Felipe Mesa Paredes
ID: 2466265
Ryan Morov
ID: 2492176

StudySnap – Flashcard Study App

Programming III Final Project

StudySnap Description

StudySnap is a self assessment flashcard study tool designed to help users/students to organize their learning materials into subject-specific groups, called Decks. The core goal of this application is to mimic a real-world quiz environment where it's users can test their recall abilities using digital flashcards.

This application generally revolves around three main interfaces: The dashboard used for managing decks, a deck editor to create content and a study mode for active learning. Users can create decks and populate them with flashcards, each having a frontside where a question is shown, and a backside, where the answer/definition will be revealed.

During a study Session, the application will first randomize the order of the cards to ensure effective testing if the deck is reused again. The user views the question, attempts to recall the answer, then reveals the back of the card to verify if they were right or wrong. Users then self-report their performance by marking each card as either correct or incorrect. The system tracks these statistics actively to calculate a final score percentage for the session and then proceeds to maintain the historical data such as the last studied date, the best score and the deck's average score.

Development approach

1- Understanding the problem

The application will represent a study tool based on the concept of flashcards. It functions as a self-assessment system where users organize study material into specific subjects. The core mechanism involves presenting a question side of a card, allowing the user to recall the answer, and then revealing the answer side for verification. The system tracks the user's self-reported performance (correct or incorrect) during a session to provide a final score percentage, mimicking a real-world quiz environment.

2- Formulating the problem

The study data will be represented as a collection of Decks, where each Deck contains a list of Flashcard objects holding two string values (Front and Back). User input will be captured to create these lists and modify them. During a study session, the order of cards within a Deck will be randomized using a shuffling algorithm. The application will maintain temporary counters for correct and incorrect answers during the session. Upon completion,

the session statistics (score, date) will be calculated and stored persistently alongside the deck data to allow for historical tracking of performance.

3- Developing the application/algorithm

1. **Main Dashboard (MainWindow)**
    - **Purpose:** The central hub where users see all their decks.
    - **Elements:** List of decks, buttons to "Create Deck", "Delete Deck", "Start Study", and "Exit".
2. **Deck Editor (DeckManagerWindow)**
    - **Purpose:** To add or edit cards within a selected deck.
    - **Elements:** Input fields for Front/Back text, a list of current cards, and "Save" button.
3. **Study/Quiz Mode (StudyWindow)**
    - **Purpose:** The active learning interface.
    - **Elements:** Large display for the card question, a "Reveal Answer" button, and "Correct/Incorrect" buttons to track progress.

4. Implementing the application/algorithm

This phase involved translating the conceptual design and the UI layouts into one functional application using C# and the WPF framework. The logic defined in the formulation phase was coded into specific classes. For example, the deck class was implemented to manage collections of flashcards, while the flashcard class was built with properties to enforce data integrity. The UI elements defined in the previous step were connected to this backend logic through event handlers in the code-behind files (.xaml.cs files) to be able to ensure that actions such as saving or revealing a card answer triggered the appropriate data processing and state updates.

5. Testing

The final step of algorithmic thinking revolved on verifying the application's correctness by testing it with various inputs and edge cases that might attempt to break it. There is two Json Files that were made with sample data for testing as well to really push the testing to it's limits in an organized way. Testing also ensured that the "Correct/Incorrect" counters accurately calculated the final score percentage and that the input validation logic correctly handled errors, such as preventing the user from saving a card with missing text.

OOP Design

DataRepository

The DataRepository class serves as the application's persistence layer, responsible for all file input/output operations. It utilizes the System.Text.Json library to serialize objects into JSON format for local storage and deserialize them back into usable objects at runtime. The class provides specific methods such as LoadDecks and SaveDecks to manage the user's library of card collections, ensuring that data is preserved between application restarts. Additionally, it handles the storage of historical performance data through LoadSessionResults and SaveSessionResults, effectively isolating data access logic from the rest of the application.

Deck

This class represents a specific subject or category of study material, acting as a container for related content. It maintains a Name property and a list of Flashcard objects (Cards), enforcing valid state by preventing null lists or empty names. The class encapsulates collection management logic through methods like AddCard and RemoveCard, allowing the application to modify contents dynamically. It also includes helper properties such as LastStudiedDisplay and BestScoreDisplay to facilitate the binding of summary statistics directly to the user interface.

Flashcard

The Flashcard class models the fundamental unit of data in the system, simulating a physical study card. It consists of two primary properties: Front (the question or prompt) and Back (the answer or definition). To ensure data integrity, the class implements validation logic within its property setters, throwing an Argument Exception if a user attempts to save a card with empty or whitespace-only text. This ensures that every card created contains valid study material.

StudySession

This class manages the logic and state of an active quiz, separating the "studying" process from the static deck data. Upon initialization, it accepts a Deck object and uses a ShuffleDeck method to randomize the order of cards, preventing users from memorizing patterns. It tracks real-time progress using internal counters for the current card index and correct answers. Key methods include GetNextCard to advance the quiz and RecordAnswer, which updates the user's score based on their self-reported performance.

StudySessionResult

Designed to capture the outcome of a completed review, the StudySessionResult class acts as a historical record. It stores essential metadata about a finished session, including the DeckName, the Date of completion, the final Score percentage, and the raw counts of correct versus total questions. These objects are intended to be serialized by the DataRepository, allowing the application to display a history of the user's learning progress over time without needing to recalculate past results.

UML Design

See Appendix B for UML design.

Contributions

The development of the StudySnap application was divided into two primary areas of focus to leverage individual strengths. **Ryan Morov** (Member A) spearheaded the backend architecture and business logic. His contributions included designing the core data models (such as the Deck and Flashcard classes), implementing the persistence logic to ensure data is correctly saved to and loaded from files, and coding the algorithms responsible for shuffling decks and calculating study session scores. **Felipe Mesa Paredes** (Member B) focused on the frontend experience and project documentation. He designed the WPF user interfaces and navigation structure, managed UI event handling (such as card flipping mechanics and user input), and compiled the final report and user guide to ensure the project met all visual and formatting requirements.

App Setup

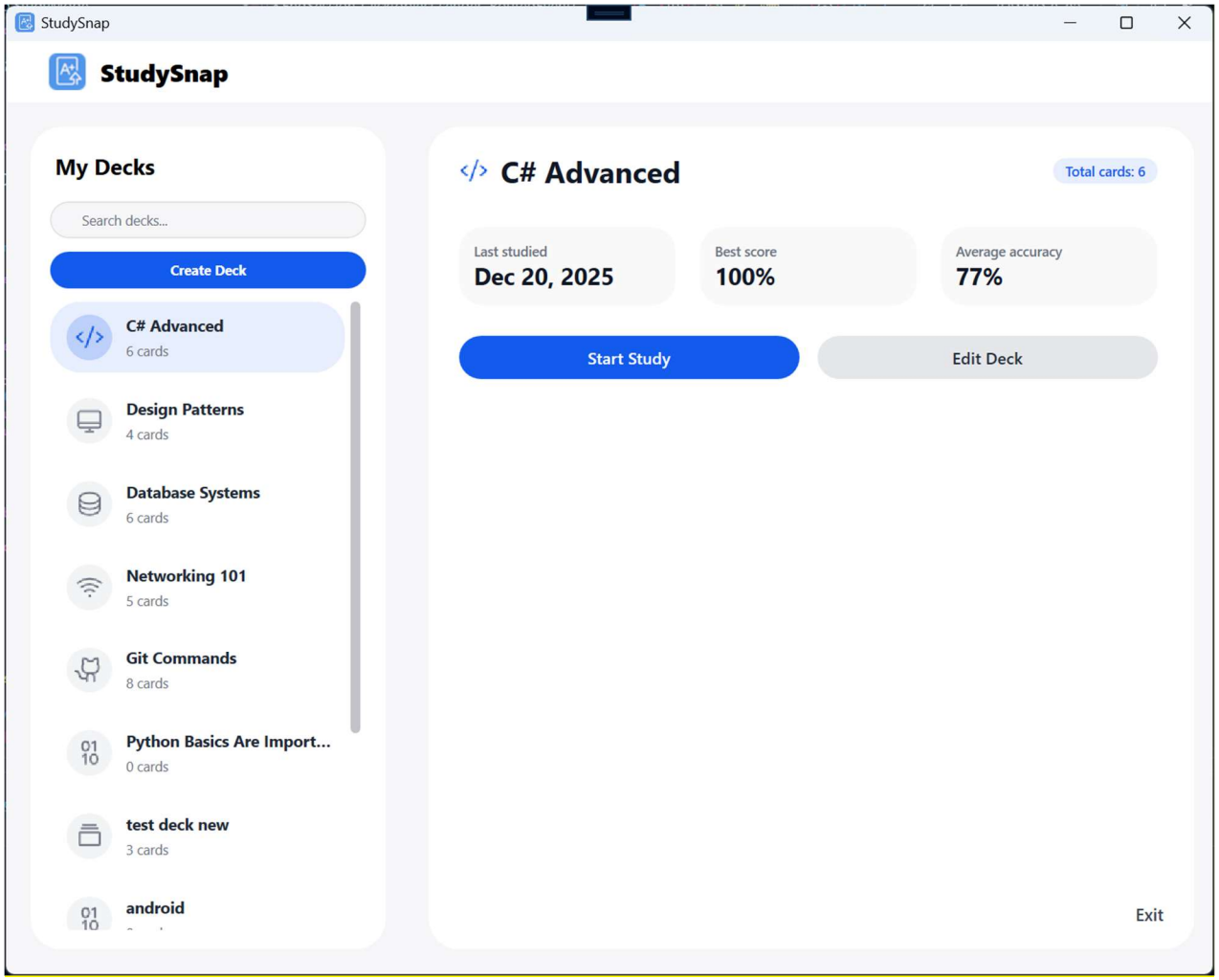
The **StudySnap** repository follows a clean, modular architecture typical of modern WPF applications targeting .NET 9.0. This codebase is organized to separate concerns effectively: the UI logic is contained within XAML files, while the business logic and data structures are isolated in the Models folder. Additionally, the project utilizes a dedicated Converters directory for UI helpers, such as the IconToGrayConverter, which dynamically alters image paths to manage visual states without cluttering the view logic.

A distinctive feature of this application's setup is its lightweight data persistence mechanism by implementing a custom DataRepository class that manages data storage using portable JSON files. By leveraging the System.Text.Json library, the app serializes complex objects, such as Decks and Study Session Results, directly into human-readable text files. This approach significantly simplifies deployment, testing and backup, as the application requires no external database installation to function, making it entirely self-contained.

To enhance the user experience beyond standard WPF controls, the project integrates external libraries via the NuGet package manager. Specifically, the application includes the **XamlRadialProgressBar** (version: 1.0.3) to provide rich visual feedback during study sessions. This demonstrates the team's ability to extend the core .NET framework with community-driven tools to achieve specific design requirements that would be difficult or time-consuming to implement from scratch.

App Snapshots

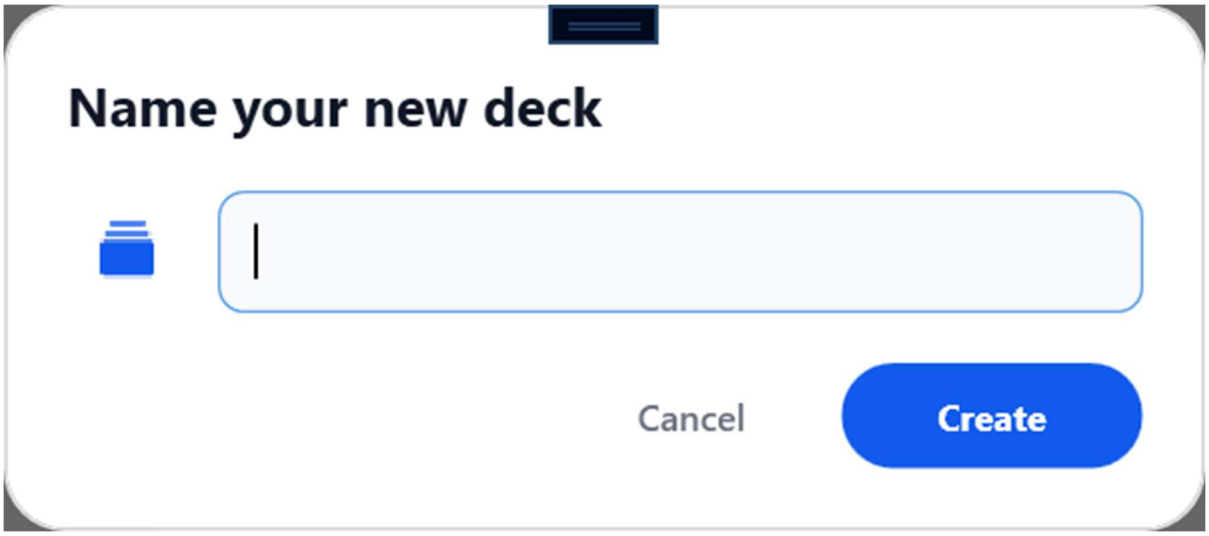
Snapshot 1



Snapshot 1: The Main Dashboard

The Dashboard serves as the entry point and command center of the application. On the left side, a scrollable list displays all the user's available "Decks," allowing for easy navigation between different subjects. When a deck is selected, the right panel populates with a summary of the user's progress. This screen also provides the primary action buttons, allowing the user to "Start Study" for the selected subject, create a new deck, or edit an existing one.

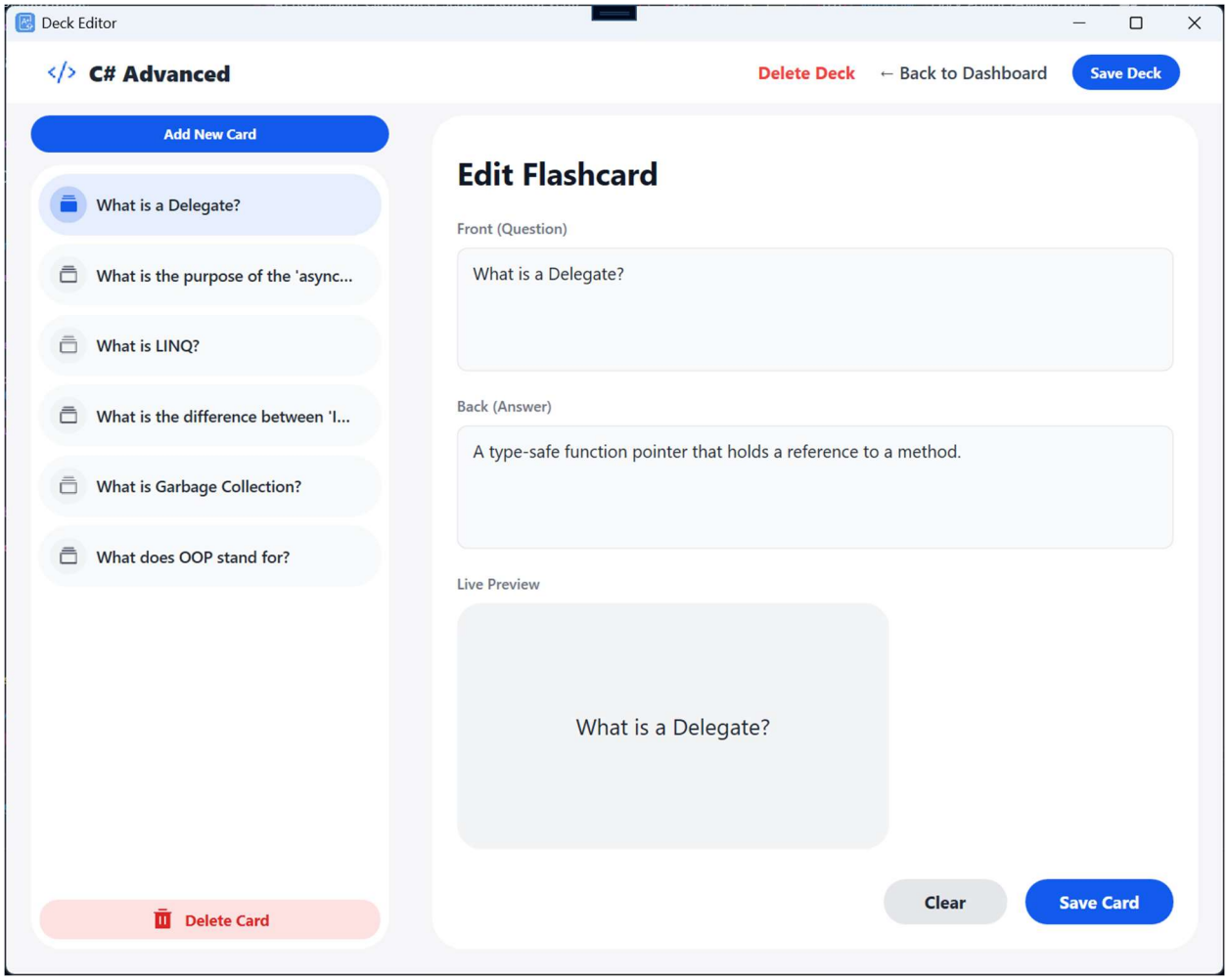
Snapshot 2



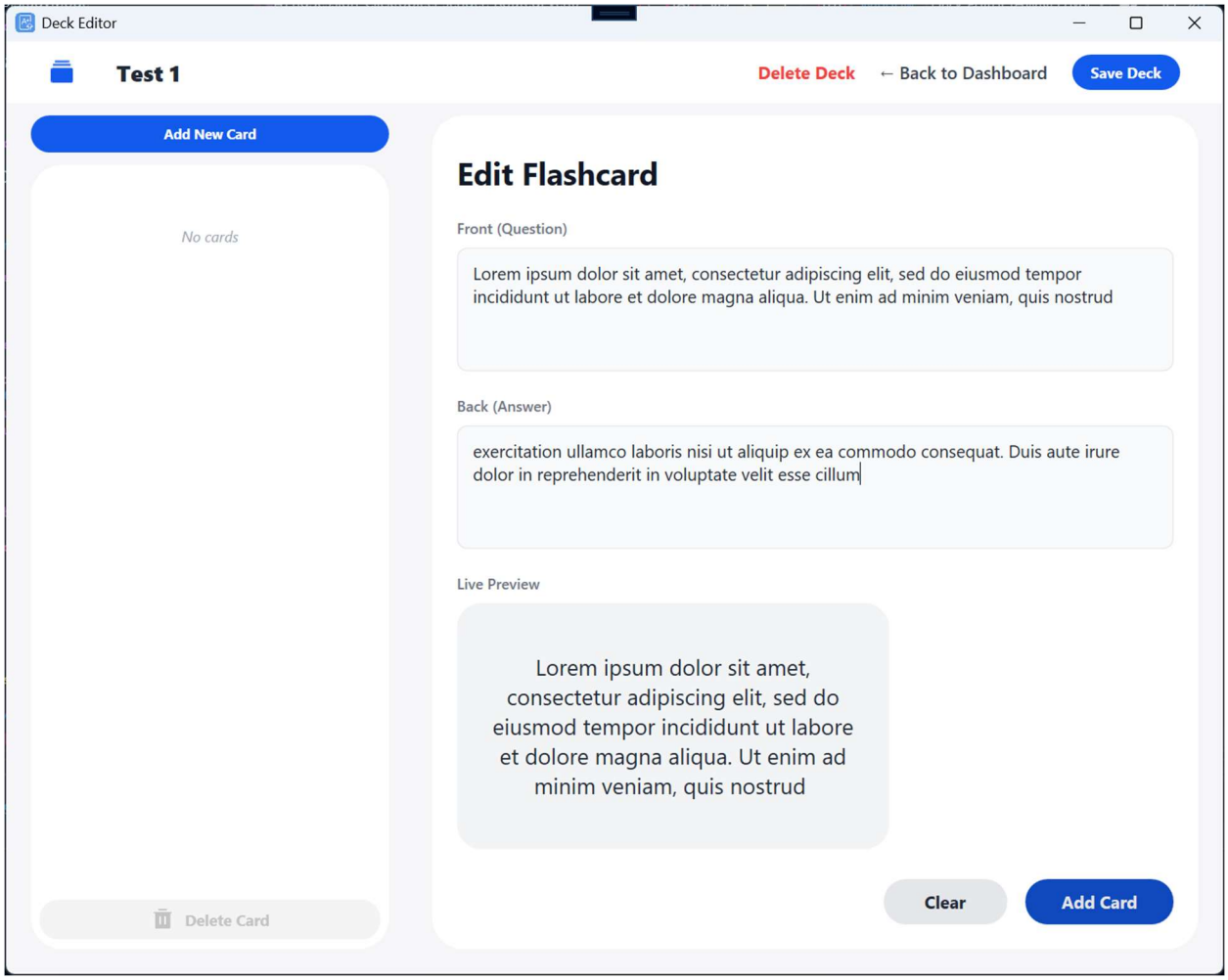
Snapshot 2: New Deck Window

This small window appears after you click Create Deck in the Dashboard and is a bridge to the deck editor window once you name the deck and press create. The create button will not function if you do not name your new deck. You can also personalize the icon by clicking on it, it will give you many options for the user to choose upon the context of the deck.

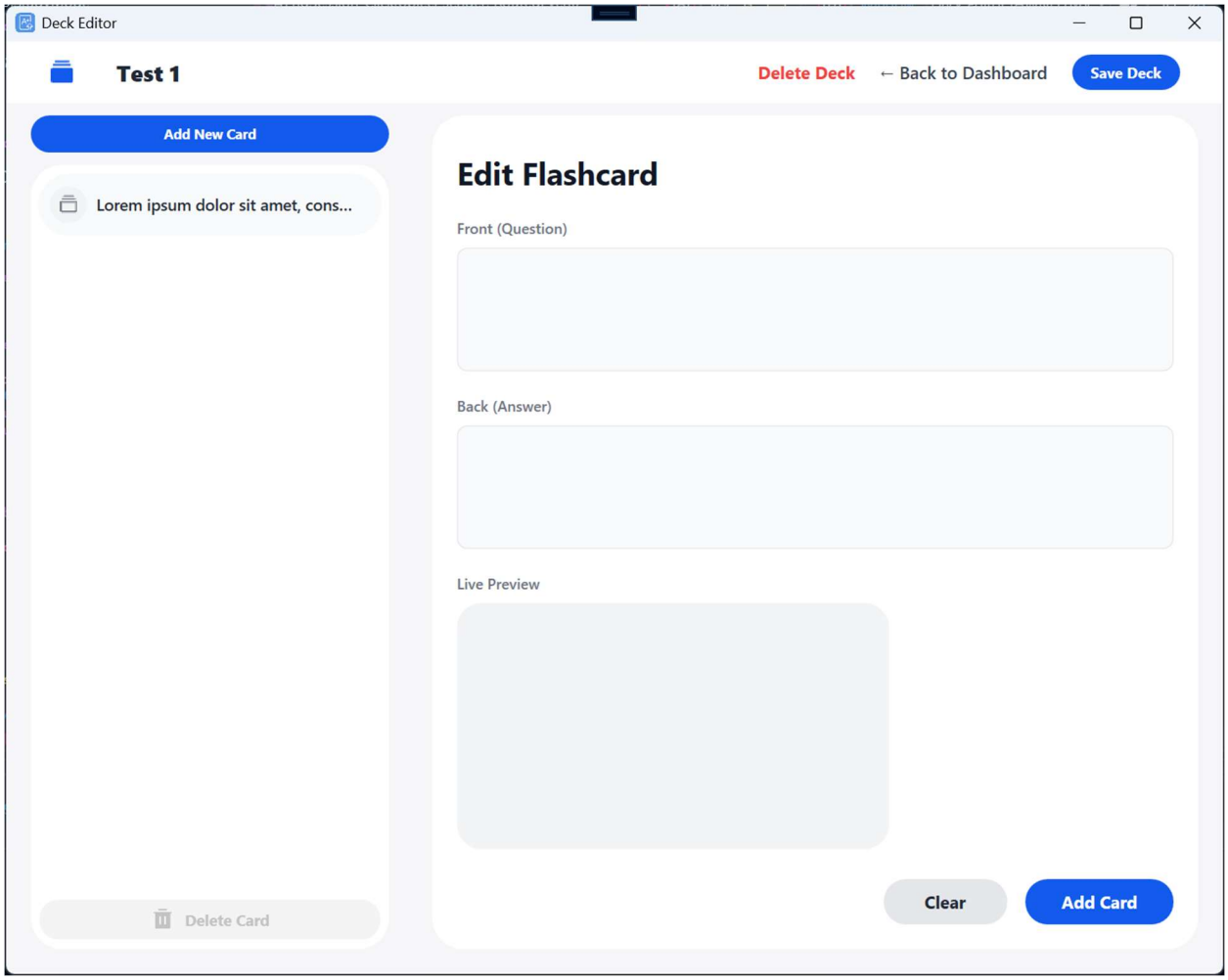
Snapshot 3


Deck Editor
C# Advanced
Delete Deck
← Back to Dashboard
Save Deck
Add New Card
What is a Delegate?
What is the purpose of the 'async...
What is LINQ?
What is the difference between 'I...
What is Garbage Collection?
What does OOP stand for?
Delete Card
Edit Flashcard
Front (Question)
What is a Delegate?
Back (Answer)
A type-safe function pointer that holds a reference to a method.
Live Preview
What is a Delegate?
Clear
Save Card

Snapshot 4


Deck Editor
Test 1
Delete Deck
← Back to Dashboard
Save Deck
Add New Card
No cards
Delete Card
Edit Flashcard
Front (Question)
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
Back (Answer)
exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum
Live Preview
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
Clear
Add Card

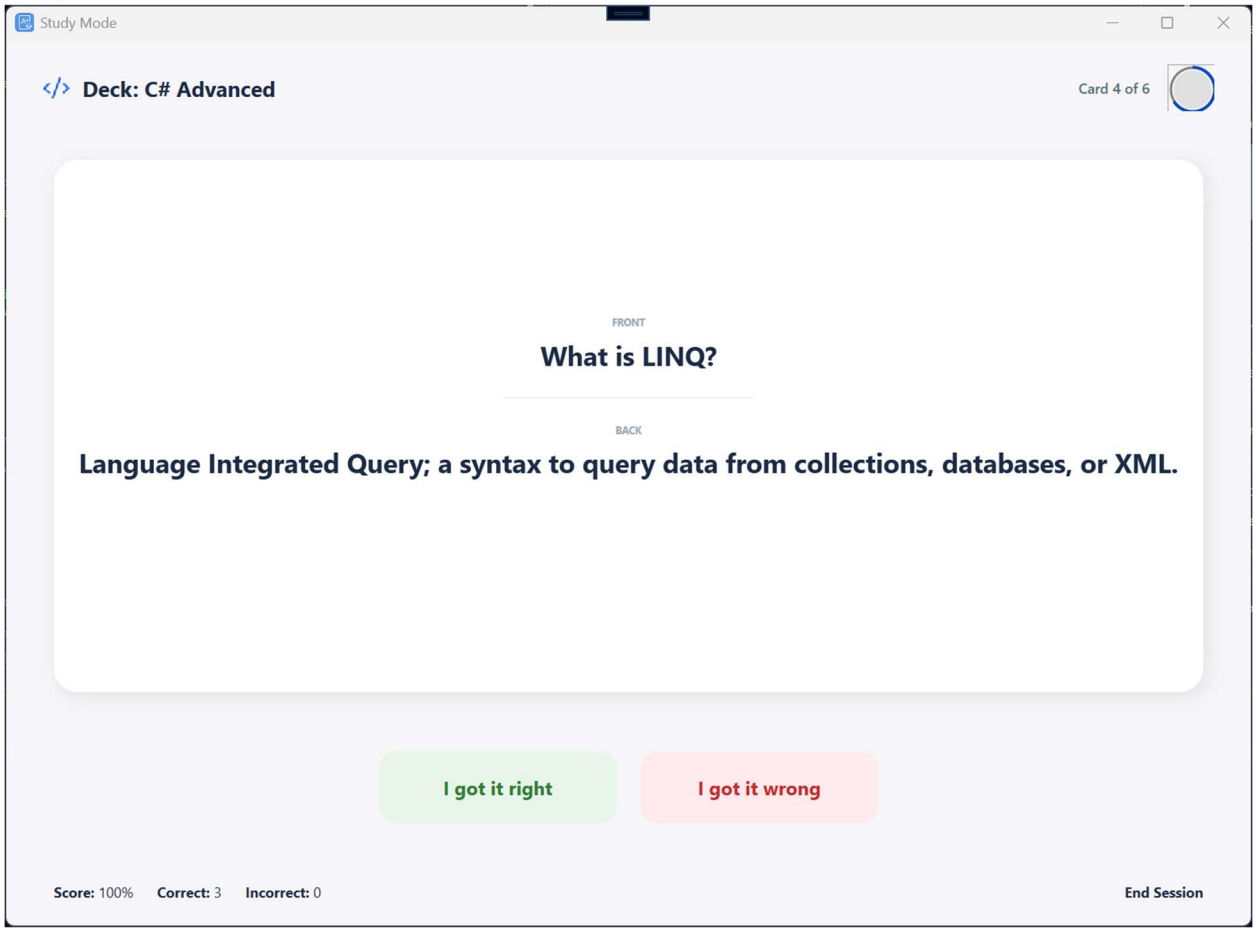
Snapshot 5



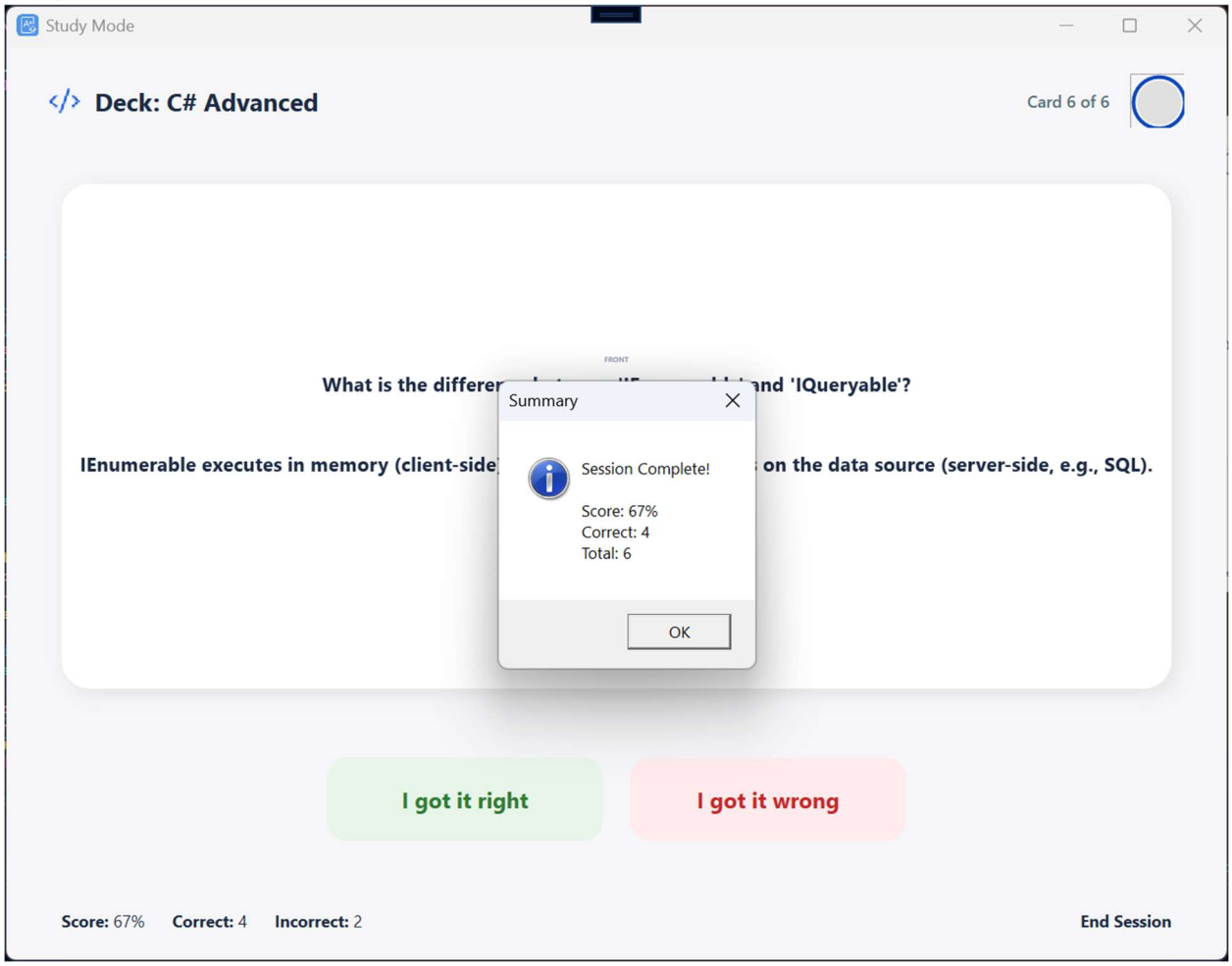
Snapshot 3,4 and 5: The Deck Editor Interface

This screen is the content creation hub where users manage the specific material within a new or existing deck. It features a split view: the bottom section provides text input fields where users type the "Front" (question) and "Back" (answer) of a new flashcard. Above this, a list view shows all currently saved cards in the deck, allowing users to verify their content as they build it. The interface enforces data integrity by disabling the "Add" feature if fields are empty, ensuring that only valid study materials are saved to the collection.

Snapshot 6


Study Mode
Deck: C# Advanced
Card 4 of 6
FRONT
What is LINQ?
BACK
Language Integrated Query; a syntax to query data from collections, databases, or XML.
I got it right
I got it wrong
Score: 100%
Correct: 3
Incorrect: 0
End Session

Snapshot 7



Snapshot 6 and 7: The Active Study Mode

The Study Mode screen is designed for distraction-free learning, mimicking a real-world quiz environment. In its initial state, it displays a large card showing only the "Front" text (the question) with a "Reveal Answer" button below it. Once the user attempts to recall the information and clicks "Reveal," the UI updates to show the "Back" text (the answer) and unlocks the two self-assessment buttons: "Correct" and "Incorrect". This interactive loop continues until all cards in the randomized session are completed, at which point the session data is recorded. The card number in deck is tracked with not only a textblock but with a Circular Progress bar using NuGet Packages.

Future Work

There are several key areas where this application could be expanded to enhance its functionality and user experience.

- Multimedia Support: Currently, flashcards are limited to string values. Future iterations could implement support for image and audio assets. This would significantly increase the application's versatility, enabling study modes for visual subjects (such as Art History) or language learning via pronunciation audio.
- UI/UX Refinements: The user interface could be improved by implementing animated circular progress bars for a more dynamic look, as well as enabling text wrapping to support multi-line content on flashcards.
- Theming: Implementing a system-wide "Dark Mode" would improve accessibility and accommodate user preferences for low-light environments.

Team Contract

Team Members

Role	Name	Student ID
Member A	**Ryan Morov**	**2492176**
Member B	**Felipe Mesa Paredes**	**2466265**

Strengths and Weaknesses

Within the context of this project, what are the strengths and weaknesses that each member brings to the team?

Member A

- **Strengths:** Strong background in C# logic, object-oriented principles, and handling file operations.
- **Weaknesses:** Less experienced with advanced WPF XAML styling and complex UI layouts.

Member B

- **Strengths:** Excellent skills in UI/UX design, creating responsive layouts, and organizing project documentation.
- **Weaknesses:** Sometimes struggles with complex algorithmic logic (like randomization algorithms) or debugging deep backend issues.

Definition of "good enough" for this project

What would the team collectively consider "good enough" of an achievement for the project? (One response for the whole team)

We aim for a grade of at least 90%. "Good enough" means the application is fully functional without crashing and meets all the core requirements: creating/editing decks, running a study session, and accurately saving/loading data. We agree that a stable, clean application submitted on time is better than a feature-rich app that is buggy or late.

Picked Topic

- **Topic 2: Flashcard Study App (Learning / Study)**

Division of Work

How will each member contribute to the project?

Member A (Backend and Logic Focus)

- **Models and Repos:** Creating the Deck and Card classes.
- **Data Persistence:** Implementing the Save/Load logic to ensure decks and cards are persisted to text/CSV files.
- **Quiz Logic:** Writing the algorithms for shuffling cards in Quiz Mode and calculating the final percentage score.

Member B (Frontend and UI Focus)

- **Views and Navigation:** Designing the windows using WPF.
- **Interactivity:** Handling UI events (for example, flipping the card from Front to Back, marking cards as "Got it" or "Missed").
- **Documentation:** Compiling the final report, formatting the user guide, and ensuring the UI meets the "Appearance" requirements.

Frequency of Communication

How often will the team be in touch and what tools will be used to communicate?

- We will communicate daily via Discord for quick updates and file sharing.

Response Delays

What is a reasonable delay to reply to messages? Is it the same for weekdays and weekends?

- **Weekdays:** We expect a reply within 4 to 6 hours.
- **Weekends:** We expect a reply within 12 hours.

If a member is unable to work on the project for a specific day, they must notify the other member in advance.

Receiving Feedback

Each member must provide a sample sentence for how they would like to receive constructive feedback from their peers.

- **Member A:** "I prefer direct, technical feedback. For example: 'The load function is throwing an error on empty lines; can you look at the loop condition again?'"
- **Member B:** "I prefer feedback that suggests improvements rather than just pointing out errors. For example: 'The study screen looks good, but maybe we can increase the font size on the cards to make them more readable?'"

In Case of Conflict

If a team member fails to communicate as described in this contract or does not respond to constructive feedback, what measures should the other teammate take? (One response for the whole team)

1. **Direct Reminder:** Send a message via Discord tagging the member, restating the pending task and the deadline.
2. **Call:** If there is no response within 24 hours, attempt a direct voice/video call.
3. **Professor Mediation:** If communication breaks down for more than 48 hours or if a critical deadline is at risk, we will contact the professor for guidance.

UML Class design