Project Report

**Data Storage Paradigms, IV1351**

Dennis Hadzialic denhad@kth.se

11/01/2022

# Tips for Report Writing

## REMOVE THIS SECTION BEFORE SUBMITTING THE REPORT.

*The target audience has exactly the same skills as the author, except they do not know anything at all about the specific application described in the report.*

Consider the following:

## The report must be *centered around the requirements*. Which are they (Introduction), how did you work to meet them (Method), what is the solution that meets them (Result), and how can you be sure they are met (Discussion). This is the IMRaD method.

* **The report must show that you have done the work yourself and that you have understood what you have done. Both of these goals are met by carefully explaining your application (both database and program).**
* Is spelling and grammar correct? Is spoken language avoided?
* Does the report have a good structure with sections, subsections and paragraphs?
* Is the solution clearly explained? Will the reader understand the application? What would you yourself want to know if you read about the application, is that included in the report?
* Is the solution analyzed and evaluated? Are important properties of the application explained? Should there have been more extensive evaluation?
* Is the text clarified with images and/or other figures, and with links to the code in your Git repository? Remember that all figures (images, tables, graphs, code listings, etc) shall be numbered and have a short explaining text.

# Introduction

## This section tells *what* are you going to do.

Explain the task and the requirements on the solution. It’s important to clearly state the requirements. *Also specify which other student you worked with when solving the tasks, or if you worked alone.* Write one single introduction covering all tasks; do not split it into one subsection per task.

# Literature Study

This section must prove that you collected sufficient knowledge before starting devel- opment, instead of just hacking away without knowing how to complete a task. State what you have read and briefly summarize what you have learned. It is your choice if you include literature study for all tasks in the same section, or if you divide this section into one subsection per task.

# Method

## This section tells *how* you solved the task.

Explain how you worked when solving the tasks and how you evaluated that your solution met the requirements. Mention diagram editor(s), IDE(s), DBMS(s) and other tools you used. This section *must be split into four subsections*, one per task. *Do not explain your solution and do not refer to code*, that belongs to the *Result* section.

//START

I used [IntelliJ](https://www.jetbrains.com/idea/) and [DataGrip](https://www.jetbrains.com/datagrip) which are both IDEs from [Jet Brains](https://www.jetbrains.com/). 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 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](https://canvas.kth.se/courses/27118/pages/project#:~:text=at%20the%20page-,Database%20Applications,-.%20The%20following%20must) 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.

# Result

**This section explains *the result* of what you did.**

Present the solution. Explain your code and prove that it meets the requirements. It’s very important to *state each requirement that is met* and explain *how you met it*. It’s also important to include links to your code in your Git repository, and to also include diagrams, see Figure 1, and other figures to illustrate your reasoning. Also, remember that these figures must be referenced in the text. This section *must be split into four subsections*, one per task.

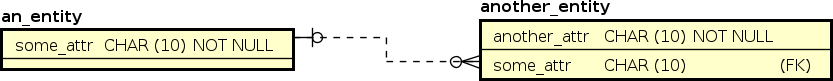


Figure 1: A sample diagram, included to illustrate caption (this text), numbering and reference in text.

// START

When starting the program, the user is prompted with an interface that lists the possible commands, see Figure 4.1.

Text

Description automatically generated

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.



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.

Text

Description automatically generated

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.Text

Description automatically generated

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.

Graphical user interface, text

Description automatically generated

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.

Text

Description automatically generated

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.

Text

Description automatically generated

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.

Text

Description automatically generated

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.

Text

Description automatically generated

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.

Text

Description automatically generated with medium confidence

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.

Text

Description automatically generated

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.

Graphical user interface, text

Description automatically generated

Figure 4.12: A printout of SgDAOs constructor.

Text

Description automatically generated

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.

Graphical user interface, website

Description automatically generated

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.

Text

Description automatically generated

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.

Text

Description automatically generated

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. Se figure 4.16 for the query.

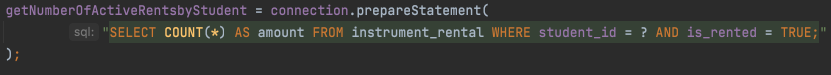


Figure 4.16: A printout of the query for retrieving the number of active rentals a student has.

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

Text

Description automatically generated

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.

Text

Description automatically generated

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.

Graphical user interface, text, application

Description automatically generated

Figure 4.19: A printout of the SQL-statement responsible for renting an instrument, credit for the idea goes to thread on [StackOverflow](https://stackoverflow.com/questions/29898244/postgresql-update-multiple-tables-in-single-query).

Text

Description automatically generated

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

# Discussion

## This section *analysis* the result presented in the previous section.

Summarize the requirements and *clearly state which of them you have met*. What lessons have you learned and what problems did you face? How were the problems solved? Should you have done something differently? This section *must be split into four sub- sections*, one per task.

# Comments About the Course

## This section is optional.

Any comment(s) related to this course offering or to coming offerings is much appre- ciated. *Please also tell approximately how much time you spent on the assignment*, including lectures and labs. This is of great help for course evaluation.