Student’s Full Name: Nguyen Cong Minh

Student ID: 2131200085

Design Pattern

I. DON’T REPEAT YOURSELF (DRY)

Exercise 1: Applying “Reflection” to Reduce Redundancy in Logging

1. What are Reflection and Expression Trees?

Reflection allows inspecting and manipulating types at runtime, while Expression Trees represent code structure as data.

2. Analyze the redundancy in the above code.

Redundancy exists in repeated `Log` method structures with different parameter types.

3. Rewrite the code using Reflection or Expression Trees to eliminate redundancy in logging.

Exercise 2: Applying Dependency Injection to Remove Redundancy in the Repository

1. Identify DRY violations in the above code.

Duplicated database connection and query logic violates DRY.

2. Rewrite the code using Generic Repository Pattern combined with Dependency Injection to reduce redundancy.

Exercise 3: Using Generic Constraints to Eliminate Redundancy in API Request Handling

1. Identify redundancy in the above code.

Repeated validation and response logic exists.

2. Rewrite the code using a Generic Base Controller and Generic Constraints to reduce duplication in API request handling.

Exercise 4: Combining Strategy Pattern with Factory Pattern to Avoid Redundancy in Payment Processing

1. What are Strategy Pattern and Factory Pattern?

Strategy Pattern defines interchangeable algorithms, and Factory Pattern encapsulates object creation.

2. Identify DRY violations in the above code.

Duplicated payment processing logic violates DRY.

3. Rewrite the code using Strategy Pattern combined with Factory Pattern to eliminate redundancy and allow easy expansion for new payment methods.

Exercise 5: Eliminating Redundancy in Cache Handling with Decorator Pattern

You have a service handling data queries with caching as follows:

1. Identify redundancy in cache handling.

Duplicated caching logic in ProductService and UserService.

2. Rewrite the code using the Decorator Pattern to avoid duplication in caching for both ProductService and UserService.

II. PACKAGES

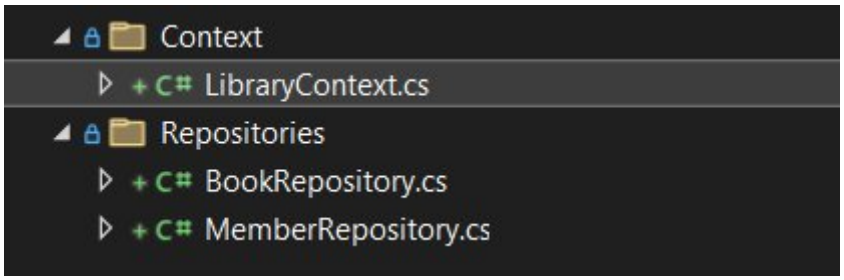
Exercise 6: Based on the principles of Packages in Architecture, design a solution in .NET consisting of multiple projects, where each project serves as a package responsible for a specific functionality. Determine the minimum number of projects required to build a complete library management system, and explain the role of each project within the overall architecture. Ensure that the system is scalable, maintainable, and can be easily upgraded in the future.

Project Structure:

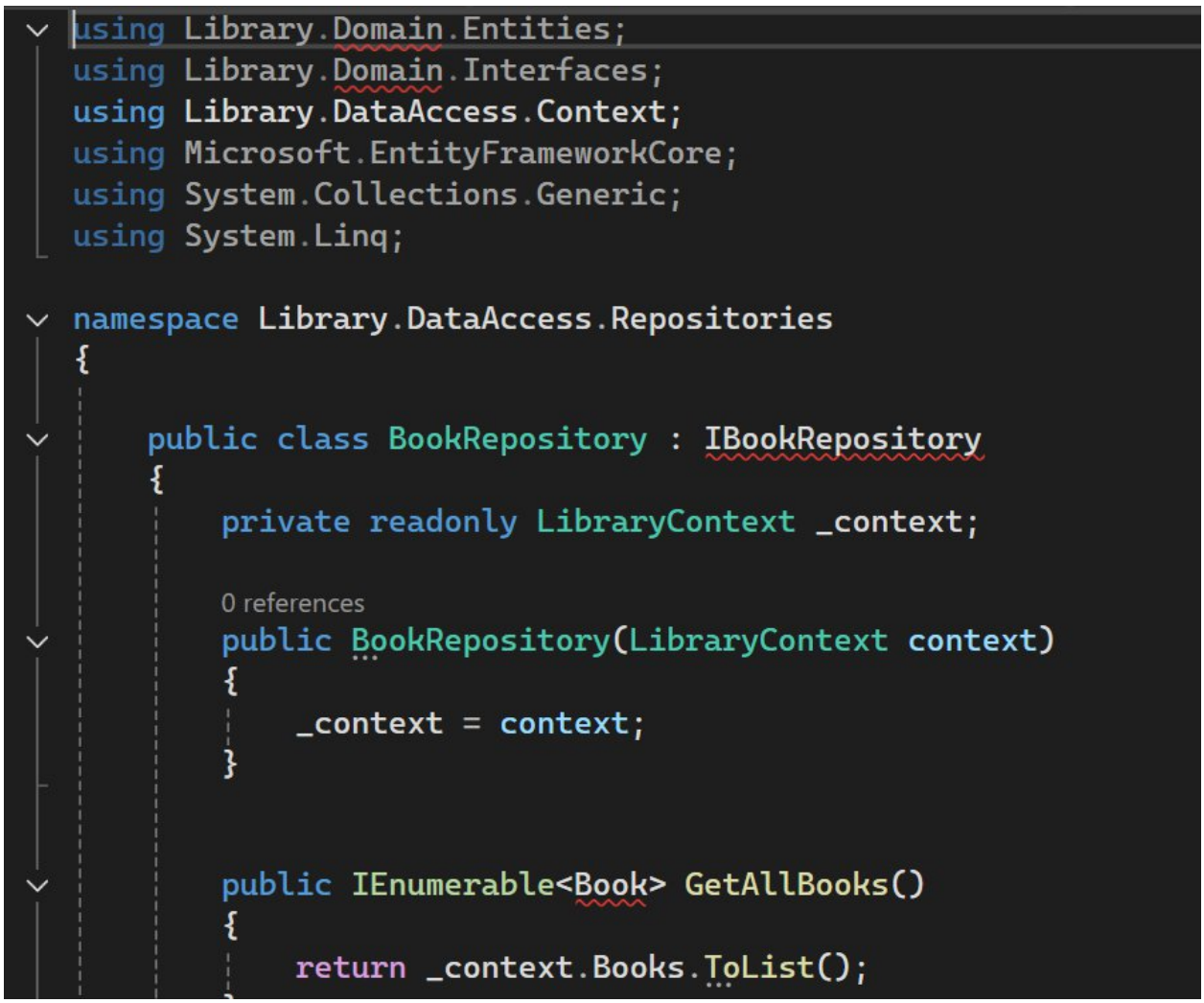
- **Library.Domain**: Contains the core business entities, domain models, and business rules.

```
Library.Domain                                  Library.Domain.Interfaces.IMemberRepository
using Library.Domain.Entities;

namespace Library.Domain.Interfaces
{
    public interface IMemberRepository
    {
        IEnumerable<Member> GetAllMembers();

        Member GetMemberById(int id);
        0 references
        void AddMember(Member member);

        void UpdateMember(Member member);

        void DeleteMember(int id);
    }
}
```

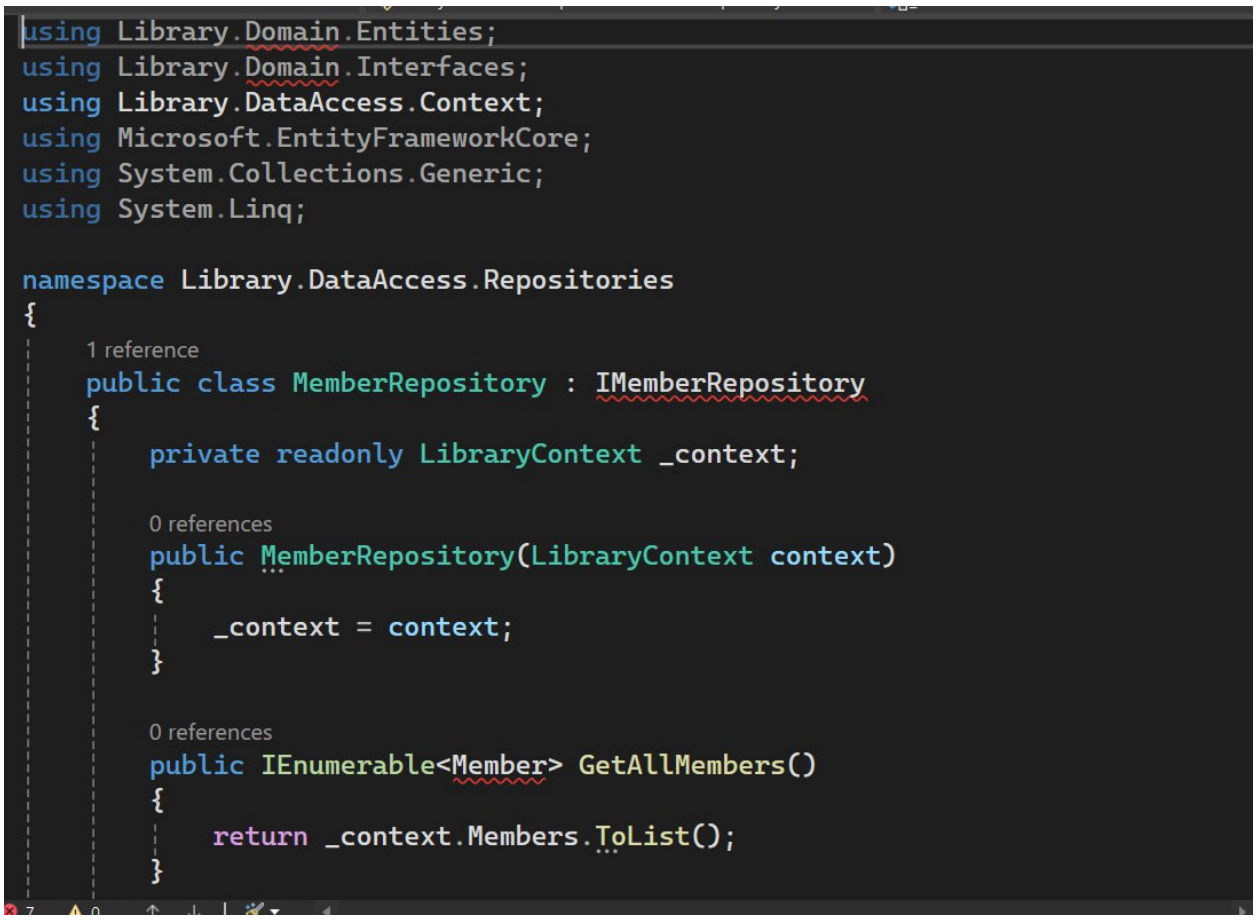
- **Library.DataAccess**: Handles data access operations, such as database interactions.




```
namespace Library.DataAccess.Context
{
    6 references
    public class LibraryContext : DbContext
    {
        0 references
        public LibraryContext(DbContextOptions<LibraryContext> options) : base(options
        {
        }

        6 references
        public DbSet<Book> Books { get; set; }
        6 references
        public DbSet<Member> Members { get; set; }
    }
}
```



```
using Library.Domain.Entities;
using Library.Domain.Interfaces;
using Library.DataAccess.Context;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;

namespace Library.DataAccess.Repositories
{

    public class BookRepository : IBookRepository
    {
        private readonly LibraryContext _context;

        0 references
        public BookRepository(LibraryContext context)
        {
            _context = context;
        }


        public IEnumerable<Book> GetAllBooks()
        {
            return _context.Books.ToList();
```



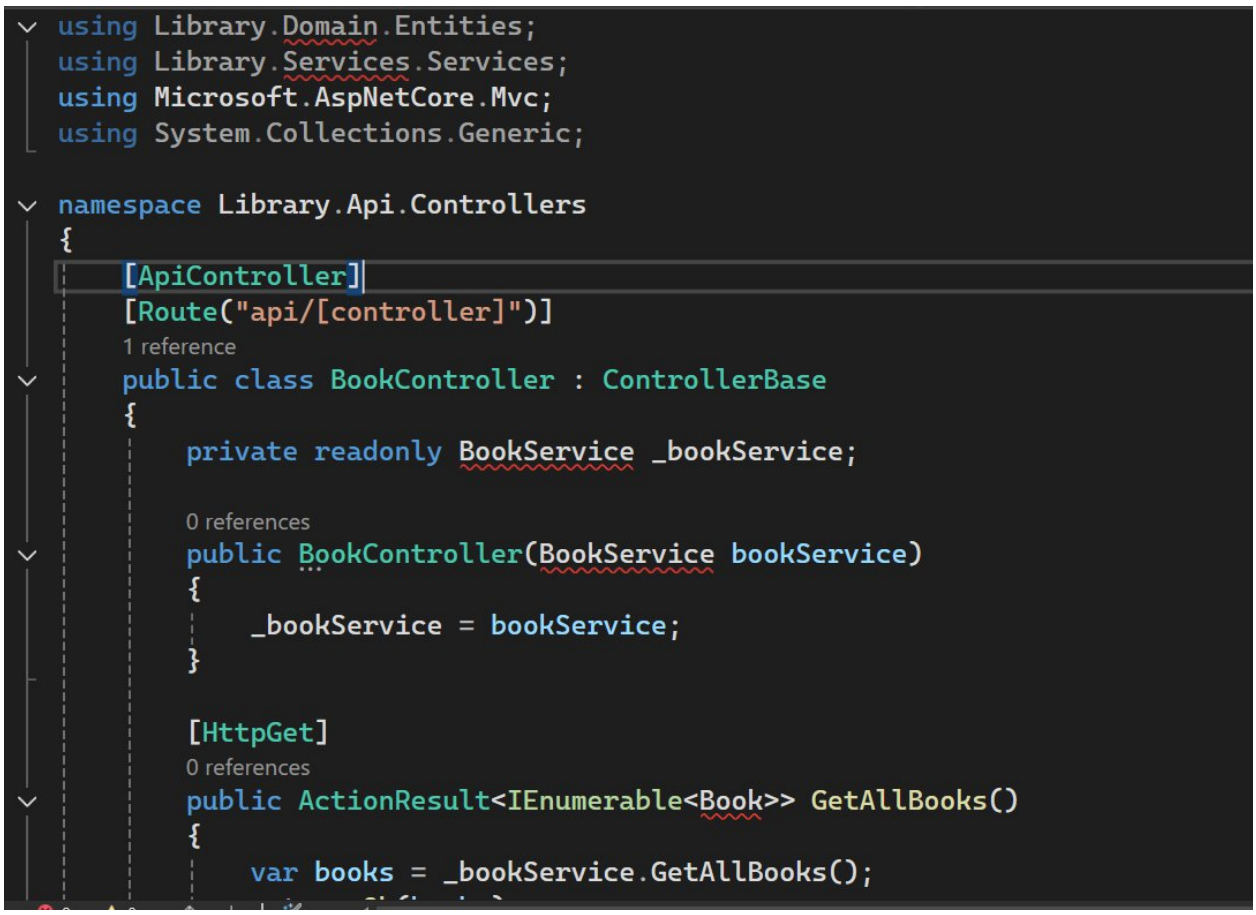
```
using Library.Domain.Entities;
using Library.Domain.Interfaces;
using Library.DataAccess.Context;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;

namespace Library.DataAccess.Repositories
{
    1 reference
    public class MemberRepository : IMemberRepository
    {
        private readonly LibraryContext _context;

        0 references
        public MemberRepository(LibraryContext context)
        {
            _context = context;
        }

        0 references
        public IEnumerable<Member> GetAllMembers()
        {
            return _context.Members.ToList();
        }
```

- **Library.Services**: Implements the application - level services that orchestrate the business logic.

```
using Library.Domain.Entities;
using Library.Domain.Interfaces;
using System.Collections.Generic;

namespace Library.Services.Services
{
    1 reference
    public class BookService
    {
        private readonly IBookRepository _bookRepository;


        public BookService(IBookRepository bookRepository)
        {
            _bookRepository = bookRepository;
        }


        public IEnumerable<Book> GetAllBooks()
        {
            return _bookRepository.GetAllBooks();
        }

        0 references
        public Book GetBookById(int id)
```

- **Library.Api**: Exposes the system's functionality through a RESTful API.

```
using Library.DataAccess.Context;
using Microsoft.EntityFrameworkCore;
using Library.Domain.Interfaces;
using Library.DataAccess.Repositories;
using Library.Services.Services;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddDbContext<LibraryContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnectior

builder.Services.AddScoped<IBookRepository, BookRepository>();
builder.Services.AddScoped<BookService>();

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuc
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
```



```
using Library.Domain.Entities;
using Library.Services.Services;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

namespace Library.Api.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    1 reference
    public class BookController : ControllerBase
    {
        private readonly BookService _bookService;

        0 references
        public BookController(BookService bookService)
        {
            _bookService = bookService;
        }

        [HttpGet]
        0 references
        public ActionResult<IEnumerable<Book>> GetAllBooks()
        {
            var books = _bookService.GetAllBooks();
```