Trần Lê Hiền Đức

2131200122131200129

Github : https://github.com/hien-duc/cse422-coding-practice/tree/master/Lab5_TranLeHienDuc_CSE422

Library Management System Report

1. Introduction

This library management system is designed to meet the requirements of document management, notification system, loan fee calculation, and database connection management. It utilizes several design patterns to achieve modularity, extensibility, and maintainability.

2. Design Patterns Used

2.1 Singleton Pattern

- **Purpose**: Ensures that the system has only one instance of the database connection throughout its operation. This is crucial for resource management and maintaining data consistency.
- **Implementation**:
  - In the `DatabaseConnection` class, a private static `Lazy<DatabaseConnection>` instance is used for lazy initialization. The constructor of `DatabaseConnection` is made private to prevent external instantiation.
  - A public static property `Instance` is provided to access the single instance of the database connection.
  - **Usage**: In the `Library` class constructor, the database connection is obtained using `DatabaseConnection.Instance`.
  - Example Code:

```
public sealed class DatabaseConnection
{
    private static readonly Lazy<DatabaseConnection> _instance =
        new Lazy<DatabaseConnection>(() => new DatabaseConnection());

    public static DatabaseConnection Instance => _instance.Value;

    private DatabaseConnection()
    {
        Console.WriteLine("Database connection initialized.");
    }

    public void ExecuteQuery(string query)
    {
        Console.WriteLine($"Executing query: {query}");
    }
}
```

2.2 Factory Method Pattern

- **Purpose**: Used to create different types of document objects (Book, Magazine, Newspaper) in a flexible and extensible way. It decouples the object creation logic from the client code.
- **Implementation**:
    - The `DocumentFactory` class has a `CreateDocument` method that takes the document type, title, publisher, and year as parameters.
    - Based on the document type, it returns an instance of the appropriate document class.
    - **Usage**: In the `DemonstrateFactoryPattern` method in the `Program` class, the `DocumentFactory` is used to create different document objects.
    - Example code:

```
public class DocumentFactory
{
    public Document CreateDocument(string type, string title, string publisher, int year)
    {
        return type.ToLower() switch
        {
            "book" => new Book(title, publisher, year),
            "magazine" => new Magazine(title, publisher, year),
            "newspaper" => new Newspaper(title, publisher, year),
            _ => throw new ArgumentException($"Invalid document type: {type}")
        };
    }
}
```

2.3 Observer Pattern

- **Purpose**: Allows registered users to be notified when certain events occur in the library, such as a new document being added or a document being borrowed or returned.
- **Implementation**:
    - Interfaces `ISubject` and `IObserver` are defined to represent the subject (library) and observers (users).
    - The `Library` class implements the `ISubject` interface and maintains a list of observers. It has methods to register, remove observers, and notify them.
    - The `User` class implements the `IObserver` interface and has an `Update` method to receive notifications.
    - **Usage**: In the `DemonstrateObserverPattern` method in the `Program` class, users are registered as observers, and various library operations are performed to trigger notifications.
    - Example code:

```
public interface ISubject
{
    void RegisterObserver(IObserver observer);
    void RemoveObserver(IObserver observer);
    void NotifyObservers(string message);
}

public interface IObserver
{
    void Update(string message);
}

public class Library : ISubject
{
    private readonly List<IObserver> _observers = new List<IObserver>();
    private readonly List<Document> _documents = new List<Document>();
    private readonly DatabaseConnection _dbConnection;

    public Library()
    {
        _dbConnection = DatabaseConnection.Instance;
    }

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

    public void RemoveObserver(IObserver observer)
    {
        if (_observers.Contains(observer))
            _observers.Remove(observer);
    }

    public void NotifyObservers(string message)
    {
        foreach (var observer in _observers)
            observer.Update(message);
    }

    public void AddDocument(Document document)
    {
        _documents.Add(document);
        _dbConnection.ExecuteQuery($"INSERT INTO Documents (Title, Publisher, Year) VALUES ('{document.Title}', '{document.Publisher}',
{document.Year})");
        NotifyObservers($"New document added: {document}");
    }

    public void BorrowDocument(Document document, User user)
    {
        if (document.IsAvailable)
        {
            document.IsAvailable = false;
            _dbConnection.ExecuteQuery($"INSERT INTO Loans (DocumentId, UserId, LoanDate) VALUES ({_documents.IndexOf(document)},
{user.Id}, '{DateTime.Now}')");
            NotifyObservers($"Document borrowed: {document}");
        }
    }

    public void ReturnDocument(Document document)
    {
        if (!document.IsAvailable)
        {
            document.IsAvailable = true;
            _dbConnection.ExecuteQuery($"UPDATE Loans SET ReturnDate = '{DateTime.Now}' WHERE DocumentId = {_documents.IndexOf(document)}
AND ReturnDate IS NULL");
            NotifyObservers($"Document returned: {document}");
        }
    }
}

public class User : IObserver
{
    public int Id { get; private set; }
    public string Name { get; private set; }
    public string Email { get; private set; }

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

    public void Update(string message)
    {
        Console.WriteLine($"Notification: {message}");
    }
}
```

2.4 Strategy Pattern

- **Purpose**: Enables the calculation of loan fees based on the document type and loan duration. Different strategies can be easily added or modified without affecting the client code.
- **Implementation**:
    - An interface `ILoanFeeStrategy` is defined with a `CalculateFee` method.
    - Different strategy classes (`BookLoanFeeStrategy`, `MagazineLoanFeeStrategy`, `NewspaperLoanFeeStrategy`) implement the `ILoanFeeStrategy` interface and provide their own fee calculation logic.
    - The `LoanProcessor` class uses the appropriate strategy based on the document type to calculate the loan fee.
    - **Usage**: In the `DemonstrateStrategyPattern` method in the `Program` class, the `LoanProcessor` is used to calculate loan fees for different documents.
    - Example code:

```
public interface ILoanFeeStrategy
{
    decimal CalculateFee(int days);
}

public class BookLoanFeeStrategy : ILoanFeeStrategy
{
    private const decimal DailyRate = 0.50m;
    private const int GracePeriod = 14;

    public decimal CalculateFee(int days)
    {
        if (days <= GracePeriod)
            return 0;

        return (days - GracePeriod) * DailyRate;
    }
}

public class MagazineLoanFeeStrategy : ILoanFeeStrategy
{
    private const decimal DailyRate = 1.00m;
    private const int GracePeriod = 7;

    public decimal CalculateFee(int days)
    {
        if (days <= GracePeriod)
            return 0;

        return (days - GracePeriod) * DailyRate;
    }
}

public class NewspaperLoanFeeStrategy : ILoanFeeStrategy
{
    private const decimal DailyRate = 2.00m;
    private const int GracePeriod = 1;

    public decimal CalculateFee(int days)
    {
        if (days <= GracePeriod)
            return 0;

        return (days - GracePeriod) * DailyRate;
    }
}

public class LoanProcessor
{
    public decimal CalculateFee(Document document, int days)
    {
        ILoanFeeStrategy strategy = document switch
        {
            Book => new BookLoanFeeStrategy(),
            Magazine => new MagazineLoanFeeStrategy(),
            Newspaper => new NewspaperLoanFeeStrategy(),
            _ => throw new ArgumentException($"Unknown document type:
{document};GetType().Name}")

        return strategy.CalculateFee(days);
    }
}
```

3. Database Design

The system uses three main tables:

- **Documents Table**: Stores information about documents, such as title, publisher, and year of publication.
- **Users Table**: Manages user information, including user ID, name, and email.
- **Loans Table**: Keeps track of loan and return information, including document ID, user ID, loan date, and return date.

4. System Implementation Flow

- **Initialization**: The database connection is initialized using the singleton pattern in the `Library` class constructor.
- **Document Creation**: The factory method pattern is used to create different types of document objects in the `DemonstrateFactoryPattern` method.
- **User Registration**: Users are registered as observers in the `DemonstrateObserverPattern` method.
- **Document Management**: Documents are added, borrowed, and returned in the `DemonstrateObserverPattern` method, and notifications are sent to registered users.
- **Loan Fee Calculation**: The strategy pattern is used to calculate loan fees in the `DemonstrateStrategyPattern` method.

5. Conclusion

The library management system effectively utilizes the singleton, factory method, observer, and strategy patterns to achieve its goals. These design patterns enhance the system's modularity, extensibility, and maintainability. The database design provides a solid foundation for managing library data, and the implementation flow ensures smooth operation of the system.