Comprehensive Tutorial: Developing a C# Desktop Application with Database Integration

Table of Contents

[Introduction 1](#_Toc461297968)

[Project Overview 1](#_Toc635064185)

[Step 1: Requirements Analysis 1](#_Toc1790328340)

[Functional Requirements 1](#_Toc418441740)

[Book Management 1](#_Toc985611320)

[Member Management 1](#_Toc1706938800)

[Borrowing Management 1](#_Toc737340460)

[Non-Functional Requirements 1](#_Toc94297685)

[Step 2: Software Architecture 1](#_Toc936104782)

[Why Use Layered Architecture? 1](#_Toc1786865956)

[Step 3: UML Diagrams 1](#_Toc1950030646)

[Use Case Diagram 1](#_Toc298140767)

[Class Diagram 1](#_Toc454220545)

[Explanation: 1](#_Toc465988534)

[Activity Diagram 1](#_Toc441574891)

[Step 4: Database Design 1](#_Toc489785262)

[Database Normalization 1](#_Toc1980016268)

[Step 5: Implementing the Database 1](#_Toc704760206)

[Step 6: C# Programming 1](#_Toc1565776966)

[Creating Models 1](#_Toc839631925)

[6.3 Data Access Layer (DAL) 1](#_Toc267443465)

[Business Logic Layer (BLL) 1](#_Toc246832658)

[Presentation Layer (UI) 1](#_Toc391869967)

[Step 7: Implementing CRUD Operations 1](#_Toc1356970463)

[Step 8: Implementing Search Functionality 1](#_Toc1782000812)

[Step 9: Object-Oriented Programming Principles 1](#_Toc1163408707)

# Introduction

This tutorial provides a comprehensive, step-by-step guide to developing a desktop application using C# and integrating it with a database. We'll focus on object-oriented programming, software architecture, UML diagrams, and database modeling, including normalization and the use of primary and foreign keys.

Our tangible example will be a **Library Management System**, which will allow us to cover a variety of programming concepts and techniques.

## Project Overview

The Library Management System will enable users (librarians) to manage books, members, and borrowing transactions. The application will feature:

- **CRUD Operations**: Create, Read, Update, Delete functionalities for books and members.

- **Database Integration**: Using SQL Server for data storage.

- **Search Functionality**: Ability to search for books and members.

- **Layered Architecture**: Separation of concerns using distinct layers (UI, BLL, DAL).

- **UML Diagrams**: Visual representations of the system's structure and behavior.

# Step 1: Requirements Analysis

## Functional Requirements

### Book Management

- Add new books.

- Update existing book details.

- Delete books.

- View a list of all books.

### Member Management

- Add new members.

- Update member information.

- Delete members.

- View a list of all members.

### Borrowing Management

- Issue books to members.

- Process the return of books.

- View borrowing history.

## Non-Functional Requirements

- User-Friendly Interface: Intuitive and easy to navigate.

- Data Security: Protect sensitive information.

- Performance: Efficient data processing.

- Scalability: Ability to handle growth.

- Maintainability: Clean and organized codebase.

# Step 2: Software Architecture

We will use a Layered Architecture to structure our application, which includes:

1. Presentation Layer (UI Layer)

- Handles user interactions.

- Contains forms and controls.

- Communicates with the Business Logic Layer.

2. Business Logic Layer (BLL)

- Implements business rules and validations.

- Processes data between the UI and DAL.

- Ensures data integrity.

3. Data Access Layer (DAL)

- Handles database operations.

- Executes CRUD operations.

- Manages database connections.

4. Database Layer

- Stores data persistently.

- Enforces data constraints.

- Manages relationships via primary and foreign keys.

## Why Use Layered Architecture?

- **Separation of Concerns**: Each layer has a specific responsibility.

- **Maintainability**: Easier to manage and update.

- **Reusability**: Components can be reused across the application.

- **Testability**: Layers can be tested independently.

# Step 3: UML Diagrams

Unified Modeling Language (UML) diagrams help visualize the system's structure and behavior. Below, we'll provide textual representations of the diagrams to illustrate the system components and their interactions.

## Use Case Diagram

**Actors:**

- Librarian

Use Cases:

- Manage Books

- Add Book

- Update Book

- Delete Book

- View Books

- Manage Members

- Add Member

- Update Member

- Delete Member

- View Members

- Manage Borrowing

- Issue Book

- Return Book

- View Borrowing History

Textual Representation:

```

[Use Case Diagram]

Actor: Librarian

Use Cases:

1. Manage Books

- Add Book

- Update Book

- Delete Book

- View Books

2. Manage Members

- Add Member

- Update Member

- Delete Member

- View Members

3. Manage Borrowing

- Issue Book

- Return Book

- View Borrowing History

Relationships:

- Librarian initiates all use cases.

Explanation:the


- The **Librarian** interacts with the system to perform various tasks.

- Each **Use Case** represents a functional requirement.

- The relationships indicate that the Librarian can perform all the listed actions.

## Class Diagram

Classes, Attributes, and Methods:

Book

- **Attributes:**

- `int BookID` \*(Primary Key)\*

- `string Title`

- `string Author`

- `string ISBN`

- `string Publisher`

- `int Year`

- \*\*Methods:\*\*

- `void AddBook()`

- `void UpdateBook()`

- `void DeleteBook()`

- `List<Book> GetBooks()`

2. \*\*Member\*\*

- \*\*Attributes:\*\*

- `int MemberID` \*(Primary Key)\*

- `string Name`

- `string Email`

- `string Phone`

- \*\*Methods:\*\*

- `void AddMember()`

- `void UpdateMember()`

- `void DeleteMember()`

- `List<Member> GetMembers()`

3. \*\*Borrowing\*\*

- \*\*Attributes:\*\*

- `int BorrowingID` \*(Primary Key)\*

- `int BookID` \*(Foreign Key)\*

- `int MemberID` \*(Foreign Key)\*

- `DateTime BorrowDate`

- `DateTime? ReturnDate`

- \*\*Methods:\*\*

- `void IssueBook()`

- `void ReturnBook()`

- `List<Borrowing> GetBorrowingHistory()`

\*\*Textual Representation:\*\*

```

[Class Diagram]

Class: Book

-----------

- BookID: int [PK]

- Title: string

- Author: string

- ISBN: string

- Publisher: string

- Year: int

Methods:

+ AddBook()

+ UpdateBook()

+ DeleteBook()

+ GetBooks(): List<Book>

Class: Member

-------------

- MemberID: int [PK]

- Name: string

- Email: string

- Phone: string

Methods:

+ AddMember()

+ UpdateMember()

+ DeleteMember()

+ GetMembers(): List<Member>

Class: Borrowing

----------------

- BorrowingID: int [PK]

- BookID: int [FK]

- MemberID: int [FK]

- BorrowDate: DateTime

- ReturnDate: DateTime?

Methods:

+ IssueBook()

+ ReturnBook()

+ GetBorrowingHistory(): List<Borrowing>

Relationships:

- Book 1 --- \* Borrowing (One Book to Many Borrowings)

- Member 1 --- \* Borrowing (One Member to Many Borrowings)

```

#### Explanation:

- **Classes** represent entities in the system.

- **Attributes** are the properties of each class.

- **Methods** define the behaviors or actions that can be performed.

- **Relationships** show how classes are connected, with multiplicity indicating one-to-many relationships.

Sequence Diagram

**Example: Issuing a Book**

\*\*Textual Representation:\*\*

```

[Sequence Diagram]

Actors and Objects:

- Librarian

- UI Layer (BookForm)

- BLL (BookBLL, BorrowingBLL)

- DAL (BookDAL, BorrowingDAL)

- Database

Sequence of Actions:

1. Librarian clicks "Issue Book" button.

2. UI Layer calls BookBLL.ValidateBookAvailability(bookID).

3. BookBLL calls BookDAL.GetBookByID(bookID).

4. BookDAL queries Database and returns Book object.

5. BookBLL checks if Book is available.

6. UI Layer collects MemberID and BookID.

7. UI Layer calls BorrowingBLL.IssueBook(memberID, bookID).

8. BorrowingBLL calls BorrowingDAL.AddBorrowingRecord(borrowing).

9. BorrowingDAL inserts record into Database.

10. UI Layer displays confirmation to Librarian.

```

\*\*Explanation:\*\*

- Shows the flow of messages between objects when a Librarian issues a book.

- Emphasizes the interactions between layers (UI, BLL, DAL).

- Highlights the validation and data processing steps.

### Activity Diagram

\*\*Example: Returning a Book\*\*

\*\*Textual Representation:\*\*

```

[Activity Diagram]

Start

|

|--> Librarian selects "Return Book" option.

|

|--> System prompts for BorrowingID.

|

|--> Librarian enters BorrowingID.

|

|--> System retrieves Borrowing record.

|

|--> [Decision] Is Borrowing record valid?

| Yes --> Proceed

| No --> Display error message --> End

|

|--> System updates ReturnDate in Borrowing record.

|

|--> System updates Book status to "Available".

|

|--> System confirms return to Librarian.

|

End

```

\*\*Explanation:\*\*

- Depicts the workflow for returning a book.

- Includes decision points and possible outcomes.

- Illustrates the steps taken by the system and the Librarian.

---

# Step 4: Database Design

**Entity-Relationship Diagram (ERD)**

**Entities, Attributes, and Relationships:**

1. Book

- Attributes:

- `BookID` \*(Primary Key)\*

- `Title`

- `Author`

- `ISBN`

- `Publisher`

- `Year`

**2. Member**

- Attributes:

- `MemberID` \*(Primary Key)\*

- `Name`

- `Email`

- `Phone`

**3. Borrowing**

- \*\*Attributes:\*\*

- `BorrowingID` \*(Primary Key)\*

- `BookID` \*(Foreign Key)\*

- `MemberID` \*(Foreign Key)\*

- `BorrowDate`

- `ReturnDate`

Textual Representation:

[Entity-Relationship Diagram]

Entities:

- Book [BookID, Title, Author, ISBN, Publisher, Year]

- Member [MemberID, Name, Email, Phone]

- Borrowing [BorrowingID, BookID, MemberID, BorrowDate, ReturnDate]

Relationships:

- Book (1) --- (M) Borrowing

[BookID in Borrowing references BookID in Book]

- Member (1) --- (M) Borrowing

[MemberID in Borrowing references MemberID in Member]

\*\*Explanation:\*\*

- **Entities** are the tables in the database.

- **Attributes** are the columns in each table.

- **Primary Keys (PK)** uniquely identify records.

- **Foreign Keys (FK)** establish relationships between tables.

- **Relationships** show how entities are connected, with cardinality indicating one-to-many relationships.

## Database Normalization

**First Normal Form (1NF):**

- Each table has a primary key.

- All columns contain atomic values.

- No repeating groups or arrays.

**Second Normal Form (2NF):**

- Satisfies 1NF.

- All non-key attributes are fully functionally dependent on the primary key.

**Third Normal Form (3NF):**

- Satisfies 2NF.

- No transitive dependencies; non-key attributes depend only on the primary key.

**Our database design meets the requirements of 3NF, ensuring data integrity and reducing redundancy.**

# Step 5: Implementing the Database

We'll use SQL Server for the database.

Creating Tables with Primary and Foreign Keys

```sql

-- Create Book Table

CREATE TABLE Book (

BookID INT PRIMARY KEY IDENTITY(1,1),

Title VARCHAR(255) NOT NULL,

Author VARCHAR(255) NOT NULL,

ISBN VARCHAR(20) UNIQUE,

Publisher VARCHAR(255),

Year INT CHECK (Year >= 0)

);

-- Create Member Table

CREATE TABLE Member (

MemberID INT PRIMARY KEY IDENTITY(1,1),

Name VARCHAR(255) NOT NULL,

Email VARCHAR(255) UNIQUE,

Phone VARCHAR(20)

);

-- Create Borrowing Table

CREATE TABLE Borrowing (

BorrowingID INT PRIMARY KEY IDENTITY(1,1),

BookID INT NOT NULL,

MemberID INT NOT NULL,

BorrowDate DATETIME NOT NULL DEFAULT GETDATE(),

ReturnDate DATETIME,

CONSTRAINT FK\_Book FOREIGN KEY (BookID) REFERENCES Book(BookID),

CONSTRAINT FK\_Member FOREIGN KEY (MemberID) REFERENCES Member(MemberID)

);

```

---

# Step 6: C# Programming

Setting Up the Project

- **IDE:** Visual Studio

- **Project Type:** Windows Forms App (.NET Framework)

- **Framework Version:** .NET Framework 4.7.2 (or higher)

**Project Structure:**

- **Solution Name:** `LibraryManagementSystem`

- \*\*Folders:\*\*

- `Models`

- `DataAccess`

- `BusinessLogic`

- `Presentation`

## Creating Models

Book.cs

```csharp

namespace LibraryManagementSystem.Models

{

public class Book

{

public int BookID { get; set; } // Primary Key

public string Title { get; set; }

public string Author { get; set; }

public string ISBN { get; set; }

public string Publisher { get; set; }

public int Year { get; set; }

}

}

```

\*\*Member.cs\*\*

```csharp

namespace LibraryManagementSystem.Models

{

public class Member

{

public int MemberID { get; set; } // Primary Key

public string Name { get; set; }

public string Email { get; set; }

public string Phone { get; set; }

}

}

```

\*\*Borrowing.cs\*\*

```csharp

namespace LibraryManagementSystem.Models

{

public class Borrowing

{

public int BorrowingID { get; set; } // Primary Key

public int BookID { get; set; } // Foreign Key

public int MemberID { get; set; } // Foreign Key

public DateTime BorrowDate { get; set; }

public DateTime? ReturnDate { get; set; } // Nullable

}

}

```

## 6.3 Data Access Layer (DAL)

\*\*DatabaseHelper.cs\*\*

```csharp

using System.Data.SqlClient;

namespace LibraryManagementSystem.DataAccess

{

public class DatabaseHelper

{

private readonly string connectionString = "Data Source=your\_server;Initial Catalog=LibraryDB;Integrated Security=True";

public SqlConnection GetConnection()

{

return new SqlConnection(connectionString);

}

}

}

```

BookDAL.cs

```csharp

using System.Collections.Generic;

using System.Data.SqlClient;

using LibraryManagementSystem.Models;

namespace LibraryManagementSystem.DataAccess

{

public class BookDAL

{

private DatabaseHelper dbHelper = new DatabaseHelper();

public void AddBook(Book book)

{

using (SqlConnection conn = dbHelper.GetConnection())

{

string query = "INSERT INTO Book (Title, Author, ISBN, Publisher, Year) VALUES (@Title, @Author, @ISBN, @Publisher, @Year)";

SqlCommand cmd = new SqlCommand(query, conn);

cmd.Parameters.AddWithValue("@Title", book.Title);

cmd.Parameters.AddWithValue("@Author", book.Author);

cmd.Parameters.AddWithValue("@ISBN", book.ISBN);

cmd.Parameters.AddWithValue("@Publisher", book.Publisher);

cmd.Parameters.AddWithValue("@Year", book.Year);

conn.Open();

cmd.ExecuteNonQuery();

}

}

public List<Book> GetBooks()

{

List<Book> books = new List<Book>();

using (SqlConnection conn = dbHelper.GetConnection())

{

string query = "SELECT \* FROM Book";

SqlCommand cmd = new SqlCommand(query, conn);

conn.Open();

SqlDataReader reader = cmd.ExecuteReader();

while (reader.Read())

{

books.Add(new Book

{

BookID = (int)reader["BookID"],

Title = reader["Title"].ToString(),

Author = reader["Author"].ToString(),

ISBN = reader["ISBN"].ToString(),

Publisher = reader["Publisher"].ToString(),

Year = (int)reader["Year"]

});

}

}

return books;

}

// Implement UpdateBook and DeleteBook methods similarly

}

}

```

## Business Logic Layer (BLL)

BookBLL.cs

```csharp

using System.Collections.Generic;

using LibraryManagementSystem.DataAccess;

using LibraryManagementSystem.Models;

namespace LibraryManagementSystem.BusinessLogic

{

public class BookBLL

{

private BookDAL bookDAL = new BookDAL();

public void AddBook(Book book)

{

// Validations

if (string.IsNullOrEmpty(book.Title) || string.IsNullOrEmpty(book.Author))

{

throw new ArgumentException("Title and Author are required.");

}

bookDAL.AddBook(book);

}

public List<Book> GetBooks()

{

return bookDAL.GetBooks();

}

// Implement UpdateBook and DeleteBook methods with business rules

}

}

```

**Explanation:**

- The **BLL** acts as an intermediary between the **UI** and \*\***DAL**.

- It contains business rules and validations to ensure data integrity.

- Separating business logic improves maintainability and scalability.

## Presentation Layer (UI)

Designing Forms:

- **BookForm**: For managing books.

- **MemberForm**: For managing members.

- **BorrowingForm**: For handling borrowing operations.

Example: BookForm.cs

```csharp

using System;

using System.Windows.Forms;

using LibraryManagementSystem.BusinessLogic;

using LibraryManagementSystem.Models;

namespace LibraryManagementSystem.Presentation

{

public partial class BookForm : Form

{

private BookBLL bookBLL = new BookBLL();

public BookForm()

{

InitializeComponent();

LoadBooks();

}

private void btnAdd\_Click(object sender, EventArgs e)

{

try

{

Book book = new Book

{

Title = txtTitle.Text.Trim(),

Author = txtAuthor.Text.Trim(),

ISBN = txtISBN.Text.Trim(),

Publisher = txtPublisher.Text.Trim(),

Year = int.Parse(txtYear.Text.Trim())

};

bookBLL.AddBook(book);

MessageBox.Show("Book added successfully!");

LoadBooks();

ClearFields();

}

catch (Exception ex)

{

MessageBox.Show($"Error: {ex.Message}");

}

}

private void LoadBooks()

{

dgvBooks.DataSource = bookBLL.GetBooks();

}

private void ClearFields()

{

txtTitle.Clear();

txtAuthor.Clear();

txtISBN.Clear();

txtPublisher.Clear();

txtYear.Clear();

}

// Implement Update and Delete event handlers

}

}

```

---

Step 7: Implementing CRUD Operations

Create (Insert)

Adding a New Book

- \*\*UI Layer\*\*: Collects data from input fields.

- \*\*BLL\*\*: Validates data and calls `AddBook()` in DAL.

- \*\*DAL\*\*: Executes an INSERT SQL command.

### \*\*Read (Select)\*\*

\*\*Retrieving Books\*\*

- \*\*DAL\*\*: Executes a SELECT SQL command.

- \*\*BLL\*\*: Calls `GetBooks()` from DAL.

- \*\*UI Layer\*\*: Displays the list in a grid or list view.

### \*\*Update\*\*

\*\*Updating Book Details\*\*

- \*\*UI Layer\*\*: User selects a book and modifies details.

- \*\*BLL\*\*: Validates updated data and calls `UpdateBook()` in DAL.

- \*\*DAL\*\*: Executes an UPDATE SQL command.

### \*\*Delete\*\*

\*\*Deleting a Book\*\*

- \*\*UI Layer\*\*: User selects a book to delete.

- \*\*BLL\*\*: Confirms action and calls `DeleteBook()` in DAL.

- \*\*DAL\*\*: Executes a DELETE SQL command.

---

# Step 8: Implementing Search Functionality

\*\*Adding Search Functionality for Books\*\*

### \*\*8.1 Updating the DAL\*\*

\*\*BookDAL.cs\*\*

```csharp

public List<Book> SearchBooks(string searchTerm)

{

List<Book> books = new List<Book>();

using (SqlConnection conn = dbHelper.GetConnection())

{

string query = @"SELECT \* FROM Book

WHERE Title LIKE @SearchTerm

OR Author LIKE @SearchTerm

OR ISBN LIKE @SearchTerm

OR Publisher LIKE @SearchTerm";

SqlCommand cmd = new SqlCommand(query, conn);

cmd.Parameters.AddWithValue("@SearchTerm", "%" + searchTerm + "%");

conn.Open();

SqlDataReader reader = cmd.ExecuteReader();

while (reader.Read())

{

books.Add(new Book

{

BookID = (int)reader["BookID"],

Title = reader["Title"].ToString(),

Author = reader["Author"].ToString(),

ISBN = reader["ISBN"].ToString(),

Publisher = reader["Publisher"].ToString(),

Year = (int)reader["Year"]

});

}

}

return books;

}

```

### \*\*8.2 Updating the BLL\*\*

\*\*BookBLL.cs\*\*

```csharp

public List<Book> SearchBooks(string searchTerm)

{

if (string.IsNullOrEmpty(searchTerm))

{

return bookDAL.GetBooks();

}

else

{

return bookDAL.SearchBooks(searchTerm);

}

}

```

### \*\*8.3 Updating the UI\*\*

\*\*BookForm.cs\*\*

```csharp

private void btnSearch\_Click(object sender, EventArgs e)

{

string searchTerm = txtSearch.Text.Trim();

List<Book> books = bookBLL.SearchBooks(searchTerm);

if (books.Count > 0)

{

dgvBooks.DataSource = books;

}

else

{

MessageBox.Show("No books found matching the search criteria.");

dgvBooks.DataSource = null;

}

}

```

---

# Step 9: Object-Oriented Programming Principles

### \*\*Encapsulation\*\*

- \*\*Definition\*\*: Bundling data with methods that operate on that data.

- \*\*Implementation\*\*: Using classes with private fields and public properties or methods.

- \*\*Example\*\*: The `Book` class encapsulates book-related data and behaviors.

### \*\*Abstraction\*\*

- \*\*Definition\*\*: Hiding complex implementation details.

- \*\*Implementation\*\*: Using interfaces or abstract classes.

- \*\*Example\*\*: The `IBookRepository` interface defines methods without exposing implementation.

### \*\*Inheritance\*\*

- \*\*Definition\*\*: Creating new classes from existing ones.

- \*\*Implementation\*\*: Using base and derived classes.

- \*\*Example\*\*: A `DigitalBook` class inheriting from `Book` with additional properties.

### \*\*Polymorphism\*\*

- \*\*Definition\*\*: Ability of different classes to be treated as instances of the same class through inheritance.

- \*\*Implementation\*\*: Overriding methods in derived classes.

- \*\*Example\*\*: Overriding a `DisplayInfo()` method in different book types.

---

## \*\*Step 10: Finalizing the Project Structure\*\*

### \*\*Testing\*\*

- \*\*Unit Testing\*\*: Test individual components using frameworks like NUnit or MSTest.

- \*\*Integration Testing\*\*: Ensure layers work together seamlessly.

- \*\*User Acceptance Testing\*\*: Validate with real-world scenarios and user feedback.

### \*\*Error Handling\*\*

- \*\*Try-Catch Blocks\*\*: Handle exceptions gracefully.

- \*\*Logging\*\*: Record errors for debugging using logging frameworks.

- \*\*User Feedback\*\*: Provide meaningful messages to guide the user.

### \*\*Documentation\*\*

- \*\*Code Comments\*\*: Explain complex logic and methods.

- \*\*User Manuals\*\*: Guide end-users on how to use the system.

- \*\*README Files\*\*: Provide setup and usage instructions.

### \*\*Deployment\*\*

- \*\*Setup Installer\*\*: Create an installer package using tools like InstallShield or WiX.

- \*\*Database Scripts\*\*: Provide scripts for database setup on target machines.

---

## \*\*Conclusion\*\*

By following this tutorial, you've developed a complete \*\*Library Management System\*\* desktop application using C#. The project covered:

- \*\*Software Architecture\*\*: Implemented a layered approach for separation of concerns.

- \*\*UML Diagrams\*\*: Visualized system structure and behavior with textual representations.

- \*\*Database Design\*\*: Created normalized tables with proper keys and relationships.

- \*\*C# Programming\*\*: Built the application using object-oriented programming principles.

- \*\*CRUD Operations\*\*: Enabled data manipulation functionalities for books and members.

- \*\*Search Functionality\*\*: Enhanced usability with search features.

---

## \*\*Appendix\*\*

### \*\*A. Complete SQL Scripts\*\*

```sql

-- SQL scripts for creating the database and tables

-- Refer to Step 5 for table creation scripts

-- Additional scripts for indexing (optional)

CREATE INDEX IDX\_Book\_Title ON Book(Title);

CREATE INDEX IDX\_Book\_Author ON Book(Author);

CREATE INDEX IDX\_Book\_ISBN ON Book(ISBN);

CREATE INDEX IDX\_Book\_Publisher ON Book(Publisher);

```

### \*\*B. UML Diagrams and ERD Representations\*\*

\*\*Note:\*\* While we cannot include actual images here, the textual representations provided in Step 3 and Step 4 can be used to create visual diagrams using UML diagramming tools like Lucidchart, Draw.io, or Microsoft Visio.

- \*\*Use Case Diagram\*\*: Illustrates the interactions between the Librarian and the system's use cases.

- \*\*Class Diagram\*\*: Shows the classes, their attributes, methods, and relationships.

- \*\*Sequence Diagram\*\*: Details the sequence of operations during processes like issuing a book.

- \*\*Activity Diagram\*\*: Depicts the workflow for activities such as returning a book.

- \*\*Entity-Relationship Diagram (ERD)\*\*: Represents the database entities, their attributes, and relationships.

\*\*Creating Diagrams:\*\*

- Use the textual representations as a guide.

- Input the classes, entities, and relationships into a diagramming tool.

- Label all components clearly for better understanding.

### \*\*C. Additional Resources\*\*

- \*\*C# Documentation\*\*: [Microsoft Docs - C#](https://docs.microsoft.com/en-us/dotnet/csharp/)

- \*\*SQL Server Documentation\*\*: [Microsoft Docs - SQL Server](https://docs.microsoft.com/en-us/sql/sql-server/)

- \*\*Data Binding in Windows Forms\*\*: [Data Binding and Windows Forms](https://docs.microsoft.com/en-us/dotnet/desktop/winforms/data-binding-and-windows-forms)

- \*\*UML Diagrams\*\*: [UML Diagram Tutorials](https://www.lucidchart.com/pages/uml-diagrams)

- \*\*Entity-Relationship Diagrams\*\*: [ERD Basics](https://www.visual-paradigm.com/guide/data-modeling/what-is-entity-relationship-diagram/)

---

\*\*Final Note:\*\* This document serves as a foundation for developing a C# desktop application with database integration. You can expand upon each section, add more features, and customize the application to suit specific needs. Remember to follow best practices and continually test your application throughout the development process.