--- PAGE 1 ---

1. Problem Statement: Building a Document Editor

We need to design a **Document Editor** (like Google Docs/MS Word) that supports basic editing features but is highly scalable.

Core Features:

- Add Text
- Add Images
- **Scalability:** In the future, it should support Videos, Tables, Fonts, etc., without rewriting the whole code.

Design Approaches:

Before coding, we must choose an approach.

The following table:

| Approach | Description |

| :--- | :--- |

| **Top-Down** | Start with the high-level system (Main Object) and break it down into smaller components. |

| **Bottom-Up** | Start by creating small, independent objects (like Text, Image) and combine them to build the system. (Preferred for this problem). |

2. The "Bad" Design (Monolithic Approach)

In a naive design, we create a single class DocumentEditor.

How it works (The Flaw):

- **Storage:** A single list (vector<string>) stores both text and image paths.
- **Rendering:** A messy loop checks if a string ends in .jpg to decide if it's an image.
- **Saving:** The same class handles file saving logic.

Hinglish Analogy:

Ye wahi baat ho gayi ki ek hi insaan Khana bhi bana raha hai, Order bhi le raha hai, aur Delivery bhi kar raha hai. (Breaking Single Responsibility Principle).

Agar kal ko "Video" add karni ho, toh puri class modify karni padegi (Breaking Open-Closed Principle).

--- PAGE 2 ---

3. The "Good" Design (Applying SOLID Principles)

To fix the bad design, we separate responsibilities using **Polymorphism** and **Abstract Classes**.

Step 1: Document Element (Polymorphism)

Instead of if-else checks for Text/Image, we create a common contract.

- **Abstract Class:** DocumentElement with a render() method.
- **Child Classes:** TextElement, ImageElement, TableElement.
- **Benefit:** The main document doesn't need to know *what* the element is; it just calls .render().

Step 2: Separation of Concerns

The following table explains the new architecture:

Class	Responsibility
DocumentElement	Knows how to render itself (Text vs Image).
Document	Holds the list of elements (CRUD operations only).
Persistence	Handles saving data (File vs Database).
DocumentEditor	Acts as a controller/facade for the user to interact with.

4. SOLID Principles in Action

SRP (Single Responsibility):

- Document manages data. Persistence manages saving. Editor manages user interaction.
  OCP (Open/Closed):
- Want to add "Video"? Just create a VideoElement class. No need to touch the existing code.
  LSP (Liskov Substitution):
- Document treats TextElement and ImageElement exactly the same (as DocumentElement).

--- PAGE 3 ---

5. Deep Dive: The Execution Flow

How the request travels from User to System:

The following table:

| Step | Action | Explanation |

| :--- | :--- | :--- |

| **1. User Action** | editor.addText("Hello") | User interacts with the DocumentEditor. |

| **2. Delegation** | document.add(new TextElement()) | Editor delegates creation to the Document class. |

| **3. Rendering** | loop { element.render() } | To show the doc, we loop through the list and call render() on each object. |

| **4. Saving** | persistence.save(doc) | Editor asks Persistence layer to save the rendered output. |

6. Advanced Concept: Principle of Least Knowledge

Definition: Also known as the *Law of Demeter*.

"Talk only to your immediate friends, not strangers."

The Conflict:

- If we create a separate DocumentRenderer class, it might need to access internal lists of Document.
- This creates tight coupling (Dependency).
- **Trade-off:** Sometimes we keep rendering logic inside Document (or close to it) to avoid exposing internal data structures to the outside world.

--- PAGE 4 ---

Interview Questions

1. Why use Bottom-Up approach for LLD?

Answer: In LLD, defining small, independent components (like TextElement, Button) first makes it easier to compose complex systems. Top-down is often better for HLD (High-Level Design).

2. How does this design follow the Open-Closed Principle?

Answer: The system is **Open for extension** (we can add VideoElement, TableElement classes) but **Closed for modification** (we don't change the Document class logic to support new types).

3. What is the difference between Document and DocumentEditor?

Answer:

- Document: The **Model**. It holds the data (list of elements).
- DocumentEditor: The **Controller**. It handles user commands (add, save, render) and coordinates between the Model and Persistence layers.

4. Why did we make Persistence an abstract class?

Answer: To support multiple storage methods (File System, SQL Database, Cloud) without changing the main application logic. We can switch from FileSave to DBSave easily.

5. Explain "Dependency Inversion" in this project.

Answer: High-level modules (DocumentEditor) do not depend on low-level modules (FileStorage). Both depend on abstractions (Persistence interface). The Editor doesn't care *how* data is saved, only *that* it is saved.