Library Management System

F# Desktop Application with Windows Forms

1. Project Overview

A desktop application built with `F#` and `Windows Forms` that manages library operations including book inventory, borrowing, and returning books. The system provides a complete solution for small to medium-sized library management needs.

Purpose & Scope

This application serves as a comprehensive library management solution, enabling librarians to efficiently track book inventory, manage borrowing operations, and maintain accurate records of all library transactions.

Key Features

- ✓ Add books with validation
- ✓ Search functionality
- ✓ Borrow/return tracking with due dates
- ✓ Persistent data storage
- ✓ Real-time status updates
- ✓ Error handling and user feedback

User Interface Components

The application features a tabbed interface with four main sections:

Tab	Description
`View All Books`	Display all books in catalog with details (Title, Author, ISBN, Status, Borrower, Due Date)
`Add Book`	Add new books with title, author, and ISBN validation

Tab	Description
Search	Search books by title using case-insensitive matching
Borrow/ Return	Manage book borrowing and returning operations

2. System Architecture

The Library Management System follows a layered architecture pattern, separating concerns into distinct layers for maintainability and testability. The system is designed using functional programming principles inherent to F#.

2.1 Architecture Layers

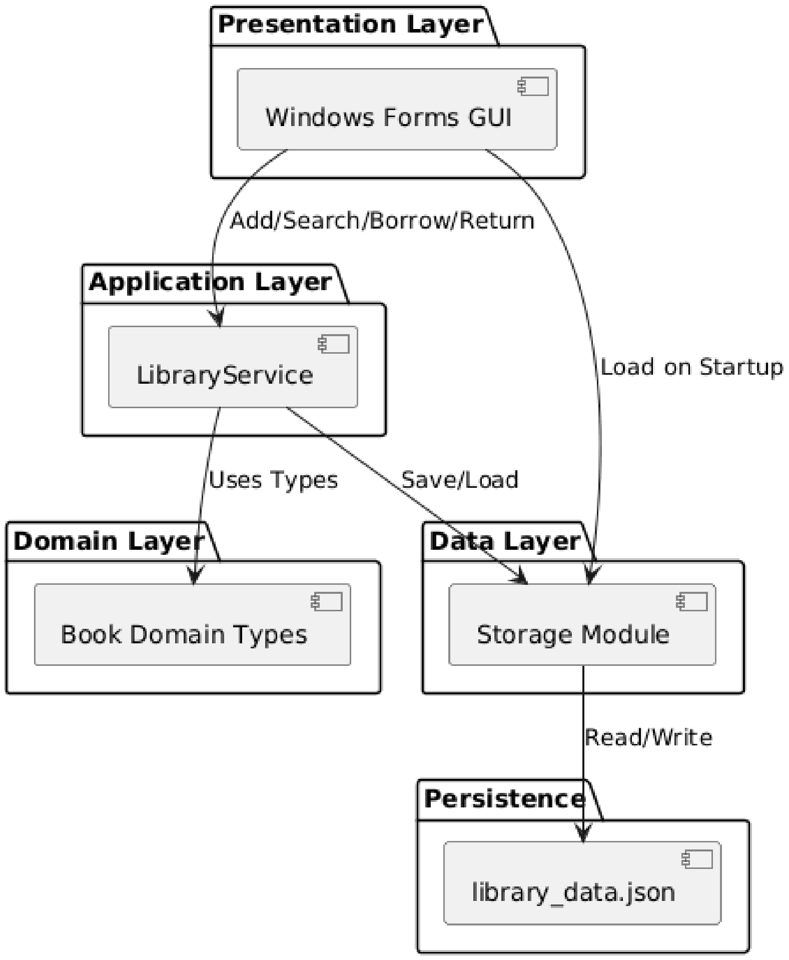

Figure 1: High-level architecture showing layer dependencies

2.2 Layer Descriptions

Presentation Layer (UI)

Windows Forms-based graphical interface providing user interactions through a tabbed layout with Add Book, Search, View All Books, and Borrow/Return tabs.

Application Layer (Business Logic)

Contains `LibraryService` with core operations: addBook(), searchByTitle(), searchAvailable(), borrowBook(), and returnBook().

Domain Layer

Defines core domain types including Book, BookStatus (Available/Borrowed), BookId, ISBN (validated smart constructor), MemberId, and LibraryError types.

Data Layer (Storage)

JSON file-based persistence using `library_data.json` . Implements auto-save after operations and data loading on startup.

2.3 Detailed Component Diagram

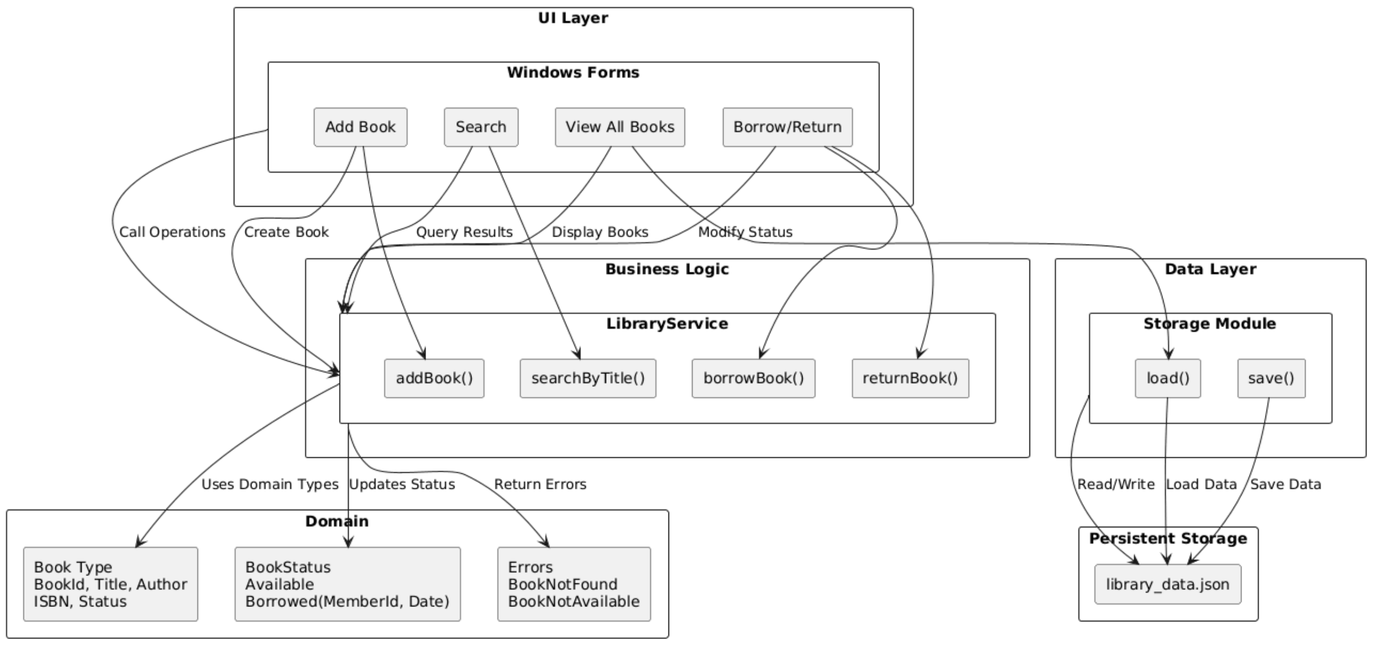

Figure 2: Detailed block diagram showing component interactions and data flow

2.4 Domain Model

Type	Description	Properties
`Book`	Core domain entity	Id, Title, Author, ISBN, Status
`BookStatus`	Discriminated union	Available | Borrowed(MemberId, DueDate)
`ISBN`	Validated value type	Smart constructor with validation
`LibraryError`	Error handling type	BookNotFound | BookNotAvailable | InvalidData | StorageError

2.5 Business Logic Functions

Function	Purpose
`addBook()`	Creates and validates new book entries with ISBN validation
`searchByTitle()`	Case-insensitive search by book title
`searchAvailable()`	Filters and returns only available books
`borrowBook()`	Marks book as borrowed with member name and 14-day due date
`returnBook()`	Returns book to available status

3. Testing Strategy

The testing approach utilizes F# pattern matching for validation and verification, implemented within a dedicated `Tests.fs` module. The framework provides clear success (✅) and failure (❌) indicators for test results.

3.1 Test Framework

> **Approach:** Simple console-based testing with print statements
> **Validation:** Pattern matching for type-safe assertions
> **Output:** Success (✅) and failure (❌) indicators

3.2 Implemented Tests

Status	Test Name	Description
✓ Implemented	`Add Book Test`	Verifies book creation functionality
✓ Implemented	`ISBN Validation Test`	Validates ISBN empty field rejection

3.3 Test Coverage Analysis

✓ Currently Tested

- ✓ ISBN validation (rejects empty values)
- ✓ Book addition to catalog

○ Pending Implementation

- ○ Search functionality testing
- ○ Borrow/return operations
- ○ Storage persistence verification
- ○ Member validation
- ○ Due date handling