constants are also upper-case. The names also describe the purpose of the variable or method. When it comes to ACID transactions it is most important in terminating the rental and while writing the report, I saw a big flaw with the program. Because every statement is committed after execution there is a problem with terminateRental. If the first one succeeds and the second one fails, the first change (archiving) is already committed to the database and even if the second one fails it will still be in the archive. Because there is no time left to change it because I am writing the last part of the report very late this cannot be changed.

Another thing that I was unsure about was when writing the models (Instrument class) I thought of making a DTO of it. But because there is no way of changing the data of the object (setters) I thought it was an unnecessary class to make.

I learned in this task about ACID and how to access/update/retrieve data from a database with a program (Java program). This was the largest task in my opinion and if I could do something different, I would fix the issue I saw whilst writing the report (terminate rental). I never encountered the problem under development and do not think a user will be able to encounter it either but if I already put all the work for a rollback in that scenario, it would be nice to use it if the edge-case ever came to be. The purpose of this design is to keep the entities from having null values. There are several 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 but is not preferred because it does not add any more value or aid in the understanding of the diagram as a whole; it will only serve to clutter it more and would result in a database application that is difficult to maintain, that is if a database application is maintained at all. Whereas in the case of inheritance, if an attribute is required for all lesson types, it must be inserted just once in the inherited lessons entity, and the same is true for removal.

6 Comments About the Course

It was a fun course, much like IV1350 which is a compliment. Think the material is great and that the tasks are fun. What I feel is that it is a bit too much work with the assignment if there is an exam too. Just the report took many hours to write and the task their self was quite big. What I would have preferred is maybe either shorten the tasks or make the report in some way smaller, maybe no need for such in-depth results and other sections but maybe only have method and discussion?