**Explanation of Design**

* Object-Oriented Design: By using inheritance for the document types, I promote code reusability and maintainability. Each document type can be managed separately, and future document types can easily be added by creating new classes.
* Database Design: The normalized structure of having separate tables for type-specific document data reduces redundancy and allows for efficient querying, especially for type-specific searches.
* Folder Hierarchy: Using a self-referencing ParentFolderID in the Folders table allows for a flexible and scalable folder system.
* Search and Reports: SQL querying offers efficient searching, while reporting is streamlined by joining the document and folder data, allowing for custom exports.

This approach combines strong OOP practices with a flexible, scalable data model to meet all the requirements for document management, searching, and reporting in a Delphi application.

For this Delphi-based desktop application, I break down the task into three key components:

1. Data model.
2. Folder and document management.
3. User interface (UI).

Here's a structured approach to design and implementation:

1. **Data Model**

The data model will use two main tables: Folders and Documents. Each document type will have its own set of type-specific attributes, and I handle them using object-oriented programming (OOP) principles in Delphi.

**Tables Structure:**

**Folders Table:**

FOLDER\_ID (Primary Key)

FOLDER\_PARENT (Foreign Key for recursive relationships, allowing n-level hierarchy)

FOLDER\_NAME

**Documents Table:**

DOCUMENT\_ID (Primary Key)

FOLDER\_ID (Foreign Key, links to Folders)

DOCUMENT\_NAME

DOCUMENT\_ADDED

DOCUMENT\_MODIFIED

DOCUMENT\_SIZE

DOCUMENT\_TYPE (e.g., "txt", "report", "article")

DOCUMENT\_DESCRIPTION

DOCUMENT\_TAGS

DOCUMENT\_AUTHOR: **Report Documents, Article Documents**

DOCUMENT\_SUMMARY: **Report Documents:**

DOCUMENT\_PUBLICATION: **Article Documents**

DOCUMENT\_VERSION: **PDF Documents**

DOCUMENT\_ENCRYPTION: **PDF Documents**

**Object-Oriented Approach**

I create a class structure where each document type inherits from a base class, leveraging Delphi's strong support for OOP.

Classes:

**TDocument (Base Class):**

ID: string;

Name: string;

DateAdded: string;

DateModified: string;

Size: Integer;

Type: string;

Description: string;

KeyTags: string;

**TTextDocument (Inherited Class)**

**TReportDocument (Inherited Class):**

Author: string;

Summary: string;

**TArticleDocument (Inherited Class):**

PublicationDate: string;

Author: string;

**TPDFDocument (Inherited Class):**

Version: string;

EncryptionStatus: Boolean;

Note: The datetime data type is changed to match any type of data base and not present incompatibility, thus making migration between data bases easier.

Each document type has methods for loading, saving, and modifying its data, making it easier to manage the specifics of each document type.

Folders and Hierarchy Management:

I use a recursive function to manage the folder hierarchy, utilizing the ParentFolderID to navigate the folder tree.

2. **Folder and Document Management**

The core of the document management system includes the following:

Adding/Removing Folders:

Users can add a folder with a parent folder reference (FolderParentID).

If the folder is at the root, FolderParentID is null.

Removing a folder should recursively remove all subfolders and associated documents.

Adding/Removing Documents:

Documents added to a specific folder, using the FolderID as a reference.

Type-specific document data handled via the inherited classes (e.g., saving an article document include the article's specific attributes like Author and PublicationDate).

**Search Functionality**

* I can implement a search function to query the database for documents based on several criteria, such as:
* Name, Date, Description, Key Tags
* Document type-specific attributes, such as Author for articles or reports.
* This done using SQL queries to the Documents table and the respective type-specific tables (e.g., searching by Author will require a join with the ReportDocuments or ArticleDocuments tables).

**Report Generation**

A reporting feature will allow the user to generate a list of documents either within a folder or across all folders. This report will display document details and can be exported to various formats such as PDF or CSV.

**3. User Interface**

* The UI include:
* A TreeView component to display the folder hierarchy, allowing users to expand and collapse folders.
* A ListView to display documents within the selected folder.
* Dialogs for adding/removing folders and documents.
* Search fields for querying documents.
* The folder management straightforward, with buttons to add/remove folders and documents. For document-specific actions, context menus can be added to the ListView.

Lesson learned, the document table should be further normalized as shown below, offering, this database design provides flexibility in handling different document types and their respective attributes.

**Tables Structure:**

* **Folders Table:**
  + FOLDER\_ID (Primary Key)
  + FOLDER\_PARENT (Foreign Key for recursive relationships, allowing n-level hierarchy)
  + FOLDER\_NAME
* **Documents Table:**
  + DOCUMENT\_ID (Primary Key)
  + FOLDER\_ID (Foreign Key, links to Folders)
  + DOCUMENT\_NAME
  + DOCUMENT\_ADDED
  + DOCUMENT\_MODIFIED
  + DOCUMENT\_SIZE
  + DOCUMENT\_TYPE (e.g., "txt", "report", "article")
  + DOCUMENT\_DESCRIPTION
  + DOCUMENT\_TAGS
* **Type-specific tables for documents:**
  + **Report Documents:**
    - DOCUMENT\_ID (Foreign Key)
    - DOCUMENT\_AUTHOR
    - DOCUMENT\_SUMMARY
  + **Article Documents:**
    - DOCUMENT\_ID (Foreign Key)
    - DOCUMENT\_PUBLICATION
    - DOCUMENT\_AUTHOR
  + **PDF Documents:**
    - DOCUMENT\_ID (Foreign Key)
    - DOCUMENT\_VERSION
    - DOCUMENT\_ENCRYPTION

All the operations of the implementation were carried out following the initial directives of the technical document delivered.