Seminar 4 - Programmatic Access

Data Storage Paradigms, IV1351

Seema Bashir

14/12-2023

1 Introduction

This report addresses the assignment to enhance the functionality of Soundgood's website by detailing the development of specific features related to instrument rentals. The primary focus includes the implementation of key functionalities, such as listing available instruments, renting instruments with student limit constraints, and terminating rentals. The paramount consideration is the establishment of robust database access, with a command line interface deemed sufficient for the task at hand. The program is designed to adeptly manage ACID transactions by appropriately incorporating commit and rollback calls and utilizing SELECT FOR UPDATE when necessary. Additionally, the code is housed in a public Git repository, featuring scripts for creating and populating the database.

Furthermore, this report underscores the commitment to achieving higher-grade standards through meticulous attention to design and architecture. The program is engineered to meet elevated requirements, placing significant emphasis on code clarity, strict adherence to MVC and Layer patterns, and the deliberate exclusion of logic within the integration layer.

In collaboration with Razan Yakoub and Teoman Köylüoğlu, this report details the approach employed and executed, providing an overview of the collaborative results in meeting the specified requirements.

2 Literature Study

In order to prepare for this seminar, chapters 20 and 21.1 in the 7th edition of Fundamentals of Database Systems by Elmasri and Navathe are evaluated.

Emphasizing the significance of proper concurrency control, chapter 20 introduces the concept of ACID transactions and discusses potential challenges in concurrent execution, such as the lost update problem and unrepeatable read problem. Additionally, the book also underscores the essential role of recovery mechanisms in addressing diverse failure scenarios, ranging from system crashes to disk failures, establishing the transaction as

a fundamental concept for maintaining database integrity in dynamic and multiuser environments.

Furthermore,the life cycle of a transaction is highlighted, explaining how it starts, progresses, and either completes successfully or faces potential cancellation. An essential aspect ensuring the system's robustness is explored through the examination of the system log. The log plays a crucial role in tracking transaction operations and changes to data, providing necessary information for system recovery. The concept of a transaction's commit point is also scrutinized, emphasizing the moment when successful operations become permanently embedded in the database. Additionally, the chapter briefly touches upon specific strategies employed by Database Management Systems (DBMS) to replace buffers efficiently, optimizing system performance.

Moreover, chapter 20 further introduces and explores the ACID properties crucial for transactions, emphasizing their enforcement by Database Management Systems (DBMS) through concurrency control and recovery methods. The ACID properties include Atomicity, Consistency preservation, Isolation, and Durability. Atomicity requires completing transactions, and the recovery subsystem handles this. Consistency preservation relies on programmers and the DBMS to ensure a transaction maintains a consistent state. Isolation is enforced by the concurrency control subsystem, isolating transactions during execution. Durability, managed by the recovery subsystem, mandates committed changes to persist in the database.

Furthermore, lectures provided by Leif Lindbäck including, *Introduction to JDBC and Architecture and Design of a Database Application* and *a*re watched and evaluated inorder to grasp an understanding of the steps required to create the JDBC (Java Database Connectivity) API, used to execute SQL statements from a Java program.

The video introduces the importance of well-organized code in software development, aiming for code that is easy to maintain, change, and extend. The concept emphasizes high cohesion within subsystems or layers, each assigned a specific task to ensure clarity and prevent unnecessary code modification. Low coupling between layers is promoted to minimize dependencies, reducing the risk of widespread changes during modifications. The Model-View-Controller (MVC) architecture is presented as a practical approach, dividing code into view, model, and controller components. The MVC pattern is likened to a construction project, where the controller acts as a mediator between the view and model, ensuring organized communication. Additionally, the Layer pattern suggests dividing the program into subsystems, highlighting the data layer, DB Handler layer, and startup layer. Lief also stresses logical organization and the avoidance of direct dependencies from higher to lower layers, concluding that effective code organization is essential for flexibility and comprehensibility in software development.

Furthermore, a lecture on Transactions provided by Paris Carbone is watched and evaluated in order to gain a better understanding of the presence of transactions and the their roles in the database.

The lecture highlights the importance of conflict serializability in maintaining data consistency and preventing conflicts in concurrent transactions. Additionally, the use of

pessimistic locking as a common mechanism, where locks on database objects are acquired before read or write operations to avoid conflicts. The two-phase locking protocol was introduced as a stricter form of conflict serializability, requiring transactions to acquire all necessary locks before releasing any. The lecture emphasized the crucial role of locking mechanisms in ensuring data integrity and preventing conflicts in database systems. The speaker also addressed the choice between pessimistic locking and optimistic concurrency control, highlighting the trade-offs between strong guarantees and performance impact. Moreover, conflict detection and resolution mechanisms were discussed as essential components to ensure data consistency. The overall message was the significance of understanding transactions, conflict serializability, and locking mechanisms for designing reliable and scalable database systems capable of handling concurrent access to data.

Lastly, the "Tips and Tricks 4" document provided vital details regarding the general tips and insights for addressing common challenges in database systems and project tasks. It emphasizes the importance of preventing the lost update anomaly in concurrent transactions by structuring transactions appropriately and acquiring exclusive locks when necessary. The document also advocates for maintaining a clear separation of concerns, particularly by avoiding business logic in classes responsible for database calls. In addition, the document introduces the notion of DAO, which is an acronym for Data Access Object. This design pattern, commonly found in the field of software engineering, serves to abstract and encapsulate access to data sources, such as databases. DAOs provide an interface for executing operations on data without exposing the underlying structure or implementation details of the database to the rest of the application. Typically, these objects define CRUD operations (Create, Read, Update, Delete) and manage various interactions with the database, including tasks like querying, inserting, updating, and deleting data. By incorporating DAOs, we can simplify the logic for accessing data, improve the maintainability of the code, and facilitate seamless modifications to data storage without affecting the broader application. The ultimate goal of these insights is to strengthen data integrity, enhance system maintainability, and support scalability in the development of projects.

3 Method

To intiate, the Eclipse Integrated Development Environment (IDE) was employed as the primary tool for creating a command line user interface to oversee functionalities of the program. The task involved listing available instruments, renting instruments, and terminating active rentals, and it was approached using the Model View Control (MVC) design pattern.

Initiating the development process, an Eclipse project named "SoundgoodMusicSchool" is created. The project revolves around the implementation of the Model View Control (MVC) design pattern structure, which involves the creation of specific packages following a predefined naming convention. Consequently, the program is systematically organized into different layers, each designated for a specialized purpose: model, view,

controller, startup, and integration. This structuring enhances the overall organization and clarity of the project by providing a clear delineation of responsibilities and functionalities within the codebase.

In the project's initial phase, a connection to the local Soundgood music school database was established, and this connection process is managed within the Integration layer through SchoolDOA. To facilitate this connection, the PostgreSQL JDBC driver library was downloaded and seamlessly incorporated into the project. This acquisition empowered the program to execute a diverse range of SQL queries, fostering effective interaction with the underlying database. Within the Integration layer, which plays a pivotal role, methods are exclusively dedicated to executing SQL queries on the connected database. This ensures a seamless interaction with the data storage, facilitating efficient retrieval and manipulation of information.

Following the successful establishment of the database connection, the next pivotal phase focused on setting up SQL prepared statements within the SchoolDAO class. These statements, housing precompiled SQL queries, were instrumental in ensuring the efficient execution of queries within the program context. The SchoolDAO class employs prepared statements to address key functionalities, including obtaining the maximum allowed number of instruments, determining the maximum renting period, listing available instruments based on type, and updating rental information. These prepared statements serve as a streamlined and optimized mechanism for executing SQL queries, significantly enhancing the program's performance.

Moreover, in the development workflow, Visual Studio Live Share played a crucial role in fostering collaboration among team members. By initiating Live Share sessions in Visual Studio Code, the team seamlessly worked on the same codebase in real-time. This approach significantly enhanced the ability to discuss and implement code changes efficiently, contributing to a more synchronized and productive development process.

Lastly, Leif's project, a JDBC application, significantly influences the architecture of our development. With a well-structured and layered design, it offers valuable insights into best practices for database interaction. One notable aspect is the SchoolDAO's handleException method, showcasing an effective approach to exception handling. This method's implementation provides a clear and effective strategy for managing database-related exceptions, ensuring data integrity and transaction reliability.

Additionally, Leif's project introduces a structured view layer, highlighted by the CmdLine class. We have seamlessly integrated and adapted this class to handle user input in our application. The CmdLine class, designed to parse user commands and parameters, serves as a model for efficient and user-friendly interaction with the program. By leveraging this component, we enhance the user experience in our Sound Good Music School project.

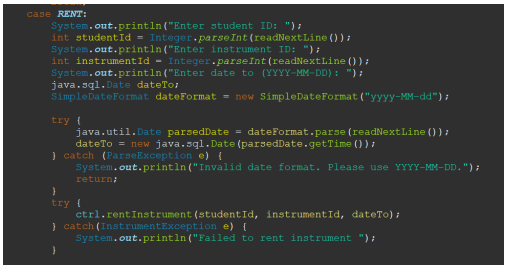
4 Results

The Git repository contains the project created to develop part of Soundgood Music School's website: `https://github.com/Seemamian/Data_Lagring.git`.

4.1 Layers

(a) Integration Layer: Rental Methods

(b) Controller Layer: Rental Method



(c) View Layer: Rental Method

(d) Utility Layer: Rental Rule Method

Figure 1: The different layers diplaying rental methods

The SchoolDAO class serves as a vital link between the SoundGood Music School's application and its PostgreSQL database, enabling seamless execution of essential tasks related to instrument rental management. The class encapsulates SQL queries and transactions related to instrument availability, rental creation, termination, and rule enforcement. Handling operations such as availability, creation, termination, and rule enforcement, the class ensures transaction integrity and robust error handling. By encapsulating SQL queries, it plays a key role in maintaining the application's reliability and functionality in instrument rental management.

Figure (a) above highlights the createRentedInstrument in the SchoolDAO class responsible for creating a new rental record in the database when a student rents an instrument. For the facilitation of renting an instrument, the user must provide: instrumentId representing the instrument being rented, studentId identifying the student

renting the instrument, and dateTo representing the date until which the student wishes to rent the instrument. Initially, the SQL statements are prepared with the corresponding parameters. The prepared statement (rentInstrumentStmt) is crucial for executing a secure and optimized SQL query to create a new rental record in the database. The use of executeUpdate() not only triggers the insertion of the new record but also allows the method to check the number of rows affected, acting as a validation step to confirm the success of the operation. Lastly, in the event of an SQL exception, the handleException method is called, which includes a rollback mechanism to maintain the database's consistency. The subsequent connection.commit() statement ensures that the transaction is permanently committed to the database, safeguarding against any inconsistencies.

The updateInstrumentStock method in the SchoolDAO class plays a vital role in ensuring the accuracy of instrument stock information in the database. Its primary function is to update the stock availability of an instrument, whether it has been rented or a rental has been terminated. To execute a secure SQL query, the method utilizes a prepared statement named updateInstrumentStmt.

```
updateInstrumentStmt = connection.prepareStatement("UPDATE " + STOCK_TABLE_NAME
+ " SET " + STOCK_AVAILABLITY_NAME + " = ? WHERE " + STOCK_ID_NAME + " = ? ");
```

Notably, the "FOR UPDATE" clause is employed to lock the selected rows when anticipating updates in the current transaction. This locking mechanism will help maintain data consistency by preventing other transactions from modifying the stock until the current transaction is committed.The available stock is retrieved through the use of anther method getInstrumentAvailableStock(). Thereby, setString() and setInt() calls on the prepared statement facilitate the incorporation of new stock information and the associated stock ID, respectively.The executeUpdate() triggers the update in the database and also validates the operation by checking the number of affected rows. In case of a discrepancy, where rowsAffected is not equal to 1, the handleException method is invoked to manage SQL exceptions.

Lastly, an additional method in the SchoolDAO that serves importance is the readInstrumentById method. The method in the SoundGood Music School's application lies in its ability to retrieve detailed information about a specific instrument based on the (instrumentId). Thereby it provides real-time details about an instrument, including its stock availability, name, brand, renting price, and availability stock status. By executing a SELECT query using the prepared statement selectInstrumentStmt, the method retrieves data, which is then processed to create an Instrument object, consolidating relevant information.

Additionally,the Controller class in the SoundGood Music School application encapsulates essential methods for managing instrument rentals. The getAvailableInstrument method retrieves a list of available instruments based on the provided instrument name. Moreover, the terminateRental method handles the termination of existing rentals, ensuring proper updates to rental statuses and instrument stocks. These methods ensure the interaction with the underlying database, promoting modularity and efficient management of instrument-related operations.

In Figure (b), the rentInstrument method within the controller layer of the SoundGood Music School application is depicting the rental process. The readInstrumentById method from the SchoolDAO class, plays a pivotal role. As before a rental is initialised, the readInstrumentById provides comprehensive information about the instrument being rented. This step ensures that the system possesses the most up-to-date details concerning the instrument's availability, name, brand, and other essential information.

Subsequently, during the rental process, the method also undertakes checks such as validating the instrument's availability. Figure (d) displays the method InstrumentAvailability in the Utility layer. The InstrumentAvailability class efficiently checks if an instrument is available for rental by verifying that its stock is greater than zero. If the stock is insufficient, it raises an InstrumentAvailabilityException.

To continue, the rentInstrument method also makes other checks including the student's eligibility, ensuring the rental period falls within permissible limits. Thereby, the application proceeds to create a rental record using the createRentedInstrument method. Simultaneously, the available stock of the rented instrument is decremented by invoking the updateInstrumentStock method in the SchoolDAO class. This centralized mechanism is integral in maintaining data consistency and accuracy, significantly enhancing the overall efficiency and reliability of the instrument rental system within the SoundGood Music School application.

Furthermore, the BlockingInterpreter class in the view of the SoundGood Music School application serves as a textual user interface (UI) for interacting with the underlying system. It is designed to handle and interpret user commands, providing a command-line interface through which users can interact with the application's functionalities. This class utilizes a Scanner to receive user input and communicates with the Controller to execute corresponding actions based on the user's commands. The handleCmds method processes user inputs in a loop, directing the flow based on the recognized commands.

Notably, the RENT case visualised by figure (c), within the switch statement handles the command to rent an instrument. It prompts the user for relevant information such as student ID, instrument ID, and rental date. It then utilizes the Controller to execute the instrument rental process. In case of exceptions, appropriate error messages are displayed to the user, ensuring a user-friendly interaction. Overall, the BlockingInterpreter class plays a pivotal role in facilitating user interactions and connecting the user's commands to the underlying application logic.

4.2 List Instruments

```
> LIST
Enter instrument type (e.g., Guitar):
> Guitar
No available instruments of type Guitar
> LIST
Enter instrument type (e.g., Guitar):
> Piano
Available instruments of type Piano:
Available Instrument {instrumentId=16, instrumentName='Piano', brand='Ludwig', rentingPrice=907, availableStock='14'}
>
```

Figure 2: Command Line: 'List Instrument'

Figure 2 is a snapshot of the terminal displaying interactions with a program designed to list and manage available instruments for rent. The user initiates the process with the "LIST" command, prompting the program to display available instruments. The user then inputs an instrument type, starting with "Guitar," but the program informs them that there are no available guitars. The user subsequently enters "Piano," leading to a positive response from the program. It lists details for available pianos, including instrument ID, instrument name ("Piano"), brand ("Ludwig"), renting price (907), and available stock (14).

To implement this functionality, the program utilizes the readAvailableInstrument method within the SchoolDAO class. This method executes a prepared SQL statement (listAvailableInstrumentStmt) that conducts a SELECT query, joining the instrument and stock tables and filtering by instrument name and stock availability greater than zero. The obtained data is then encapsulated in the Instrument class, allowing for convenient manipulation and presentation. The toString method within the Instrument class ensures a tidy display of instrument details during the iteration through the retrieved list of instruments.

Moreover, the interaction highlights the impact of rental operations on the available stock count. In the event of a successful rental, the program utilizes the decrementAvailableStock method in the Instrument class to reduce the available stock count for the rented instrument. Conversely, when a rental is terminated, the available stock count is increased through the incrementAvailableStock method, ensuring accurate tracking of available instruments.

4.3 Rent an Instrument

```
> RENT
Enter student ID:
> 4
Enter instrument ID:
> 24
Enter date to (YYYY-MM-DD):
> 2024-11-11
Instrument stock is zero - Not available!
Failed to rent instrument
> RENT
Enter student ID:
> 4
Enter instrument ID:
> 13
Enter date to (YYYY-MM-DD):
> 2025-11-11
Date is beyond the allowed range.
Failed to rent instrument
> RENT
Enter student ID:
> 4
Enter instrument ID:
> 15
Enter date to (YYYY-MM-DD):
> 2024-11-11
Rental is sucessful :)
> RENT
Enter student ID:
> 4
Enter instrument ID:
> 4
Enter date to (YYYY-MM-DD):
> 2024-11-11
Maximum number of rentals reached for this student.
Failed to rent instrument
```

Figure 3: Command Line 'Rent Instrument'

Figure 3 is a command sequence that depicts the many potential error scenarios seen when a student seeks to rent an instrument through the Soundgood Music School system. Each attempt exhibits a unique fault instance, offering a thorough understanding of the system's error handling capabilities. The commands emphasise cases such as attempting to rent an instrument with no available stock, picking a date outside of the allowed 12-month rental period, and exceeding a student's maximum rental limit. In addition, a successful rental transaction is displayed among these error scenarios, demonstrating the system's ability to handle successful activities alongside the encountered problems.

4.4 Terminate an Active Rental

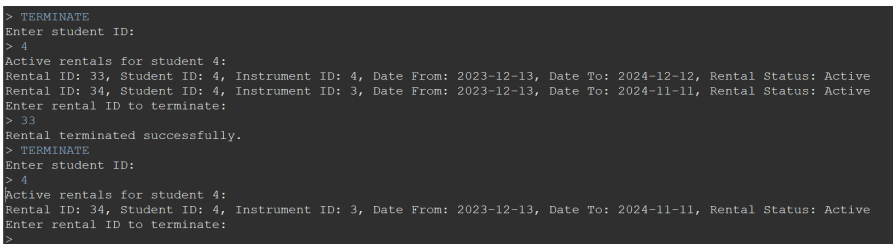


Figure 4: Command Line 'Terminate Rental'

The console window in the figure 4 shows the process of terminating a rental for student 4, rental ID 33. The user is first prompted to enter their student ID. Once the student ID is entered, the console displays a list of active rentals for that student. The user is then prompted to enter the rental ID of the rental they want to terminate. Once the rental ID is entered, the console terminates the rental and displays a success message.

The termination process is streamlined through the terminateRental function within the Controller class. This function updates the rental status to indicate termination and sets the end date to the current date in the database. Additionally, it utilizes the Instrument class's incrementAvailableStock function to decrease the available stock count for the terminated instrument.

This streamlined feature is accomplished by systematically interacting with the database tables and making precise modifications to the rental and instrument records. When a rental is discontinued, the program ensures accurate tracking of both current and terminated rentals by updating the rental status and end date. The reduction in available stock guarantees that the terminated instrument becomes available for rental once again. This method ensures precise control over rental statuses and available instruments, maintaining data accuracy and providing a consistent user experience.

4.5 Command Line of Instrument Stock Management for a given user

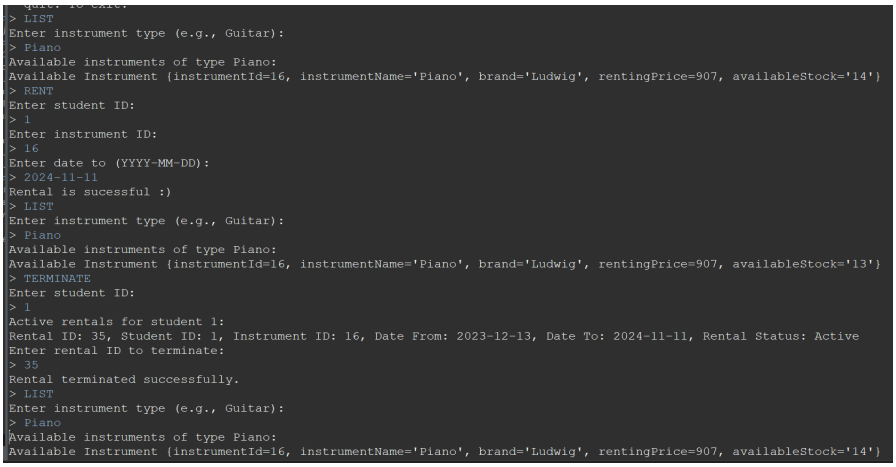


Figure 5: Commands: List, Rent, Terminate

Figure 5 displays a user renting a piano from a rental program. The program allows users to search for available instruments, and then rent them if desired. Thus, the user first enters the command LIST to see a list of available instruments. The program thereby, lists the available instrument of that type which are available. The student is then able to rent the instrument by specifying their student ID and the instrument ID of the instrument which the previously relieved when listing the piano. The student additionally specifies a rental period with in the next 12 month in order to successfully rent the instrument. In the figure, a termination of a rental can also be displayed by in which the student first specifies their studentID. The console thereby shows all the active rentals for such student. Thereby the student can give the rentalID of the specific rental they wish to terminate in-order to successfully terminate the rental.

5 Discussion

The program created aims to assist users in listing available instruments by name, renting desired instruments using their student ID and instrument ID, and terminating ongoing rentals according to a specific method. Consequently, the executed code effectively generates an operational program, allowing students of Soundgood Music School to seamlessly execute fundamental operations on their website, including listing available instruments, renting instruments, and terminating rentals.

In order to address the termination requirement, the Soundgood Music School database created in the previous in seminar 2 is updates. Thereby, a status column is introduced in the ”renting_period” table within the database. This status attribute is intially set to ”Active”, signifying ongoing rentals. Hence, in case a termination is necessary, the status is updated to ”Terminated.” This approach ensures the database’s retention of all

rental data. However, a potential drawback might arise from accumulating a substantial volume of information over time, leading to an enlarged table size that could impact long-term performance and data management. Nonetheless, the implemented code seamlessly incorporates these functionalities, precisely adhering to the criteria outlined in the results section and meeting the specified task requirements.

In adherence to the code requirements stipulated in this task, a deliberate decision was made to disable the auto-commit feature upon the establishment of the database connection. This intentional measure signifies that transactions are not automatically committed following each SQL operation. Rather, manual handling of commits has been implemented. This underscores our strategic approach to executing SQL statements within a transactional context, which constitutes a focal point in meeting the specified code requirements.

The process initiates with a purposeful strategy for managing database transactions. Upon establishing the database connection, a proactive step is taken by explicitly disabling the auto-commit feature through the connection.setAutoCommit(false) method within the connectToSchoolDB method. This singular action establishes a foundation for consistent transactional behavior across all subsequent methods in our SchoolDAO class.

Every method within the class functions within a transactional context, employing a shared global connection variable and utilizing prepared statements. This encapsulation ensures that all SQL operations carried out within these methods are inherently part of a transaction. This setup provides explicit control over the commit and rollback processes, ensuring precise management of the database transactions. Consequently, each method within the class incorporates a comprehensive try-catch block to encapsulate the SQL operation, facilitating verification of its success or failure.

In instances where our SQL operations are executed without any issues, the try block includes a deliberate commitment using the connection.commit() method. This crucial step finalizes the modifications made during the transaction, rendering them permanent and integral to the database. Conversely, when faced with a failure or exception, our robust exception-handling system comes into play. Within the catch block, the handleException method takes charge of orchestrating a rollback. This essential measure reverts any changes that were not committed, ensuring the preservation of the database's consistency and integrity.

The handleException method performs a dual role, overseeing both the rollback process and exception handling. It enhances error messages in the event of complications and ensures the appropriate addressal or propagation of the original exception, thereby guaranteeing reliable and consistent error reporting. The methodology adopted centers on the manual management of transactions, aligning with the ACID (Atomicity, Consistency, Isolation, Durability) properties. This meticulous approach ensures the dependable execution of database operations and preserves data integrity and coherence within the system.

Additionally, the strategic incorporation of 'SELECT FOR UPDATE' in specific

database transactions within the application plays a crucial role in ensuring data consistency and integrity in the system. This construct is applied in key methods, such as renting an instrument or terminating a rental. The 'SELECT FOR UPDATE' feature serves to lock relevant rows in the database, preventing concurrent access or modification by multiple users during the transaction. For instance, in the 'readInstrumentById' method, 'SELECT FOR UPDATE' ensures the secure locking of the instrument's availability, facilitating subsequent operations like updating availability and committing the transaction without interference from other transactions.

Thus, data access and modifications during these transactional operations are effectively controlled through the application of this approach, thereby mitigating potential conflicts and upholding the consistency and reliability of database operations.

Furthermore, a careful structuring of six layers characterizes the architecture: integration, view, startup, controller, model, and utility. Deliberately, business logic is not embedded in the integration layer; rather, complex logic is directed to the utility layer. Specific logical methods, such as validating rental end dates within a 12-month range, checking instrument availability, and ensuring students have not exceeded their instrument rental limits, are housed in different classes within the utility layer. This approach ensures the separation of concerns. Notably, the SchoolDAO does not incorporate any business logic in the integration layer, controller, or view classes.

The SchoolDAO exclusively oversees CRUD methods (Create, Read, Update, Delete) for database operations, while the controller delegates responsibilities to distinct utility classes. This delegation is exemplified in the 'rent' method, where diverse responsibilities are assigned to various utility classes crucial for overseeing the rental process. This approach orchestrates the rental workflow, engaging with utility classes for specific tasks.

Initially, the procedure involves the retrieval of instrument details and the validation of pivotal parameters, including the instrument's stock availability. Subsequently, it utilizes the StudentRental utility class to ascertain whether the student has surpassed their maximum allowable rentals. Following this verification, the rental period undergoes scrutiny against predefined constraints to ensure its compliance within an acceptable range. Upon successful validation, the method employs integration layer methods (soundGoodDB) to execute the rental transaction in the database. This process encompasses the reduction of available stock for the rented instrument and the corresponding updating of rental records.

Integral to this method is the incorporation of exception handling, which involves capturing specific exceptions associated with database operations, rental availability, instrument availability, or date constraints. In the event of an exception, it guarantees the proper management of the ongoing transaction by invoking the 'commitOngoingTransaction' method. Fundamentally, the 'rent' method operates as a central orchestrator, utilizing utility classes for distinct checks and validations. This approach promotes modularity, ease of maintenance, and a clear separation of concerns within the controller layer, thereby contributing to a more robust and organized code base. Lastly,

the view layer can be exemplified by the BlockingInterpreter class. The class exclusively interacts with the controller, establishing a distinct separation between user interactions and business logic. This architectural methodology ensures clarity, ease of maintenance, and a thorough adherence to established principles.