|  |  |
| --- | --- |
| cid:3461564935_486740 | Faculty of Natural and Applied Sciences  Computer Science Department |

|  |
| --- |
| **CSC 323: Object-Oriented Design**  **Final Project**  **Library Management System - Report** |

**Project Description**

|  |
| --- |
| The Library Management System represents a sophisticated software solution crafted in C# with an integrated Windows Forms application, designed to revolutionize the operations of libraries and optimize resource management. At its core is the strategically implemented Library class, designed as a singleton to ensure a singular, consistent instance overseeing the entire library collection, including books and members. The adoption of the Singleton pattern not only guarantees a unified source of truth for library data but also establishes centralized control over essential operations, ensuring that there is only one instance of the Library class throughout the application's lifecycle.  The Book class serves as a foundational element for representing individual books, featuring a meticulously implemented builder pattern (BookBuilder) for the dynamic instantiation of book objects with optional parameters. This design choice enhances code readability and flexibility, allowing for the instantiation of books with varying attributes in a clear and concise manner. Simultaneously, the Member class plays a crucial role in representing library members and implements the IMember interface, ensuring a standardized set of member-related functionalities. This interface-driven approach promotes abstraction, modularity, and code consistency, aligning with established best practices in software design.  An innovative addition to the system is the LibraryFacade class, strategically positioned as a simplified interface to encapsulate the intricacies of complex book borrowing and returning processes. This abstraction shields users from underlying complexities, providing a more intuitive and user-friendly interaction experience. The seamless integration of a Windows Forms application (LibraryForm.cs) elevates user interactions by offering a graphical interface for library staff. This interface empowers staff members to effortlessly add and remove books, manage members, and facilitate borrowing and returning processes in an intuitive and visually appealing environment.  The project upholds the highest standards of code quality, emphasizing best practices such as clear separation of concerns, encapsulation of functionalities, and strategic use of comments for enhanced code comprehension. A meticulous code refactoring process ensures the delivery of a clean, maintainable, and future-proof codebase, adhering to industry standards and promoting a collaborative development environment. The systematic adoption of the Singleton pattern in the Library class exemplifies the project's commitment to sound architectural principles, fostering a cohesive and scalable foundation for library management.  Looking ahead, the Library Management System stands as a resilient and scalable solution, well-positioned for future enhancements. Potential features in the pipeline include advanced search functionalities, intricate book categorization systems, and the seamless integration of diverse media types beyond conventional books. In summary, this project serves as a testament to the effective implementation of object-oriented principles, design patterns, and user interface integration to deliver an organized, scalable, and exceptionally user-friendly library management experience. |

**Class Diagram (UML)**

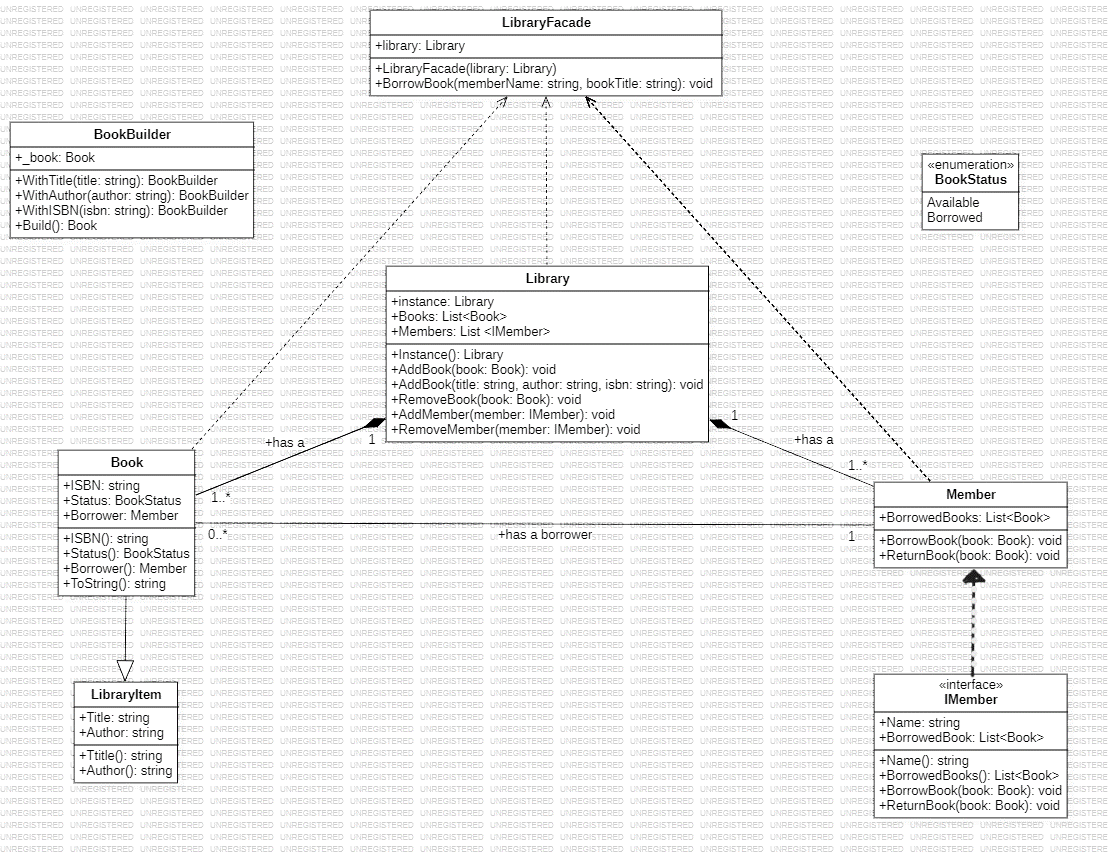


Figure 1. Class Diagram

**Source Code (C#)**

{

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using System.Windows.Forms;

using LibraryManagementSystem;

static class Program

{

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new Form1());

}

}

// LibraryManagementSystem namespace

namespace LibraryManagementSystem

{

// Enum representing the status of a book

public enum BookStatus

{

Available, // Book is available for borrowing

Borrowed // Book is currently borrowed by a member

}

// Abstract class representing common properties of library items

public abstract class LibraryItem

{

public string Title { get; set; } // Title of the library item

public string Author { get; set; } // Author of the library item

}

// Concrete class representing a book, inheriting from LibraryItem

public class Book : LibraryItem

{

public string ISBN { get; set; } // ISBN (International Standard Book Number) of the book

public BookStatus Status { get; set; } // Status of the book (Available or Borrowed)

public Member Borrower { get; set; } // Member who borrowed the book

// Constructors

public Book(string title, string author, string isbn)

{

Title = title;

Author = author;

ISBN = isbn;

Status = BookStatus.Available;

Borrower = null;

}

public Book(string title, string author) : this(title, author, "Unknown") { }

// Overriding method to customize the string representation of the book

public override string ToString()

{

return $"{Title} by {Author} / {ISBN}, {Status}";

}

}

// Builder class for creating Book objects

public class BookBuilder

{

private Book \_book;

public BookBuilder()

{

\_book = new Book(null, null);

}

public BookBuilder WithTitle(string title)

{

\_book.Title = title;

return this;

}

public BookBuilder WithAuthor(string author)

{

\_book.Author = author;

return this;

}

public BookBuilder WithISBN(string isbn)

{

\_book.ISBN = isbn;

return this;

}

public Book Build()

{

return \_book;

}

}

// Interface defining the contract for library members

public interface IMember

{

string Name { get; set; } // Name of the member

List<Book> BorrowedBooks { get; set; } // List of books borrowed by the member

void BorrowBook(Book book); // Method to borrow a book

void ReturnBook(Book book); // Method to return a borrowed book

}

// Concrete class representing a library member, implementing IMember interface

public class Member : IMember

{

public string Name { get; set; } // Name of the member

public List<Book> BorrowedBooks { get; set; } // List of books borrowed by the member

// Constructor

public Member(string name)

{

Name = name;

BorrowedBooks = new List<Book>();

}

// Override ToString to customize the display of the member

public override string ToString() => Name;

// Methods

public void BorrowBook(Book book)

{

// Check if the book is available

if (book.Status == BookStatus.Available)

{

MessageBox.Show($"Borrowing book: {book.Title}, Status: {book.Status}");

book.Status = BookStatus.Borrowed;

book.Borrower = this;

BorrowedBooks.Add(book);

MessageBox.Show($"Book borrowed successfully. New status: {book.Status}");

}

else

{

MessageBox.Show($"{book.Title} is not available for borrowing. Come back later!");

}

}

public void ReturnBook(Book book)

{

// Check if the member borrowed the book and it's marked as borrowed

if (book.Status == BookStatus.Borrowed && book.Borrower == this)

{

book.Status = BookStatus.Available;

book.Borrower = null;

BorrowedBooks.Remove(book);

}

}

}

// Library class with Singleton pattern

public class Library

{

private static Library \_instance; // Singleton instance of the Library class

// Singleton property to access the instance

public static Library Instance => \_instance ?? (\_instance = new Library());

public List<Book> Books { get; set; } // List of books in the library

public List<IMember> Members { get; set; } // List of library members

// Private constructor to enforce Singleton pattern

private Library()

{

Books = new List<Book>();

Members = new List<IMember>();

}

// Methods for adding/removing books and members

public void AddBook(Book book) => Books.Add(book);

public void AddBook(string title, string author, string isbn)

{

var book = new BookBuilder()

.WithTitle(title)

.WithAuthor(author)

.WithISBN(isbn)

.Build();

Books.Add(book);

}

public void RemoveBook(Book book)

{

// Remove the book only if it's not currently borrowed

if (book.Status != BookStatus.Borrowed)

Books.Remove(book);

}

public void AddMember(IMember member) => Members.Add(member);

public void RemoveMember(IMember member) => Members.Remove(member);

}

// Concrete class representing a library facade

public class LibraryFacade

{

private Library library;

public LibraryFacade(Library library)

{

this.library = library;

}

public void BorrowBook(string memberName, string bookTitle)

{

Member member = (Member)library.Members.Find(m => m.Name == memberName);

Book book = library.Books.Find(b => b.Title == bookTitle);

if (member != null && book != null)

{

member.BorrowBook(book);

}

}

}

}

}

Figure 2. Program.cs

{

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

{

// Get selected book and member from lists

Book book = (Book)lstBooks.SelectedItem;

Member member = (Member)lstMembers.SelectedItem;

// Check if book and member are not null

if (book != null && member != null)

{

//MessageBox.Show($"Selected Book: {book.Title}, Selected Member: {member.Name}");

LibraryFacade facade = new LibraryFacade(library);

facade.BorrowBook(member.Name, book.Title);

// Update book list display

UpdateBookList();

UpdateMemberList();

}

else

{

MessageBox.Show("Selected book or member is null.");

}

}

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

{

// Get selected book and member from lists

Book book = (Book)lstBooks.SelectedItem;

Member member = (Member)lstMembers.SelectedItem;

// Check if book and member are not null

if (book != null && member != null)

{

// Member returns book

member.ReturnBook(book);

// Update book list display

UpdateBookList();

UpdateMemberList();

}

}

private void UpdateBookList()

{

lstBooks.DataSource = null;

lstBooks.DataSource = library.Books;

lstBooks.DisplayMember = "DisplayText"; // DisplayText property should be added to the Book class

}

private void UpdateMemberList()

{

lstMembers.DataSource = null;

lstMembers.DataSource = library.Members;

lstMembers.DisplayMember = "Name"; // Assuming Member class has a Name property

}

private void lstBooks\_SelectedIndexChanged(object sender, EventArgs e)

{

// Check if an item is selected

if (lstBooks.SelectedItem != null)

{

Book selectedBook = (Book)lstBooks.SelectedItem;

txtTitle.Text = selectedBook.Title;

txtAuthor.Text = selectedBook.Author;

txtISBN.Text = selectedBook.ISBN;

}

else

{

txtTitle.Text = string.Empty;

txtAuthor.Text = string.Empty;

txtISBN.Text = string.Empty;

}

}

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

{

StringBuilder summary = new StringBuilder();

foreach (Member member in library.Members)

{

summary.AppendLine($"{member.Name}:");

if (member.BorrowedBooks.Any())

{

foreach (Book book in member.BorrowedBooks)

{

summary.AppendLine($"\t{book.Title}");

}

}

else

{

summary.AppendLine("\tN/A");

}

}

MessageBox.Show(summary.ToString(), "Member Borrowing Summary");

}

}

}}

Figure 3. LibraryForm.cs (Windows Form)

**Descriptions of how and where you have used key concepts**

|  |  |
| --- | --- |
| **Concept** | **Description** |
| Abstraction (abstract classes and/or interfaces) | The IMember interface indeed defines a contract that concrete classes, like Member, must implement. It encapsulates methods (BorrowBook, ReturnBook) and properties (Name, BorrowedBooks), enforcing a standardized set of functionalities for classes representing library members. |
| Concrete classes | The Book and Member classes are correctly identified as concrete implementations, extending abstract structures (LibraryItem and IMember). They provide specific functionalities for library items and members, contributing to the overall structure and functionality of the system. |
| Namespaces | The use of namespaces, specifically the LibraryManagementSystem namespace, is appropriately highlighted. It promotes organization and helps avoid naming conflicts with classes from other projects. |
| Methods | The methods, such as BorrowBook and ReturnBook in IMember and Member, are correctly identified as encapsulating behaviors related to borrowing and returning books. This promotes modularity and reusability within the code. |
| Properties | We can accurately recognize the use of properties like Title and Author in LibraryItem to encapsulate attributes of library items. Additionally, the properties in IMember represent member attributes, providing controlled access. |
| Overriding of methods and/or properties | The overridden ToString method in the Book class is appropriately mentioned. This demonstrates customization for the string representation of a book, enhancing display flexibility. |
| Constructor overloading | The use of constructor overloading in the Book class is correctly identified. It provides flexibility in instantiation by offering constructors with varying parameters, such as title, author, and ISBN. |
| Enumeration (Enum) | The BookStatus enum is appropriately highlighted as defining possible states (Available and Borrowed) for a book. This provides a clear and standardized representation of book status. |
| Collections (e.g., Lists) | The use of Lists (Books and Members) in the Library class, managing dynamic collections. This allows easy addition, removal, and retrieval of books and members in a structured manner. |
| Loops (e.g., foreach) | The use of foreach loops in LibraryForm.cs iterating through the library.Books and library.Members collections is correctly identified. This facilitates the update of book and member displays in the user interface. |

|  |  |
| --- | --- |
| **SOLID Principle** | **Description of where the principle is realized and for what purpose** |
| 1) Single Responsibility | The Library class adheres to the Single Responsibility Principle, focusing solely on library-related operations like adding/removing books and members. By avoiding unnecessary responsibilities, the code remains focused, maintainable, and comprehensible. |
| 2) Open-Closed | The Book class upholds the Open-Closed Principle, allowing extension without modification. This design enables clients to introduce new book types seamlessly, promoting code stability and scalability without altering existing code. |
| 3) Liskov Substitution | Liskov Substitution is exemplified through the IMember interface. Any class implementing it, such as Member, can be substituted interchangeably, ensuring consistent behavior throughout the library system and maintaining compatibility with client code. |
| 4) Interface Segregation | Interface Segregation is applied in IMember, offering a specific set of methods relevant only to library members. This ensures that implementing classes, like Member, aren't burdened with unnecessary functionalities, promoting a lean and focused interface. |
| 5) Dependency Inversion | Dependency Inversion is manifested in the LibraryFacade, which depends on abstractions (Library and Member). This design choice allows flexibility in substituting concrete implementations, aligning with the principle of depending on abstractions rather than concretions for increased adaptability. |

|  |  |
| --- | --- |
| **GOF Design Pattern** | **Description of where the pattern is realized and for what purpose** |
| 1) Singleton Pattern | Implemented in the Library class, the Singleton Pattern ensures a single, consistent instance for the entire application. This guarantees centralized control over the library's state, prevents multiple instances, and maintains data management consistency across the system. |
| 2)Builder Pattern | The BookBuilder class employs the Builder Pattern (indirectly) to construct Book objects with optional parameters, promoting code readability and flexibility. This design allows for the creation of books with varying attributes in a concise and easily understandable manner. |
| 3)Façade Pattern | Acting as a Facade, the LibraryFacade provides a simplified interface for borrowing books. It encapsulates complex interactions between members and books, offering a straightforward method (BorrowBook) for users while abstracting internal complexities, enhancing user experience and code maintainability. |

**User Interface (Windows Forms)**

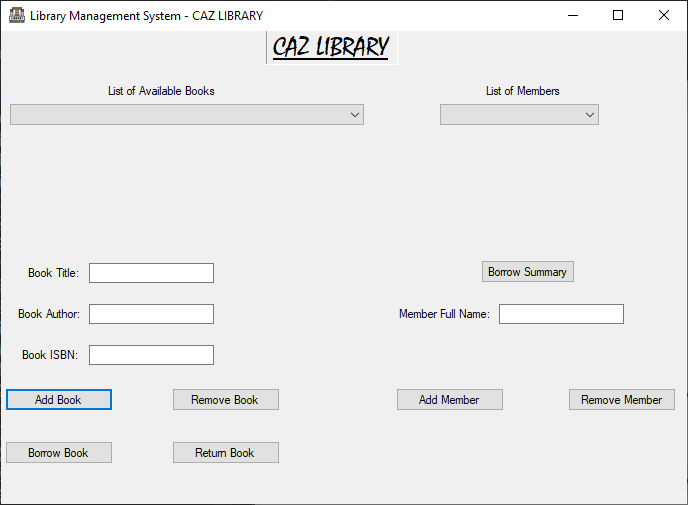


Figure 4. Main User Interface

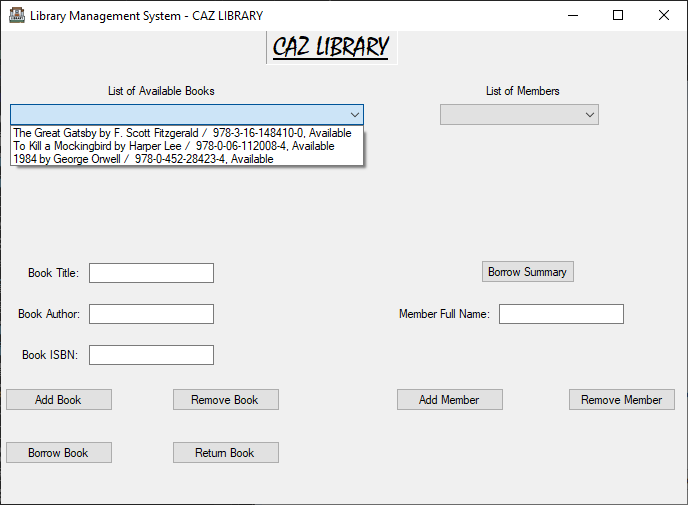


Figure 5. User Interface Showing initial list of books.

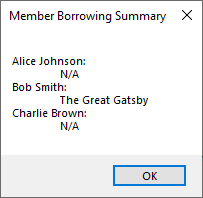


Figure 6. Pop Up window showing the Member Borrowing Summary

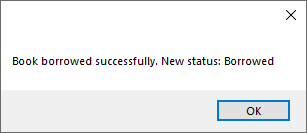


Figure 7. Pop Up window appearing when a borrow process is done

|  |
| --- |
| **Brief Description of UI** |
| CAZ LIBRARYTop of Form  The Library Management System's user interface (UI) enables users to seamlessly manage books and members. Staff can add or remove books and members with dedicated buttons. Selecting a book or member from lists displays detailed information. Borrowing or returning books is facilitated by intuitive buttons, interacting with the LibraryFacade. The "Summary" button provides a quick overview of each member's borrowed book. Error messages ensure effective feedback. The UI's functionality promotes efficient library operations, enhancing user experience. |

* **We kindly recommend running the application, to discover all the provided features.**