Library Management System - Design Patterns Implementation Report

Introduction

This comprehensive report details the implementation of various design patterns and SOLID principles in the Library Management System. The system was designed to efficiently manage library resources, handle user interactions, and provide a flexible, maintainable architecture.

Design Patterns Implementation

1. Observer Pattern

Purpose: To establish a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. In our Library Management System, this pattern notifies users about library events such as document additions, borrowings, and returns.

Implementation Details:

- Subject Interface: `ILibrarySubject` defines the contract for objects that can be observed
- Observer Interface: `ILibraryObserver` defines the contract for objects that observe subjects
- Concrete Subject: `LibraryManager` maintains a list of observers and notifies them of state changes
- Concrete Observer: `User` class implements the observer interface to receive notifications
- Event Encapsulation: `LibraryEvent` class contains event information (type, document, user)
- Event Types: `EventType` enum defines different notification categories (DocumentAdded, DocumentBorrowed, DocumentReturned)

Complete Implementation:

```
// Subject interface
public interface ILibrarySubject
{
    void RegisterObserver(ILibraryObserver observer);
    void RemoveObserver(ILibraryObserver observer);
```

```
    void NotifyObservers(LibraryEvent libraryEvent);
}

// Observer interface
public interface ILibraryObserver
{
    void Update(LibraryEvent libraryEvent);
}

// Concrete Subject
public class LibraryManager : ILibrarySubject
{
    private List<ILibraryObserver> _observers = new
List<ILibraryObserver>();
    private List<Document> _documents = new List<Document>();

    public void RegisterObserver(ILibraryObserver observer)
    {
        if (!_observers.Contains(observer))
        {
            _observers.Add(observer);
        }
    }

    public void RemoveObserver(ILibraryObserver observer)
    {
        _observers.Remove(observer);
    }

    public void NotifyObservers(LibraryEvent libraryEvent)
    {
        foreach (var observer in _observers)
        {
            observer.Update(libraryEvent);
        }
    }

    // Library operations that trigger notifications
    public void AddDocument(Document document)
    {
        _documents.Add(document);
        NotifyObservers(new LibraryEvent(EventType.DocumentAdded,
document));
    }

    public void BorrowDocument(Document document, User user)
```

```
    {
        if (document.IsAvailable)
        {
            document.IsAvailable = false;
            NotifyObservers(new
LibraryEvent(EventType.DocumentBorrowed, document, user));
        }
    }

    public void ReturnDocument(Document document, User user)
    {
        if (!document.IsAvailable)
        {
            document.IsAvailable = true;
            NotifyObservers(new
LibraryEvent(EventType.DocumentReturned, document, user));
        }
    }
}

// Concrete Observer
public class User : ILibraryObserver
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public List<Document> InterestedDocuments { get; set; } = new
List<Document>();

    public User(int id, string name, string email)
    {
        Id = id;
        Name = name;
        Email = email;
    }

    public void AddInterest(Document document)
    {
        if (!InterestedDocuments.Contains(document))
        {
            InterestedDocuments.Add(document);
        }
    }

    public void Update(LibraryEvent libraryEvent)
    {
```

```
93.         // Only process notifications for documents the user is
    interested in
94.         if (InterestedDocuments.Contains(libraryEvent.Document) ||
    libraryEvent.User == this)
95.         {
96.             Console.WriteLine($"Notification for {Name}:
    {GetNotificationMessage(libraryEvent)}");
97.             // In a real application, this could send an email or push
    notification
98.         }
99.     }
100.
101.     private string GetNotificationMessage(LibraryEvent
    libraryEvent)
102.     {
103.         return libraryEvent.EventType switch
104.         {
105.             EventType.DocumentAdded => $"New document added:
    {libraryEvent.Document.Title}",
106.             EventType.DocumentBorrowed => $"Document borrowed:
    {libraryEvent.Document.Title} by {libraryEvent.User.Name}",
107.             EventType.DocumentReturned => $"Document returned:
    {libraryEvent.Document.Title} by {libraryEvent.User.Name}",
108.             _ => "Unknown event"
109.         };
110.     }
111. }
112.
113. // Event class to encapsulate event information
114. public class LibraryEvent
115. {
116.     public EventType EventType { get; }
117.     public Document Document { get; }
118.     public User? User { get; }
119.
120.     public LibraryEvent(EventType eventType, Document document,
    User? user = null)
121.     {
122.         EventType = eventType;
123.         Document = document;
124.         User = user;
125.     }
126. }
127.
128. // Event types enum
```

```
public enum EventType
{
    DocumentAdded,
    DocumentBorrowed,
    DocumentReturned
}
```

Benefits:

1. Loose Coupling: The library system (subject) doesn't need to know the specifics of its observers, only that they implement the `ILibraryObserver` interface.

2. Dynamic Registration: Users can register or unregister for notifications at runtime, allowing for flexible notification preferences.

3. Extensibility: New event types can be added to the `EventType` enum without modifying existing code, following the Open/Closed Principle.

4. Targeted Notifications: Users can specify which documents they're interested in, receiving only relevant notifications.

5. Separation of Concerns: The notification logic is separated from the core library operations, making the system more maintainable.

2. Factory Method Pattern

Purpose: To define an interface for creating objects, but allow subclasses to decide which classes to instantiate. The Factory Method Pattern lets a class defer instantiation to subclasses, promoting loose coupling and adherence to the dependency inversion principle.

Implementation Details:

- Abstract Product: `Document` class serves as the base for all document types
- Concrete Products: `Book`, `Magazine`, and `Newspaper` classes implement specific document types
- Abstract Creator: `DocumentFactory` abstract class defines the factory method interface
- Concrete Creators: `BookFactory`, `MagazineFactory`, and `NewspaperFactory` implement the creation logic

Complete Implementation:

```
// Abstract Product
public abstract class Document
{
```

```
    public int Id { get; set; }
    public string Title { get; set; }
    public string Author { get; set; }
    public DateTime PublicationDate { get; set; }
    public bool IsAvailable { get; set; } = true;

    public Document(string title, string author, DateTime publicationDate)
    {
        Title = title;
        Author = author;
        PublicationDate = publicationDate;
    }

    public abstract string GetDocumentType();
}

// Concrete Products
public class Book : Document
{
    public string ISBN { get; set; }
    public int Pages { get; set; }

    public Book(string title, string author, DateTime publicationDate, string isbn, int pages)
        : base(title, author, publicationDate)
    {
        ISBN = isbn;
        Pages = pages;
    }

    public override string GetDocumentType() => "Book";
}

public class Magazine : Document
{
    public string IssueNumber { get; set; }
    public string Publisher { get; set; }

    public Magazine(string title, string author, DateTime publicationDate, string issueNumber, string publisher)
        : base(title, author, publicationDate)
    {
        IssueNumber = issueNumber;
        Publisher = publisher;
    }
```

```
47.
48.    public override string GetDocumentType() => "Magazine";
49.}
50.
51.public class Newspaper : Document
52.{
53.    public string Edition { get; set; }
54.
55.    public Newspaper(string title, string author, DateTime
   publicationDate, string edition)
56.        : base(title, author, publicationDate)
57.    {
58.        Edition = edition;
59.    }
60.
61.    public override string GetDocumentType() => "Newspaper";
62.}
63.
64.// Abstract Creator
65.public abstract class DocumentFactory
66.{
67.    public abstract Document CreateDocument();
68.
69.    // Factory Method that creates and returns a document
70.    public Document GetDocument()
71.    {
72.        Document document = CreateDocument();
73.        return document;
74.    }
75.}
76.
77.// Concrete Creators
78.public class BookFactory : DocumentFactory
79.{
80.    private string _title;
81.    private string _author;
82.    private DateTime _publicationDate;
83.    private string _isbn;
84.    private int _pages;
85.
86.    public BookFactory(string title, string author, DateTime
   publicationDate, string isbn, int pages)
87.    {
88.        _title = title;
89.        _author = author;
90.        _publicationDate = publicationDate;
```

```
        _isbn = isbn;
        _pages = pages;
    }

    public override Document CreateDocument()
    {
        return new Book(_title, _author, _publicationDate, _isbn,
    _pages);
    }
}

    public class MagazineFactory : DocumentFactory
    {
        private string _title;
        private string _author;
        private DateTime _publicationDate;
        private string _issueNumber;
        private string _publisher;

        public MagazineFactory(string title, string author, DateTime
    publicationDate, string issueNumber, string publisher)
        {
            _title = title;
            _author = author;
            _publicationDate = publicationDate;
            _issueNumber = issueNumber;
            _publisher = publisher;
        }

        public override Document CreateDocument()
        {
            return new Magazine(_title, _author, _publicationDate,
    _issueNumber, _publisher);
        }
    }

    public class NewspaperFactory : DocumentFactory
    {
        private string _title;
        private string _author;
        private DateTime _publicationDate;
        private string _edition;

        public NewspaperFactory(string title, string author, DateTime
    publicationDate, string edition)
        {
```

```
        _title = title;
        _author = author;
        _publicationDate = publicationDate;
        _edition = edition;
    }

    public override Document CreateDocument()
    {
        return new Newspaper(_title, _author, _publicationDate,
_edition);
    }
}
```

Benefits:

1. **Encapsulation of Creation Logic: The creation logic for each document type is encapsulated in its respective factory class.**
2. **Extensibility: New document types can be added by creating new concrete product classes and corresponding factory classes without modifying existing code.**
3. **Consistency: The factory method ensures that all documents are created consistently, with all required properties properly initialized.**
4. **Dependency Inversion: Client code depends on abstractions (Document and DocumentFactory) rather than concrete implementations.**
5. **Simplified Client Code: Client code only needs to work with the abstract factory and product interfaces, not the concrete implementations.**

3. Strategy Pattern

Purpose: To define a family of algorithms, encapsulate each one, and make them interchangeable. The Strategy Pattern lets the algorithm vary independently from clients that use it, promoting flexibility and reusability.

Implementation Details:

- **Strategy Interface: `ILoanFeeStrategy` defines the contract for fee calculation algorithms**
- **Concrete Strategies: `BookLoanFeeStrategy` and `MagazineLoanFeeStrategy` implement specific fee calculation algorithms**
- **Context: Client code that uses the strategies to calculate fees for different document types**

Complete Implementation:

```csharp
// Strategy interface
public interface ILoanFeeStrategy
{
 decimal CalculateFee(Document document, int daysLoaned);
}

// Concrete Strategies
public class BookLoanFeeStrategy : ILoanFeeStrategy
{
 private const decimal BaseRate = 0.10m; // $0.10 per day
 private const int StandardLoanPeriod = 14; // 14 days standard loan period
 private const decimal LateFeeMultiplier = 1.5m; // 50% more for late
returns

   1. public decimal CalculateFee(Document document, int daysLoaned)
   2. {
   3.     if (document is not Book)
   4.         throw new ArgumentException("This strategy can only be used
      with books");
   5.
   6.     // Standard fee calculation
   7.     decimal fee = daysLoaned * BaseRate;
   8.
   9.     // Apply late fee if applicable
   10.    if (daysLoaned > StandardLoanPeriod)
   11.    {
   12.        int lateDays = daysLoaned - StandardLoanPeriod;
   13.        fee += lateDays * BaseRate * LateFeeMultiplier;
   14.    }
   15.
   16.    return Math.Round(fee, 2);
   17.}

}

public class MagazineLoanFeeStrategy : ILoanFeeStrategy
{
 private const decimal BaseRate = 0.20m; // $0.20 per day
 private const int StandardLoanPeriod = 7; // 7 days standard loan period
 private const decimal LateFeeMultiplier = 2.0m; // Double for late returns

   1. public decimal CalculateFee(Document document, int daysLoaned)
   2. {
   3.     if (document is not Magazine)
   4.         throw new ArgumentException("This strategy can only be used
      with magazines");
   5.
   6.     // Standard fee calculation
```

```
    decimal fee = daysLoaned * BaseRate;

    // Apply late fee if applicable
    if (daysLoaned > StandardLoanPeriod)
    {
        int lateDays = daysLoaned - StandardLoanPeriod;
        fee += lateDays * BaseRate * LateFeeMultiplier;
    }

    return Math.Round(fee, 2);
}

}

// Context class that uses the strategies
public class LoanFeeCalculator
{
 private ILoanFeeStrategy _strategy;

public LoanFeeCalculator(ILoanFeeStrategy strategy)
{
    _strategy = strategy;
}

public void SetStrategy(ILoanFeeStrategy strategy)
{
    _strategy = strategy;
}

public decimal CalculateFee(Document document, int daysLoaned)
{
    return _strategy.CalculateFee(document, daysLoaned);
}

}


    }
}
```

- **Factory Pattern**: Client code depends on the abstract `Document` class and `DocumentFactory` class, not concrete implementations.

```csharp
// Client code depends on abstractions
DocumentFactory factory = new BookFactory("Design Patterns", "Gang of Four", new DateTime(1994, 10, 21), "978-0201633610", 416);
Document document = factory.GetDocument();
```

Benefits:
- Reduced coupling between components
- Improved code maintainability
- Enhanced system flexibility and testability

Conclusion

The Library Management System demonstrates effective use of design patterns and SOLID principles to create a maintainable, extensible, and robust system.

Design Patterns Summary

- **Observer Pattern**: Enables efficient notification handling between the library system and users, allowing for loose coupling and dynamic registration of observers.

- **Factory Method Pattern**: Manages document creation, encapsulating creation logic and promoting extensibility for new document types.

- **Strategy Pattern**: Provides flexible fee calculation algorithms that can be selected and switched at runtime.

- **Singleton Pattern**: Ensures efficient resource management for database connections.

SOLID Principles Summary

- **Single Responsibility Principle**: Each class has a well-defined, focused purpose.

- **Open/Closed Principle**: The system is designed to be extended without modifying existing code.

- **Liskov Substitution Principle**: Subclasses can be used in place of their parent classes without affecting program correctness.

- **Interface Segregation Principle**: Interfaces are focused and minimal, containing only necessary methods.

- **Dependency Inversion Principle**: High-level modules depend on abstractions, not concrete implementations.

These patterns and principles, when applied together, result in a well-structured, maintainable, and extensible codebase that can evolve to meet changing requirements with minimal risk and effort.

Implementation Challenges and Solutions

During the implementation of the Library Management System, several challenges were encountered and addressed:

```
// Usage example
public class Program
{
 static void Main(string[] args)
 {
 // Create documents
 Document book = new Book("Design Patterns", "Gang of Four", new
DateTime(1994, 10, 21), "978-0201633610", 416);
 Document magazine = new Magazine("Scientific American", "Various Authors",
DateTime.Now.AddMonths(-1), "Vol 326, No 5", "Springer Nature");
```

```
    // Create fee calculator with initial strategy
    LoanFeeCalculator calculator = new LoanFeeCalculator(new
    BookLoanFeeStrategy());

    // Calculate fee for book (14 days)
    decimal bookFee = calculator.CalculateFee(book, 14);
    Console.WriteLine($"Fee for borrowing '{book.Title}' for 14 days:
    ${bookFee}");

    // Calculate fee for book (21 days - with late fee)
    decimal bookLateFee = calculator.CalculateFee(book, 21);
    Console.WriteLine($"Fee for borrowing '{book.Title}' for 21 days
    (includes late fee): ${bookLateFee}");

    // Switch strategy for magazine
    calculator.SetStrategy(new MagazineLoanFeeStrategy());

    // Calculate fee for magazine (7 days)
    decimal magazineFee = calculator.CalculateFee(magazine, 7);
    Console.WriteLine($"Fee for borrowing '{magazine.Title}' for 7
    days: ${magazineFee}");

    // Calculate fee for magazine (10 days - with late fee)
    decimal magazineLateFee = calculator.CalculateFee(magazine, 10);
    Console.WriteLine($"Fee for borrowing '{magazine.Title}' for 10
    days (includes late fee): ${magazineLateFee}");
}
```

```

**Benefits**:

```

1. 1. **Algorithm Encapsulation**: Each fee calculation algorithm is encapsulated in its own class, making them easy to understand and maintain.
2.
3. 2. **Runtime Strategy Selection**: The system can switch between different fee calculation strategies at runtime based on document type or other criteria.
4.
5. 3. **Easy Extension**: New fee calculation strategies can be added by implementing the `ILoanFeeStrategy` interface without modifying existing code.
6.
7. 4. **Single Responsibility**: Each strategy class has a single responsibility - calculating fees for a specific document type.
8.
9. 5. **Testability**: Each strategy can be tested independently, making the system more testable.
10.
11. ### 4. Singleton Pattern
12.
13. **Purpose**: To ensure a class has only one instance and provide a global point of access to it. In our Library Management System, this pattern is used for the database connection to ensure only one connection is maintained.
14.
15. **Implementation Details**:
16.
17. - **Private Constructor**: Prevents instantiation from outside the class
18. - **Private Static Instance**: Holds the single instance of the class
19. - **Public Static Access Method**: Provides global access to the instance
20. - **Thread Safety**: Implemented using double-check locking pattern
21.
22. **Complete Implementation**:
23.
24. ```csharp
25. public sealed class DatabaseConnection
26. {
27.   // Private static instance variable
28.   private static DatabaseConnection? _instance;
29.
30.   // Lock object for thread safety
31.   private static readonly object _lock = new object();
32.
33.   // Connection string
34.   private string _connectionString;
35.
36.   // Connection string
37.   private string _connectionString;

```
38.
39.     // Private constructor to prevent instantiation from outside
40.     private DatabaseConnection()
41.     {
42.         _connectionString = "Server=localhost;Database=LibraryDB;User
    Id=admin;Password=password;";
43.         Console.WriteLine("Database connection initialized.");
44.     }
45.
46.     // Public method to get the instance
47.     public static DatabaseConnection Instance
48.     {
49.         get
50.         {
51.             // Double-check locking pattern for thread safety
52.             if (_instance == null)
53.             {
54.                 lock (_lock)
55.                 {
56.                     if (_instance == null)
57.                     {
58.                         _instance = new DatabaseConnection();
59.                     }
60.                 }
61.             }
62.             return _instance;
63.         }
64.     }
65.
66.     // Method to open connection
67.     public void OpenConnection()
68.     {
69.         Console.WriteLine("Database connection opened.");
70.         // In a real application, this would open an actual database
    connection
71.     }
72.
73.     // Method to execute query
74.     public void ExecuteQuery(string query)
75.     {
76.         Console.WriteLine($"Executing query: {query}");
77.         // In a real application, this would execute the query on the
    database
78.     }
79.
80.     // Method to close connection
```

```
    public void CloseConnection()
    {
        Console.WriteLine("Database connection closed.");
        // In a real application, this would close the database connection
    }
}
```

Benefits:

1. Resource Conservation: Ensures only one database connection is created, conserving system resources.
2. Global Access: Provides a global point of access to the database connection throughout the application.
3. Lazy Initialization: The instance is only created when first requested, improving application startup performance.
4. Thread Safety: The implementation ensures thread safety using the double-check locking pattern.

SOLID Principles Application

1. Single Responsibility Principle (SRP)

Definition: A class should have only one reason to change, meaning it should have only one responsibility.

Implementation in Library Management System:

- Document Classes: Each document class (`Book`, `Magazine`, `Newspaper`) is responsible only for maintaining document-specific properties and behaviors.

```
public class Book : Document
{
    public string ISBN { get; set; }
    public int Pages { get; set; }

    // Book is only responsible for book-specific properties and behaviors
    public override string GetDocumentType() => "Book";
}
```

- LibraryManager: Focuses solely on managing library operations (adding, borrowing, returning documents).

```
public class LibraryManager : ILibrarySubject
{
```

```
    // LibraryManager is only responsible for managing library operations
    public void AddDocument(Document document)
    {
        _documents.Add(document);
        NotifyObservers(new LibraryEvent(EventType.DocumentAdded, document));
    }
}
```

- LoanFeeCalculator: Responsible only for calculating loan fees, delegating the specific calculation algorithm to strategy classes.

```
public class LoanFeeCalculator
{
    // LoanFeeCalculator is only responsible for coordinating fee calculation
    public decimal CalculateFee(Document document, int daysLoaned)
    {
        return _strategy.CalculateFee(document, daysLoaned);
    }
}
```

Benefits:

- Improved maintainability as each class has a clear, focused purpose
- Easier testing as classes have fewer responsibilities
- Reduced coupling between components

2. Open/Closed Principle (OCP)

Definition: Software entities should be open for extension but closed for modification.

Implementation in Library Management System:

- Document Hierarchy: The abstract `Document` class is closed for modification but open for extension through inheritance.

```
// Base Document class is closed for modification
public abstract class Document
{
    // Common properties and methods
    public abstract string GetDocumentType();
}

// But open for extension through inheritance
public class Newspaper : Document
{
    public string Edition { get; set; }
```

```
12.
13.    public override string GetDocumentType() => "Newspaper";
14.}
```

- Strategy Pattern: The `ILoanFeeStrategy` interface is closed for modification, but new strategies can be added by implementing the interface.

```
1. // Strategy interface is closed for modification
2. public interface ILoanFeeStrategy
3. {
4.     decimal CalculateFee(Document document, int daysLoaned);
5. }
6.
7. // But open for extension through new implementations
8. public class NewspaperLoanFeeStrategy : ILoanFeeStrategy
9. {
10.    // New strategy implementation without modifying existing code
11.    public decimal CalculateFee(Document document, int daysLoaned)
12.    {
13.        // Implementation details
14.    }
15.}
```

- Observer Pattern: The notification system is designed to allow new event types without modifying existing code.

```
1. // Adding a new event type without modifying existing code
2. public enum EventType
3. {
4.     DocumentAdded,
5.     DocumentBorrowed,
6.     DocumentReturned,
7.     DocumentReserved  // New event type added without changing
   existing code
8. }
```

Benefits:

- Reduced risk when adding new features
- Improved code stability
- Enhanced reusability

3. Liskov Substitution Principle (LSP)

Definition: Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

Implementation in Library Management System:

- Document Hierarchy: All document types can be used wherever a `Document` is expected.

```
// Method that works with any Document type
public void ProcessDocument(Document document)
{
    Console.WriteLine($"Processing {document.GetDocumentType()}: {document.Title}");
    // Any document-specific operations use polymorphism
}

// Usage with different document types
Document book = new Book("Design Patterns", "Gang of Four", new DateTime(1994, 10, 21), "978-0201633610", 416);
Document magazine = new Magazine("Scientific American", "Various Authors", DateTime.Now, "Vol 326, No 5", "Springer Nature");

ProcessDocument(book);      // Works with Book
ProcessDocument(magazine);  // Works with Magazine
```

- **Observer Pattern: All observers implement the `ILibraryObserver` interface consistently.**

```
// Method that works with any ILibraryObserver
public void RegisterObserver(ILibraryObserver observer)
{
    if (!_observers.Contains(observer))
    {
        _observers.Add(observer);
    }
}

// Usage with different observer types
ILibraryObserver user = new User(1, "John Doe", "john@example.com");
ILibraryObserver adminUser = new AdminUser(2, "Admin", "admin@example.com");

RegisterObserver(user);       // Works with User
RegisterObserver(adminUser);  // Works with AdminUser
```

Benefits:

- **Improved code reusability**
- **Enhanced system flexibility**
- **Reduced coupling between components**

4. Interface Segregation Principle (ISP)

Definition: Clients should not be forced to depend on interfaces they do not use.

Implementation in Library Management System:

- Observer Interfaces: The ILibraryObserver interface is focused and minimal, containing only the methods needed by observers.

```
// Focused interface with only necessary methods
public interface ILibraryObserver
{
    void Update(LibraryEvent libraryEvent);
}
```

- Strategy Interfaces: The ILoanFeeStrategy interface contains only the methods needed for fee calculation.

```
// Focused interface with only necessary methods
public interface ILoanFeeStrategy
{
    decimal CalculateFee(Document document, int daysLoaned);
}
```

Benefits:

- Reduced coupling between components
- Improved code maintainability
- Enhanced system flexibility

5. Dependency Inversion Principle (DIP)

Definition: High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

Implementation in Library Management System:

- Observer Pattern: LibraryManager depends on the ILibraryObserver interface, not concrete observer classes.

```
public class LibraryManager : ILibrarySubject
{
    // Depends on abstraction (ILibraryObserver), not concrete classes
    private List<ILibraryObserver> _observers = new
List<ILibraryObserver>();

    public void NotifyObservers(LibraryEvent libraryEvent)
    {
        foreach (var observer in _observers)
        {
            // Works with any class that implements ILibraryObserver
            observer.Update(libraryEvent);
        }
    }
}
```

- Strategy Pattern: LoanFeeCalculator depends on the ILoanFeeStrategy interface, not concrete strategy classes.

```
public class LoanFeeCalculator
{
    // Depends on abstraction (ILoanFeeStrategy), not concrete classes
    private ILoanFeeStrategy _strategy;

    public LoanFeeCalculator(ILoanFeeStrategy strategy)
    {
        _strategy = strategy;
    }

    public decimal CalculateFee(Document document, int daysLoaned)
    {
        // Works with any class that implements ILoanFeeStrategy
        return _strategy.CalculateFee(document, daysLoaned);
    }
}
```

- Factory Pattern: Client code depends on the abstract `Document` class and `DocumentFactory` class, not concrete implementations.

```
// Client code depends on abstractions
DocumentFactory factory = new BookFactory("Design Patterns", "Gang of Four", new DateTime(1994, 10, 21), "978-0201633610", 416);
Document document = factory.GetDocument();
```

Benefits:

- Reduced coupling between components
- Improved code maintainability
- Enhanced system flexibility and testability

Conclusion

The Library Management System demonstrates effective use of design patterns and SOLID principles to create a maintainable, extensible, and robust system.

Design Patterns Summary

- Observer Pattern: Enables efficient notification handling between the library system and users, allowing for loose coupling and dynamic registration of observers.

- Factory Method Pattern: Manages document creation, encapsulating creation logic and promoting extensibility for new document types.

- Strategy Pattern: Provides flexible fee calculation algorithms that can be selected and switched at runtime.

- **Singleton Pattern: Ensures efficient resource management for database connections.**

SOLID Principles Summary

- **Single Responsibility Principle: Each class has a well-defined, focused purpose.**
- **Open/Closed Principle: The system is designed to be extended without modifying existing code.**
- **Liskov Substitution Principle: Subclasses can be used in place of their parent classes without affecting program correctness.**
- **Interface Segregation Principle: Interfaces are focused and minimal, containing only necessary methods.**
- **Dependency Inversion Principle: High-level modules depend on abstractions, not concrete implementations.**

These patterns and principles, when applied together, result in a well-structured, maintainable, and extensible codebase that can evolve to meet changing requirements with minimal risk and effort.