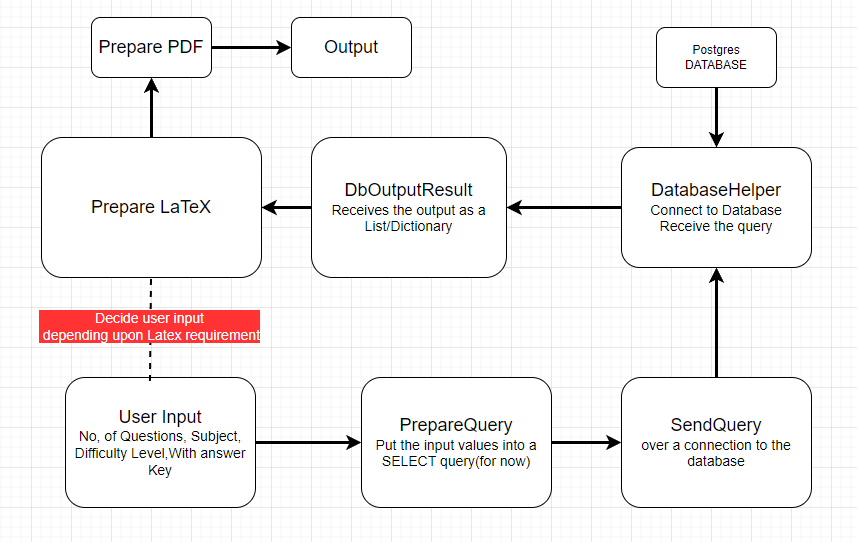
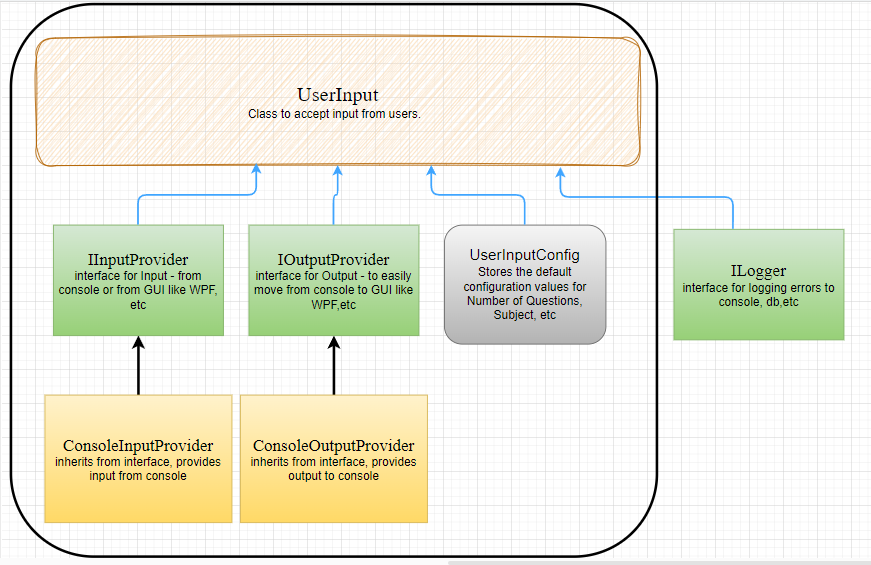
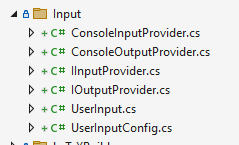
**//----------------------------------------------------** Paper Generator App **----------------------------------------------------------//**// The aim of this project is as follows:  
**0) The high level overview is that I can create a pdf quickly containing a certain number of questions from custom subject and difficulty level after accessing it from a database. This helps me in providing Daily Practise Papers, Assignments, Test Papers to students. The primary backbone is A HUGE DATABASE.**  
1) I have a database (created in a separate project) that contains Questions, Answers in LaTeX format. For the database schema, refer to the other project. I might discuss more about it later.   
2)I want to access questions and answer options from the database through a query using ADO.NET.  
3) The data thus returned is then read and passed to a LaTeX generator class – which uses stringbuilder, to build a LaTeX file of those questions and answer options. Both questions and answer options might contain images – depending upon which LaTeX snippets are Added for creating the final LaTeX string.  
4) Then the string is converted to a TeX file using File.  
5) The pdflatex / pdftex process is then started to convert the TeX file into pdf. This is the final output.

Architecture:   
I will try to maintain and achieve all the coding standards I keep learning in books – like reduce redundancy (less repetition), separation of concerns, single responsibility of a class (so that scaling or modifying code in future becomes easy), promote loose coupling as much as possible, dependency inversion using constructor injection (dependency injection pattern) , etc.   
But this was not how I started. I first tried to just check whether my idea (those 5 points above) would work in practice or not. A lot of trials went into getting the LaTeX format right ( using online LaTeX compilers ). First I tried to simply render the pdf with at least 2 questions or just the document title. Then tried to slowly apply the above coding standards. I also had to visualize the folder structure & the high level over view. The draw.io app helped me a lot to visualize. I need to work more on this.  
This was the initial overview after a lot of discussion and going back and forth between DeepSeek and StackExchange and ChatGPT. I realized that I must be very clear about the goals of every class or section- write exactly what it needs to achieve.

Input System : The broad idea was that I must take input from the user for the Number of Questions he wants, the Difficulty Level of those questions, the Subject of those questions and whether he wants an answer key included at the end of final pdf. I want to expand more on this, for example, introducing a custom difficulty level where a user can ask for few medium difficulty level questions and few easy level, or any such combination thereof.  
The implementation was initially done in the Program.cs class itself,but obviously Input could be a class in itself. All user input can be & should be handled separately from the other parts of the code.  
To make code maintainable, scalable, testable, alltypesofAble and apply separation of concerns, this Input class quickly involved into something weird.

//Encapsulating user input logic within a dedicated class (UserInput), is a good practice since we now have **Separation of Concerns, Testability and easy Validation.** Separating input handling from query generation aligns with the Single Responsibility Principle. By isolating user input logic, you can mock or test the UserInput class independently. Public properties make it easier to inject test data without relying on Console.ReadLine. Using properties to store input values allows for flexibility in extending the class with additional validation or transformation logic. The modular design makes it easier to adapt to future requirements, such as adding new input parameters.   
// Moreover, we can still **decouple Input Source ,** using dependency injection to pass an input source (e.g., IInputProvider interface) instead of directly using Console.ReadLine. This improves reusability and testability. **Validation and Immutability:** Add validation logic to ensure input values are valid before setting properties. Consider making properties immutable (e.g., using init or constructor injection) to prevent unintended changes.  
//Don’t make the default values hardcoded within the class, otherwise makes it inflexible. We can improve this by allowing default values to be configurable (e.g., via constructor parameters or configuration files). This improves flexibility and adheres to Open/Closed Principle.  
**(Not Yet Implemented)**// Scalability Problem: Adding new input parameters requires modifying the class, which can lead to frequent changes. Improvement: Use a more flexible design (e.g., a dictionary or configuration object) to handle additional parameters without modifying the class.