ARA Software Design Specification

Brad Bailey, Elizabeth Bailey, Brandon Dodd, Jesse Lunger, Nathan Malamud

Table of Contents

1. SDS Revision History

Date	Author Initials	Description
4.3.22	BB	Created draft/outline
4.5.22	BD	Start “System Overview” and fill in outline a little bit
4.5.22	JL	Gantt Chart/TOC/sm edits
4.5.22	EB	First draft of GUI Manager module
4.5.22	BB	Database module
4.6.22	JL	File Manager module Added references
4.6.22	NM	GUI Documentation, text citations, styling and formatting, added figures
4.6.22	EB	Edits to GUI manager module, reformatting of module sections, add figures
4.6.22	BD	Small edits to all sections in preparation for submission
4.9.22	NM	Updated GUI Manager Documentation
4.24.22	NM	Revised SDS draft for final submission

2. System Overview

The Active-Reading Assistant (ARA) is an application that assists the user in learning and practicing the SQ3R (Survey, Question, Read, Recite, Review) reading comprehension technique (Johnson, 2013). The program is organized into a few major components:

1. **Multiple GUI Views** - display text and graphics, prompt user for input. Send and receive notes from the GUI Manager.
2. **GUI Manager** - creates and manages the GUI views. Passes information to and from the Document Manager.
3. **Document Manager -** takes in user input from the GUI manager and passes it along to the MySQL database. Also retrieves data from the database when needed.
4. **MySQL Document Database -** stores notes made by the user on a remote server.

3. Software architecture

Multiple GUI Views: The different visual interfaces for the user. These will prompt the user for text input (chapter headings, questions, and notes) and display previously recorded notes using the SQ3R methodology.

GUI Manager: The underlying code for the GUI that will receive user input from one of the GUI views and prepare it for storage on the MySQL Document Database. It also receives input sent from the MySQL Document Database so that it can be displayed on one of the GUI Views.

Document Manager: Serves as an ambassador between the GUI Manager and the database. Handles queries and thereby modularizes logic for data writing and retrieval.

MySQL Document Database: Stores all input from the user into an online SQL database. All of the data that is stored here is organized by book and chapter titles.

Our rationale with this architecture was to provide a separate component for each major function of the program while not adding unneeded complexity to the overall design. By dividing the program in this way, it becomes easier to organize the code and work with it through the development and maintenance phases of the project.

Our high-level architecture is illustrated below in **Figure 1**. Bold arrows indicate transfer of information and dashed arrows indicate module dependencies.

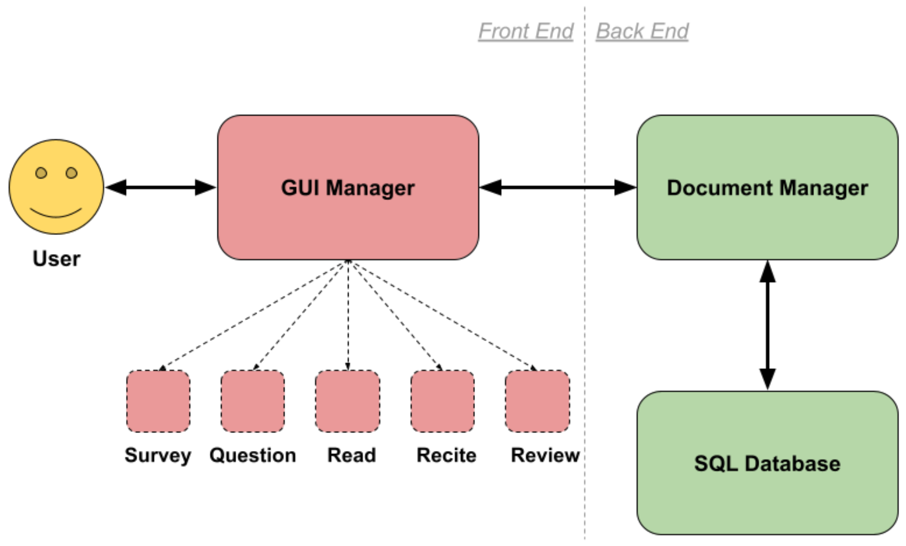

Figure 1. ARA Software architecture.

To understand our application design, it is essential to note that each view generated by the GUI is handled as a separate module. For the front end, it made sense to have a separate component for each view of the UI for the following reasons:

- **Modularization of each frame as a functional entity**. This made testing and debugging a lot more manageable for a group of 5 software developers.
- **Reduction of the amount of labor needed** to be spent on designing widgets for each UI frame.

The GUI manager was created to handle each of these separate views. In addition to the five views representing the stages of the SQ3R learning process, we also had a start page for when the user first runs the application.

4. Software Modules

4.1. GUI Views

The user will be guided through the SQ3R learning process through a series of pages in a single application window. Every step of the process will be handled in a separate page of the application, and the user will be guided through each step sequentially. Every page will be generated by a different python script using the Python Tkinter library. The UI workflow is illustrated below in **Figure 2**.

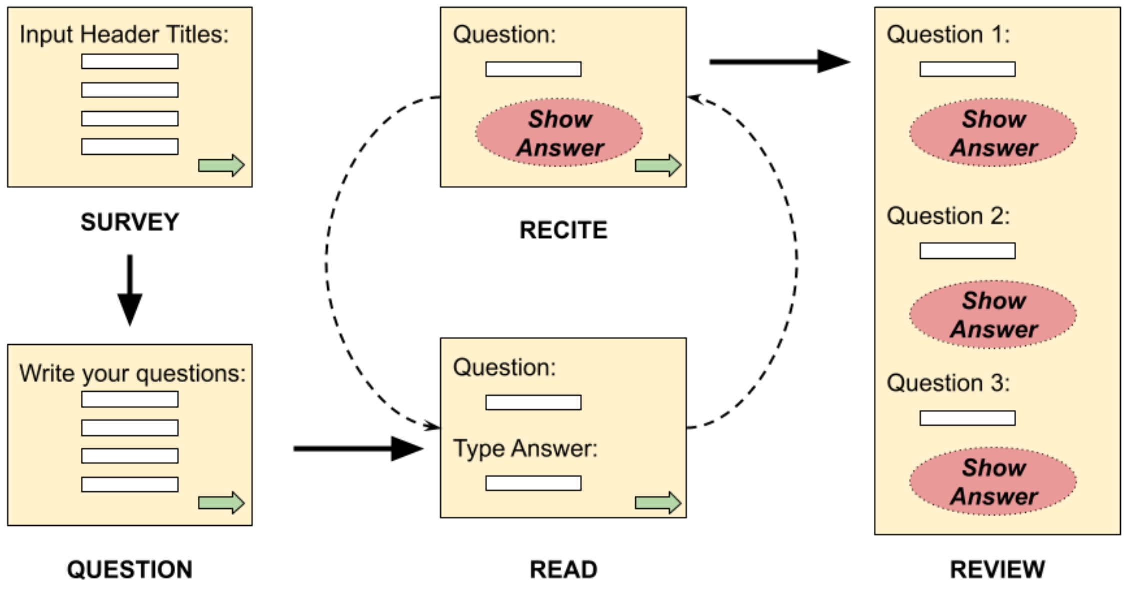

Figure 2. Ordering of GUI views for the ARA application. Arrows show the sequence of pages that the user is guided through. Note that the *recite* and *read* pages are illustrated as a loop.

The *survey* page prompts the user to input chapter section titles via a series of text boxes.

The *question* page prompts the user to rephrase every header title as a question that they will answer through the next stage. They can also add any additional questions that they may have for each header.

The *read* page prompts the user to answer every question they entered in the *question* view.

The *recite* page hides the answer to a question behind a button, and then reveals the answer to the user when prompted. It is expected that the user will recite the answer from memory before showing the answer.

Finally, the *review* page lists all questions in a drop down menu. The user can select which chapter header, and which question to recite the answer to. Like in the *recite* page, the answer is hidden from the user until they press the “show answer” button.
As with every stage of the process, the user can continue to the next view by clicking the “next step” button in the bottom right corner of the view (illustrated as a green arrow).

In our GUI workflow, we decided to illustrate the *read* and *recite* views as a loop because we believe that they should be co-processes. By writing out the answer to the question in the *read* view and having to *recite* it immediately in the next view, the student user will be practicing active learning (Braxton *et al.*, 2008). Following the *read* and *recite* stages, the *review* window will present all questions in a list format so that the student can view them all on one page.

The GUI windows are directly responsible for prompting user input. It does this through various interactive elements: drop-down menus, text boxes, and buttons. The GUI manager will be responsible for changing the views and relaying changes to the document manager following any interaction between the user and the GUI.

There were a few options we considered for the layout and functionality of the user interface. Ultimately, we decided this format was best because it doesn’t overload the user with information which makes it even easier to learn the SQ3R method one step at a time. Since the user can disable the extra information, it won’t be cumbersome for them. Also, by sending information to the GUI Manager as it gets updated, information in the database is always up to date.

4.2. GUI Manager

The GUI manager is responsible for updating the GUI following user interaction, tracking the user’s stage in SQ3R, and making calls to and from the document manager. It will import all Python scripts for generating Tkinter windows and will run each respective script when needed (e.g., first run *survey.py*, then run *question.py*, and so on).

- **Moving through GUI Views**: When the user clicks the “next step” button in their respective GUI view, a message will be sent to the GUI manager to close the current view and move to the next view. The GUI Manager will keep track of the current SQ3R stage and move through windows appropriately based on user interaction and existing data from the Document Manager. For example, the *read* and *recite* views will occur in a loop based on the number of headings initially entered by the user, moving on to the *recite* window only when there are no remaining “heading” data entries.

- ***Sending Input to the Document Manager***: When a user inputs text-based information (header names, questions, answers), the GUI manager will send this information to the Document Manager so that the Document Manager can communicate those changes to the database.

- ***Requesting Data from the Document Manager***: When the GUI view requires stored information from the database (header names, questions, answers), the GUI manager will request that information.
  The last two bullet points will require the implementation of “getter” and “setter” methods, respectively, so that the GUI manager will be able to precisely specify what changes need to be made to the database or what info needs to be retrieved.

This module will gather user input that is to be sent to the database using setter methods defined by the document manager. Additionally, this module will need to make calls to the database using getter methods defined by the document manager to retrieve stored user input, displaying this text to the user.

This module is designed to interface between the GUI windows and the document manager. Each window is sequential and is displayed based on the current stage of SQ3R and the remaining database entries. This module keeps track of the stage, handles making the appropriate calls to the Document manager, and interprets the data sent from the Document manager. This design separates the functionality of the window presentation from the logic behind the sequential flow and calls to the Document manager. Breaking it down in this fashion allows a clearer distribution of work and makes planning milestones for the GUI easier to break down.

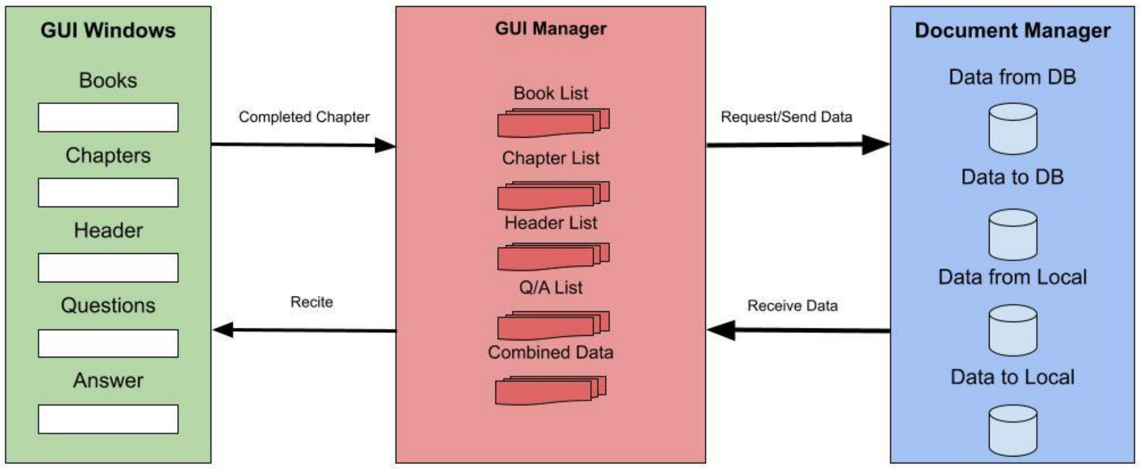

Figure 3. Illustration of the interactions between the GUI Manager and adjacent modules in the software architecture.

4.3. Document Manager

The goal of the Document Manager is to package information in such a way that it will make the application easier to work with and more versatile for other databases. If we get ahead in our project we will give the Document Manager the ability to save to the current directory and to restore its directories from the MySQL database, thus allowing data to be accessible offline.

The Document Manager interfaces with other modules by communicating with the GUI Manager and the MySQL DB to retrieve and send data using getter and setter methods defined by this module. We may attempt to work with Pymongo.

Below in **Figure 4** is a static model of the document manager, while **Figure 5** illustrates a dynamic model.

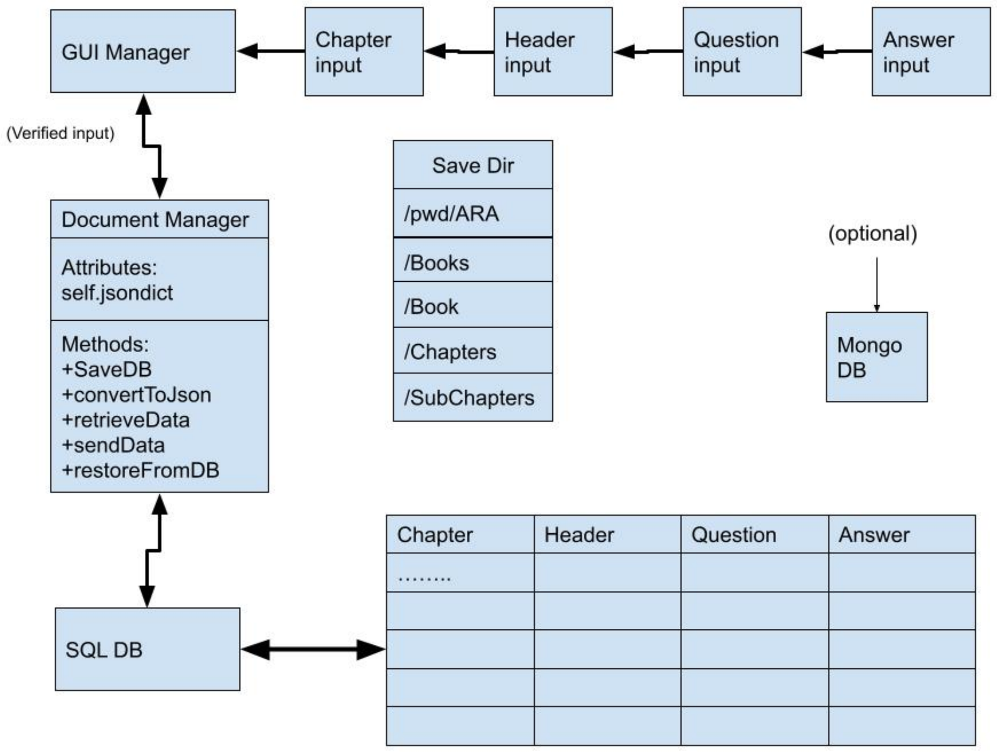

Figure 4. Document manager static model.

This is to create a more stable application that is both easier to work on and upgrade in terms of readability and data structures. It will also give extra redundancy in case of database failure or operation offline.

4.4. MySQL Document Database

Overview: The SQL database serves as the primary data structure for storing and retrieving user input.

The module interfaces with the GUI/local application via Python’s MySQL library, which allows a developer to construct SQL statements and queries directly in python, and then execute them using a connection to the SQL server. See Chapter 1 Introduction to MySQL Connector/Python for more information.

History of Alternatives: In the course of development 2 main static (i.e. structural) models were considered: 1.) a single-table schema, and 2.) a multi-table schema.

The two alternatives have their pros and cons: a single table is more robust in that it is less susceptible to data entry errors, but incurs a higher overhead cost as each row must contain all values. Multiple tables are more efficient, but need careful treatment to avoid errors from foreign key checks on insert, and a developer must carefully design queries to join the tables correctly.

The multi-table schema design considered is best documented by the following diagram:

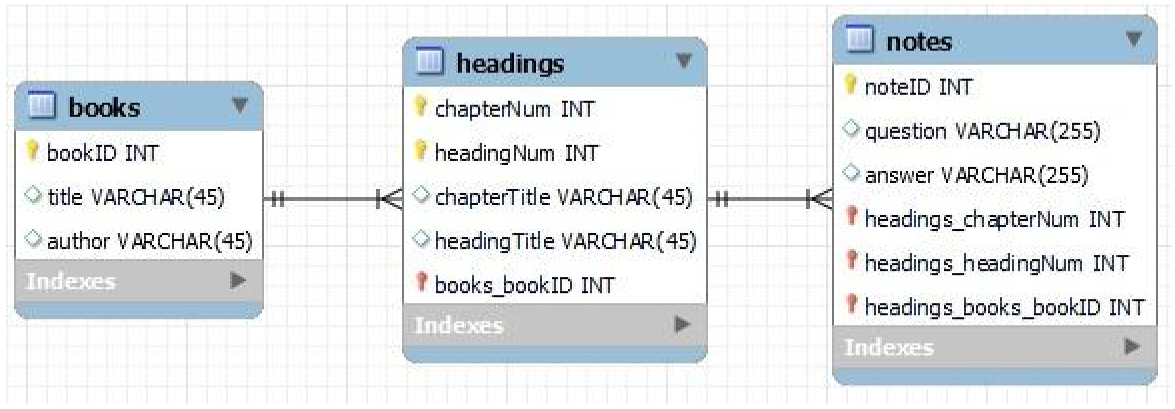

Figure 5: a multi-table schema.

In this design, each table tracks information regarding its title, and the tables are linked by the keys: bookID, chapterNum, headingNum, and noteID. This would allow for each question-answer pair to be uniquely identified by its bookID-chapterNum-headingNum-noteID tuple. While this design is theoretically ideal for the system requirements, in practice it was cumbersome to use and test with the GUI. As such, the authors favored a single table design for its ease of use and robustness to error.

Implementation: In the final version, we use a schema consisting of a single table, 'notes', pictured below in Figure 6.

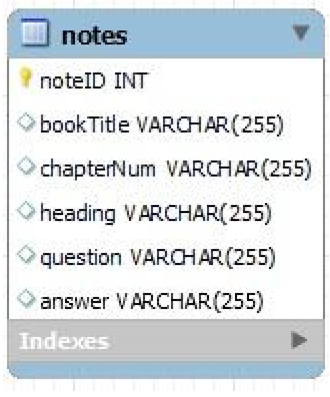

Figure 6: single-table database schema

This schema consists of a single table with six fields: noteID, bookTitle, chapterNum, headingTitle, question, and answer. The types of these fields are: int, varchar(255), varchar(255), varchar(255), varchar(255), and varchar(255), respectively. The noteID column is set to auto-increment and serves as a unique identifier for each entry in the table. The type VARCHAR(255) was chosen as it is the largest value for VARCHAR allowable by mysql. While the authors doubt a user would need this much space to store information in their use of the ARA application, the volume of information stored in the database is low enough that performance and memory concerns from allocating the max value for each VARCHAR field are negligible.

Rationale: As data security and consistency are not life-or-death concerns in the ARA application, a single table was the most straightforward way to fulfill the software requirements. The single table gives the user flexibility regarding data entry in that multiple books, chapters, and headings of the same name may be entered into the database. As each entry is tracked by a unique identifier, conflicts between these multiple entries are avoided. Additionally, the single table is the easiest design to implement and frees the developer from cumbersome error checking and flow control that would distract from the core functionality of the software.

Dynamic model: Regardless of design choice, the dynamic model of the SQL database is similar. Users begin by creating the tables in the database manually, with the actual SQL statements abstracted by the GUI. The user then proceeds through the SQ3R method, creating additional entries as need be. Data entered by a user is saved upon completion of the SQ3R method. Upon the return of the user, the program retrieves this data from the server with an SQL query, allowing the user to resume where they left off, or quiz themselves on previous material. Books exist in perpetuity on the SQL server. This sequence of operations is illustrated in the diagram below in **Figure 7**:

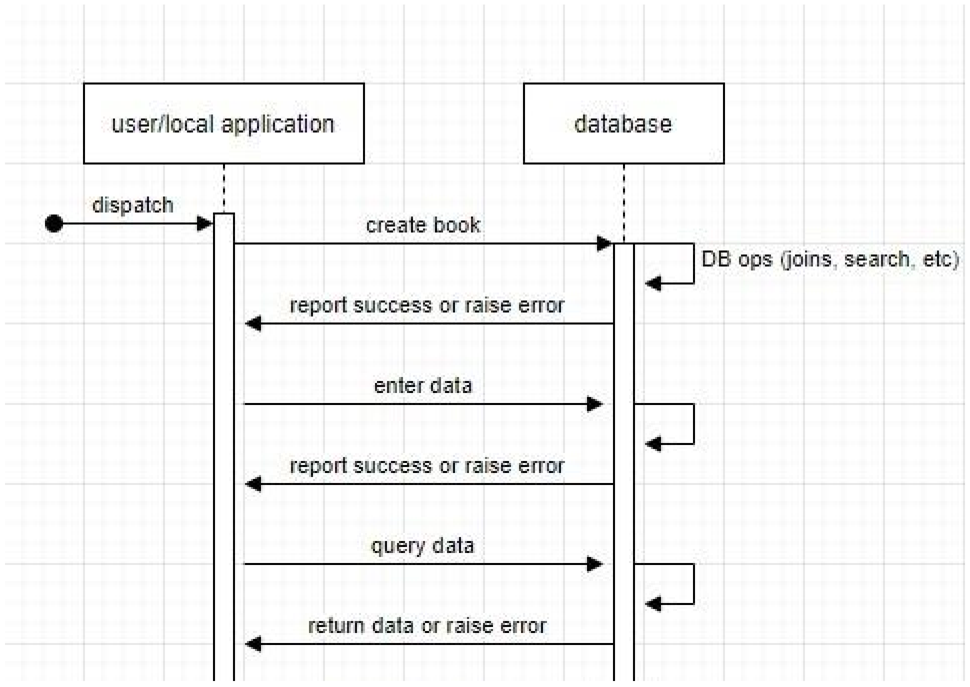
user/local application
database
dispatch
create book
DB ops (joins, search, etc)
report success or raise error
enter data
report success or raise error
query data
return data or raise error

Figure 7: dynamic UML diagram illustrating interactions between local application and SQL database.

Of course, this diagram does not reflect the full range of possibilities when it comes to user interaction. In reality, users can perform the steps in a number of different orderings. The only strict requirement is that at least one book exists before a user enters data into the database. If we are going to consider what module interactions need to occur for each specific use case, we will need to design some *dynamic models of operational scenarios* (see section 5, which is coming up next).

---

5. Dynamic Models of Operational Scenarios

Student Use Case: Use case for students proceeds in several steps. First, students are prompted to enter a book and a chapter that they wish to review using the SQ3R method. Next, they will input all of the chapter headings in the *survey* page. These entries for questions are stored locally as well as on the SQL database. Once the student has finished answering all of the chapter headings, they can input questions that they have on the *questions* page. Answers to these questions will be made by reading through each heading they wrote out.

Once the entire process is complete the book entry will be able to be accessed in the drop down menu on the opening page of the app. If a student wishes to review the material the book can be selected to review all of the notes they inputted. Below, in **Figure 7** and **Figure 8**, dynamic UML diagrams are used to illustrate the interactions between all of the program modules (user and database included).

Figure 7 is a much more simplified dynamic model while **Figure 8** is a rough sketch of all the details.

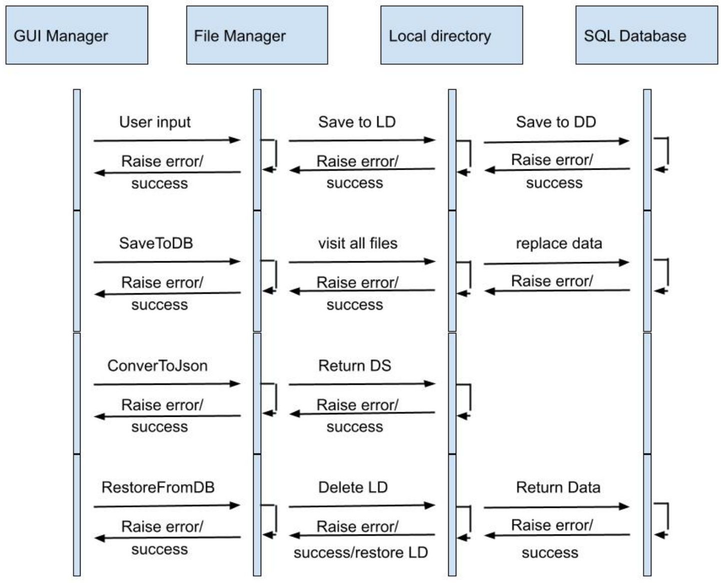

Figure 8. Document manager dynamic model.

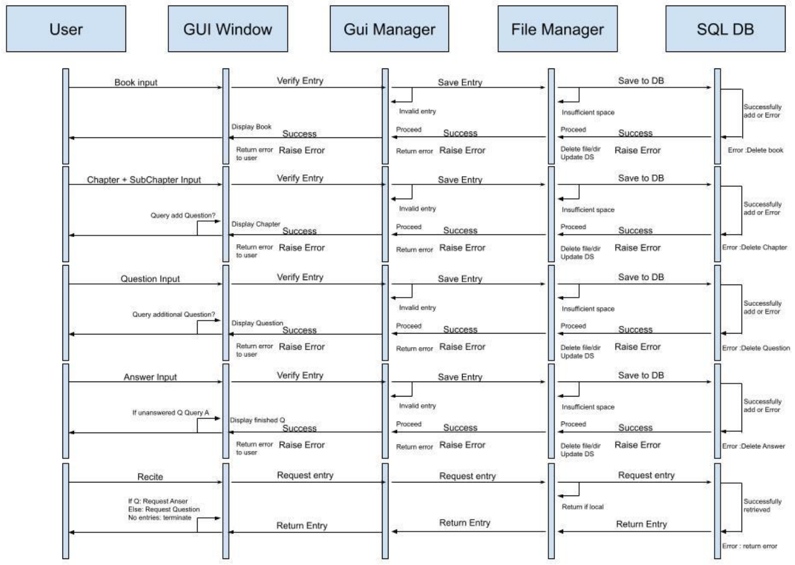

Figure 9. Dynamic model of operational scenarios.

In **Figure 8**, we see that regardless of the form of user input (chapter headings, questions, answers, or a request for data from prior note-taking sessions), the GUI manager is responsible for relaying the data to the File Manager (Document Manager) and from there the File Manager is responsible for saving the files to the local directory. The data is then sent from the local directory to the database, or it fetches the data from the SQL back-end and relays it in reverse order (right-to-left in the diagram) to the GUI front-end. For more detailed sequences of operations for every possible class of user input, **Figure 9** considers the exact operations needed for: book input, chapter and sub-chapter input, question input, answer input, and finally reciting the answers to prior questions.

Admin Use Case:
The use case for an administrator is relatively straightforward. They would first request instructions from the program on setting up a MySQL server instance. Then, using a shell client of their choosing, they would create the SQL server as per the instructions provided. The diagram below (**Figure 10**) illustrates the interactions that need to occur for the SQL server to get up and running.

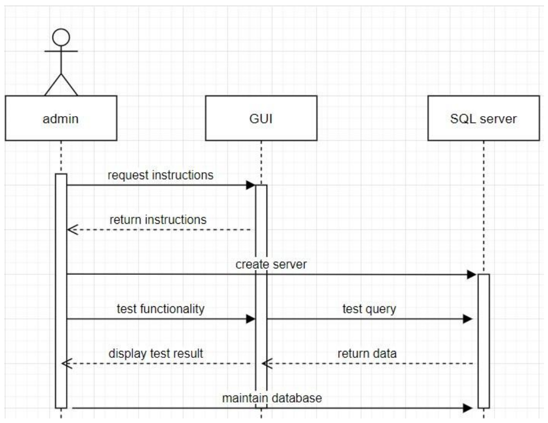

Figure 10. Illustration of admin use case.

The shell client is not pictured in the diagram as it is completely external to the development of our software. After creating the server, the GUI will provide functionality to automatically run a number of tests to ensure correct functionality, and provide documentation for both general database management and for error troubleshooting.

For more details on the admin use case, view the **instructions** attached in the .zip file submission.

6. References

Braxton, J. M., Jones, W. A., Hirschy, A. S., & Hartley III, H. V. (2008). The role of active learning in college student persistence. New Directions for Teaching and Learning, 2008(115), 71-83.

Jonson, J. (2013) 'SQ3R Reading Method'. Available at: https://www.youtube.com/watch?v=0dhcSP_Myjg (Accessed: 10 November 2020).

Shirgoldbird. (2022). *Microsoft SQL Documentation - SQL Server*. SQL Server | Microsoft Docs. Retrieved April 6, 2022, from https://docs.microsoft.com/en-us/sql/?view=sql-server-ver15

Wilson, Chris. *Instructions for creating a SQL server on IX.* Retrieved April 4th, 2022, from: https://classes.cs.uoregon.edu/21F/cis451/links.html

- -

Python GUI Programming. Python gui programming. (n.d.). Retrieved April 6, 2022, from https://www.w3schools.in/python/gui-programming

Python 3.10.4 documentation. 3.10.4 Documentation. (n.d.). Retrieved April 6, 2022, from https://docs.python.org/3/

MySQL Connector/Python Developer Guide. Retrieved April 4th, 2022, from: https://dev.mysql.com/doc/connector-python/en/

7. Acknowledgements

The design of this SDS was based heavily on the outline given to us by Professor Hornof for CIS 422. All included diagrams were also inspired by his UML Diagram Notation.

We would like to thank Chris Wilson for his instruction in CIS 551 (Database Management) on creating and maintaining a SQL server on the UO's IX machine, and for his instruction regarding creating database schemas and writing SQL queries.