Programmatic Access

Data Storage Paradigms, IV1351

Group 50

2025-01-03

Project members:
[Allan Al Saleh, Allan2@kth.se]
[Mostafa Faik, mfaik@kth.se]
[Derfesh Mariush, derfesh@kth.se]

Declaration:

Introduction

In this assignment, we developed a command-line application for managing instrument rentals for Soundgood, a musical instrument rental service. The application includes functionalities to list available instruments, rent instruments, and terminate rentals, with a focus on database access and transaction handling.

Literature Study

To gain the foundational knowledge for this task, we relied on these resources:

- Lecture video(Database Transactions): We got most of our theoretical understanding of transactions from here. Everything ACID, schedules, operation conflicts, exclusive and shared locks, anomalies and weaker isolation levels.
- Lecture video(Introduction to JDBC): Teaches the underlying theory behind bridging the database from SQL to java. Everything in the video was used, without this video, performing the assignment would be impossible.
- Lecture video(Architecture and Design of a Database Application): This video taught us how to construct a layered database. Specifically, we learned how to program the different layers, what to avoid and special cases to look out for.

Method

We started of with Leif's bank application as the base of code. from there we started to replace different layers to work with our database. To begin with, we remodeled the model layer to contain the tables and attributes for Soundgoods renting service. After that we started working with the integration and controller layer, which was the bulk of the code. The main and view layer only required some small changes and worked without a problem. After creating a test version of the code, we compiled it with maven and debug the code.

Tools

The tools and environment used to construct the program and code were the following:

- IDE: Visual Studio Code, pgAdmin.
- Database: PostgreSQL was used as the relational database management system for its reliability and compatibility with JDBC.
- Version Control: Git and GitHub managed version control and repository hosting, ensuring development traceability and accessibility.

- Build Tool: Maven handled dependencies and compilation, streamlining the inclusion of libraries such as the PostgreSQL JDBC driver.
- User Interface: A command-line interface (CLI) was employed to focus on database functionality without a graphical interface.

Result

The program was designed using a layered architecture following the MVC (Model-View-Controller) pattern, ensuring a clean separation between the different components of the system.

ACID Transaction Handling:

The handling of ACID transactions (Atomicity, Consistency, Isolation, Durability) was achieved by implementing the following principles:

- Autocommit turned off: The program disables the autocommit mode, meaning that every operation is wrapped in a transaction. This ensures that database changes are not finalized until explicitly committed, maintaining atomicity and consistency.
- Commit and Rollback: For each operation (such as renting an instrument, terminating a rental, etc.), the program calls commit() if the transaction is successful, ensuring the changes are persisted in the database. If any error occurs during the operation, the program uses rollback() to undo the changes, maintaining the integrity of the database.
- SELECT FOR UPDATE: To avoid race conditions when multiple users might try to rent the same instrument, the program uses SELECT FOR UPDATE to lock the rows involved in the transaction until it is committed or rolled back, ensuring isolation between transactions.

In addition, the database operations are handled by a DAO (Data Access Object) layer, which is solely responsible for performing CRUD operations. The controller manages the business logic and ensures that all conditions are met before invoking the database operations.

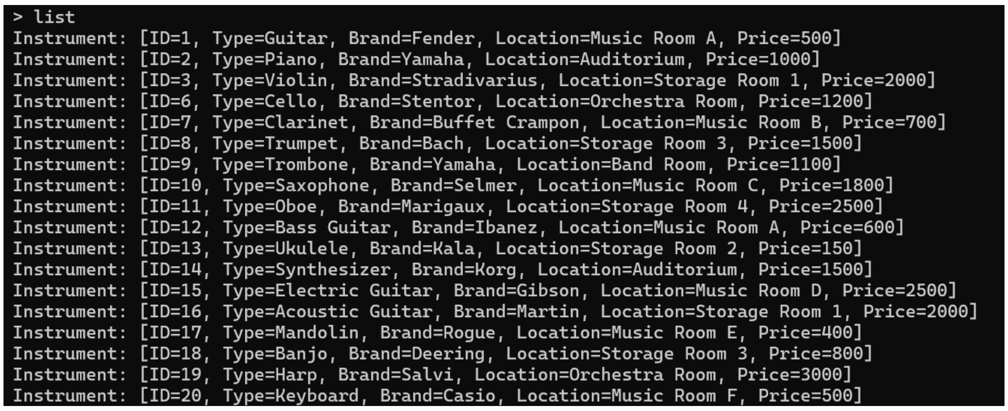
Database and Testing:

The database schema includes tables for instruments, rentals, and students. The database scripts to create the schema and insert initial data are included in the repository. To test the solution, the following steps are required:

- Execute the script that creates the database schema.

- Run the script that inserts sample data into the database.
- Run the application to interact with the database via the command-line interface.

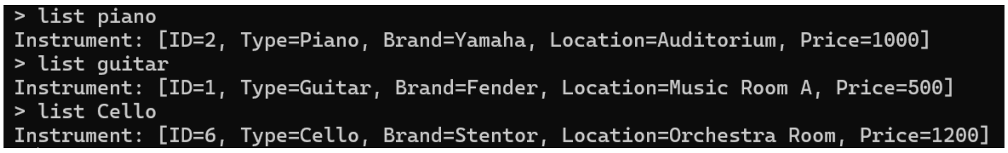
The following picture shows the list of the available instruments that students can rent. To rent an instrument, we simply write and run ”rent 1 1 2024-12-30”. This command equals with ”rent studentID instrumentID end_date” and it means that student with ID 1 is renting (want to rent) instrument with ID 1, and specifies the end_date (until when he wants to rent it). To show the list of available instruments, we simply write and run list (as shown below).



```
> list
Instrument: [ID=1, Type=Guitar, Brand=Fender, Location=Music Room A, Price=500]
Instrument: [ID=2, Type=Piano, Brand=Yamaha, Location=Auditorium, Price=1000]
Instrument: [ID=3, Type=Violin, Brand=Stradivarius, Location=Storage Room 1, Price=2000]
Instrument: [ID=6, Type=Cello, Brand=Stentor, Location=Orchestra Room, Price=1200]
Instrument: [ID=7, Type=Clarinet, Brand=Buffet Crampon, Location=Music Room B, Price=700]
Instrument: [ID=8, Type=Trumpet, Brand=Bach, Location=Storage Room 3, Price=1500]
Instrument: [ID=9, Type=Trombone, Brand=Yamaha, Location=Band Room, Price=1100]
Instrument: [ID=10, Type=Saxophone, Brand=Selmer, Location=Music Room C, Price=1800]
Instrument: [ID=11, Type=Oboe, Brand=Marigaux, Location=Storage Room 4, Price=2500]
Instrument: [ID=12, Type=Bass Guitar, Brand=Ibanez, Location=Music Room A, Price=600]
Instrument: [ID=13, Type=Ukulele, Brand=Kala, Location=Storage Room 2, Price=150]
Instrument: [ID=14, Type=Synthesizer, Brand=Korg, Location=Auditorium, Price=1500]
Instrument: [ID=15, Type=Electric Guitar, Brand=Gibson, Location=Music Room D, Price=2500]
Instrument: [ID=16, Type=Acoustic Guitar, Brand=Martin, Location=Storage Room 1, Price=2000]
Instrument: [ID=17, Type=Mandolin, Brand=Rogue, Location=Music Room E, Price=400]
Instrument: [ID=18, Type=Banjo, Brand=Deering, Location=Storage Room 3, Price=800]
Instrument: [ID=19, Type=Harp, Brand=Salvi, Location=Orchestra Room, Price=3000]
Instrument: [ID=20, Type=Keyboard, Brand=Casio, Location=Music Room F, Price=500]
```

Figure 1: List of available instruments

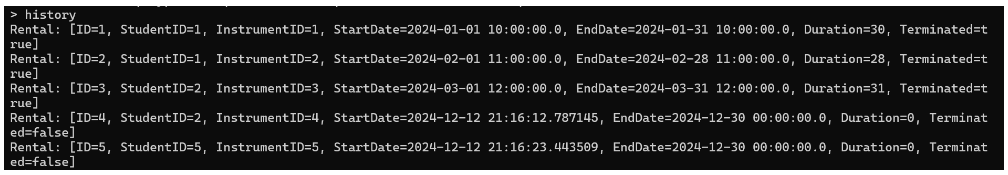
We can also list a type of instrument, and see its price, location and brand. This is used to find/list instruments of the same type, but different price and/or brand (if there exists instruments of the same type). We can do this by simply writing and running ”list name_of_instrument” (as shown below).



```
> list piano
Instrument: [ID=2, Type=Piano, Brand=Yamaha, Location=Auditorium, Price=1000]
> list guitar
Instrument: [ID=1, Type=Guitar, Brand=Fender, Location=Music Room A, Price=500]
> list Cello
Instrument: [ID=6, Type=Cello, Brand=Stentor, Location=Orchestra Room, Price=1200]
```

Figure 2: List type of instruments

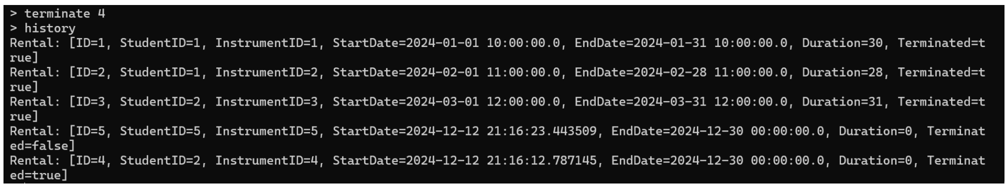
We can also show and list the history of all rentals (both that are terminated and not). This is used to have and save the information of a rental even if it was terminated. This can be done by writing and running ”history”.



```
> history
Rental: [ID=1, StudentID=1, InstrumentID=1, StartDate=2024-01-01 10:00:00.0, EndDate=2024-01-31 10:00:00.0, Duration=30, Terminated=t
rue]
Rental: [ID=2, StudentID=1, InstrumentID=2, StartDate=2024-02-01 11:00:00.0, EndDate=2024-02-28 11:00:00.0, Duration=28, Terminated=t
rue]
Rental: [ID=3, StudentID=2, InstrumentID=3, StartDate=2024-03-01 12:00:00.0, EndDate=2024-03-31 12:00:00.0, Duration=31, Terminated=t
rue]
Rental: [ID=4, StudentID=2, InstrumentID=4, StartDate=2024-12-12 21:16:12.787145, EndDate=2024-12-30 00:00:00.0, Duration=0, Terminat
ed=false]
Rental: [ID=5, StudentID=5, InstrumentID=5, StartDate=2024-12-12 21:16:23.443509, EndDate=2024-12-30 00:00:00.0, Duration=0, Terminat
ed=false]
```

Figure 3: History of Rentals

The following picture shows the updated history list after terminating a rental. In the previous picture, a rental was ongoing for instrument with ID 4 (terminated=false) by student with ID 2. After terminating this rental, the history list is updated (as shown below) and the terminated field is set to true (terminated=true).



```
> terminate 4
> history
Rental: [ID=1, StudentID=1, InstrumentID=1, StartDate=2024-01-01 10:00:00.0, EndDate=2024-01-31 10:00:00.0, Duration=30, Terminated=t
rue]
Rental: [ID=2, StudentID=1, InstrumentID=2, StartDate=2024-02-01 11:00:00.0, EndDate=2024-02-28 11:00:00.0, Duration=28, Terminated=t
rue]
Rental: [ID=3, StudentID=2, InstrumentID=3, StartDate=2024-03-01 12:00:00.0, EndDate=2024-03-31 12:00:00.0, Duration=31, Terminated=t
rue]
Rental: [ID=5, StudentID=5, InstrumentID=5, StartDate=2024-12-12 21:16:23.443509, EndDate=2024-12-30 00:00:00.0, Duration=0, Terminat
ed=false]
Rental: [ID=4, StudentID=2, InstrumentID=4, StartDate=2024-12-12 21:16:12.787145, EndDate=2024-12-30 00:00:00.0, Duration=0, Terminat
ed=true]
```

Figure 4: Updated History

Some things to consider and will lead to a constraint/error:

- A student can not rent more than 2 instruments at the same time.
- A student can not rent an instrument that is already rented by another.
- When renting an instrument, the end date (rent studentID instrumentID end_date) can not be less than the start date and can not be set to be larger/more than 12 months (1 year), because a student can not rent an instrument for more than 12 months (as specified by the school).

Repository:

The source code, along with the database creation and insertion scripts, is available on the following GitHub link, specifically in the folder ”task4”:

https://gits-15.sys.kth.se/derfesh/IV1351-HT24-Data-Storage-Paradigms-Soundgood

Discussion

This section covers the use of the MVC pattern, transaction handling, and the general maintainability and readability of the code.

Code Readability and Understandability:

The program is structured to prioritize clarity and maintainability, with careful attention given to naming conventions and comments. The names of methods, variables, and classes are chosen to be descriptive and self-explanatory. For example, the class names Instrument and Rental clearly indicate their purpose, while method names like findInstruments(), createInstrument(), and updateRental() describe the exact actions performed by the program.

While some subjective aspects of readability are open to interpretation, several steps were taken to ensure that the code is accessible and understandable:

- Descriptive Naming: As mentioned, method and class names reflect their functionality and purpose.
- Modularization: The program is split into logical, well-defined classes and packages, ensuring that functionality is encapsulated and easy to trace.
- Minimal Complexity: The code avoids deep nesting and unnecessary complexity. Each method does one thing and does it well, which is key to making the code more readable and understandable.

Use of MVC and Layer Patterns:

The program follows the Model-View-Controller (MVC) design pattern and layered architecture in a clean and structured manner. The following describes how each layer is implemented:

- Model Layer: The model consists of Instrument and Rental, which represent the data entities of the application. The model classes contain only the data fields, their respective getters and setters, and any necessary constructors. They do not contain any business logic, ensuring that the program adheres to the single responsibility principle.
- View Layer: The view is represented by the command-line interface. It handles user input and output but contains no business logic. All data passed to the view is already processed by the controller. The view simply displays results, such as available instruments, rental statuses, and error messages.
- Controller Layer: The controller layer manages the business logic. It coordinates the actions between the view and model layers, ensuring that all rules are followed before database operations are performed. For example, when renting an instrument, the controller checks if the instrument is available and whether the student is eligible to rent. The controller also manages the transaction logic, ensuring that operations are committed or rolled back depending on success or failure.

- DAO Layer (Integration Layer): The DAO (Data Access Object) layer is responsible for performing CRUD operations on the database. All methods in the DAO layer follow the naming convention of create, read, update, and delete. There is no business logic in the DAO layer. For instance, methods like createRental() and terminateRental() directly map to SQL queries but do not involve any logic beyond interacting with the database.

The separation of concerns is strictly followed, with no business logic in the model or DAO layers, and no database interaction in the controller or view layers. This ensures that each class has a clear and singular responsibility, following good software design practices.

Evaluation of the Program:

- Naming Conventions: The naming conventions are consistently followed. Classes, methods, and variables are named intuitively, which enhances the readability and understanding of the code. For example, method names like terminateRental() clearly convey their purpose, and the class names align with the objects they represent (Instrument, Rental, etc.).
- Transaction Handling: The transactions are correctly handled (as explained in the result section). The program ensures that auto-commit is disabled for all database operations. Each method that interacts with the database calls commit() if the operation succeeds and rollback() if it fails.
- SQL Statements and Transaction Integrity: The SQL queries are carefully constructed to maintain integrity. For example, when an instrument is rented, the program ensures that the student can only rent up to two instruments at a time. The SQL queries are wrapped in transactions, and the logic ensures the proper flow of operations.
- Rentals Marked as Terminated: When a rental is terminated, no data is deleted from the database. Instead, a terminated flag is set to true, marking the rental as terminated without losing any rental history. This ensures that the program retains all historical data while still indicating that a rental is no longer active.
- Duplication of Code: Duplication of code has been avoided throughout the program. Methods are kept short, and reusable logic is encapsulated into functions, ensuring that the program is easy to maintain and extend. For example, separate methods for checking rental availability and checking student eligibility avoid repeating similar logic in multiple places.
- No Business Logic in the DAO: The DAO layer contains only methods for creating, reading, updating, or deleting data from the database. There is no business logic in the DAO layer, adhering to the principle that database interaction should be purely concerned with data access.

- No Business Logic in the View or Controller Layers: Both the view and controller layers are free from any logic related to how the data is stored or retrieved. The view layer handles only the display of information, and the controller orchestrates actions by interacting with the model and DAO layers.
- Clarity and Maintainability: The program is designed with readability and maintainability in mind. The code is well-structured, making it easy to understand and extend. Comments are used to explain key sections of the code, ensuring future developers can follow the logic.

Changes and Improvements:

Feedback we received from Leif:

When we first sent this assignment we received some feedback which we improved upon:

- 1. ”The updateRental method always does the same update, to mark the rental as terminated. A better name would have been something like markRentalAsTerminated, you shouldn't create strange method names just to follow the create/find/update/delete naming convention. There's anyway no business logic in this method, since it doesn't do any checks to see if the rental can be terminated.”
- 2. ”You're not using transactions correctly, since you always commit the transaction at the end of each method in the DAO. This means for example that, in the controller method createRental, the calls to the DAO on lines 71, 76 and 84 are executed in different transactions. Because of this, it's possible for two students to rent the same instrument at the same time.”

  When we enquired about clearification on what to exactly do we recieved this answer:

  ”A better solution is to do as in the bank example, see for example the deposit method in Controller. First, it calls findAccountByAcctNo on line 147, to retrieve the current balance. Specifying lockExclusive true means that findAccountByAcctNo will lock FOR UPDATE, and also that it won't commit. Then, after having deposited, the controller calls a DAO method that just commits the transaction, on line 188.

  If you solve the problem as is done in the bank example, your solution will be accepted. If you solve it by just deleting the commit statements as you suggest, there's a big risk it won't be accepted.”

The changes we made to fix these problems were as follows:

- 1. We did as instructed and changed the methods name to markRentalsAsTerminated. Additionally, we ensured that other parts of the code that call upon the method use the new name.
- 2. The main problem with the code was that we hadn't implemented a good enough mechanism to lock the tables when an instrument was rented. The solution we opted for was to mimic Leifs bank database example.

  We started by changing the methods which were called when "createRental" is triggered from the controller, which is the "findCurrentRentalsByStudent" and "findRentalsByInstrument". These methods normally check that: 1. The student has not rented 2 instruments. 2. The instrument the student wants to rent is available.

  What Leif wanted us to do is to use make sure that these methods lock the tables and won't commit until: 1. The code does everything successfully. 2. The code fails. We did this by adding another input variable to both methods called "lockExclusive". When "lockExclusive" is true, the rows which are selected by the methods become locked. For this case we lock the students row and the instrument. In the controller we implemented a method called "commitOngoingTransactions", which commits the "createRental" after everything is done and unlocks the rows that we have used.