**Practice Assignment 6**

Exercise 1:

1. What are Reflection and Expression Trees?

* Reflection is the process of describing the metadata of types, methods and fields in a code. The namespace **System.Reflection** enables you to obtain data about the loaded assemblies, the elements within them like classes, methods and value types.
* Expression tree

1. Analyze the redundancy in the above code.

* The provided code has redundant logging methods (LogUserAction, LogTransaction, and LogError). Each method logs a different type of message, but they all follow a similar pattern, which leads to code duplication.

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

A screenshot of a computer program

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

A close-up of a computer code

AI-generated content may be incorrect.

A close-up of a computer code

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

Exercise 2:

1. Identify DRY violations in the above code.

* **Duplicate Code Structure**:
* Both StudentRepository and TeacherRepository have nearly identical logic for database connection and retrieval.
* The GetAllStudents() and GetAllTeachers() methods follow the same pattern, differing only in table names and entity types.
* **Redundant Database Handling**: Both repositories directly create SqlCommand objects and execute ExecuteReader(), which can be abstracted.
* **Lack of Reusability**: If a new entity (e.g., Course) is introduced, similar repository code would need to be written again.

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

A close-up of a computer code

AI-generated content may be incorrect.

A computer screen shot of a code

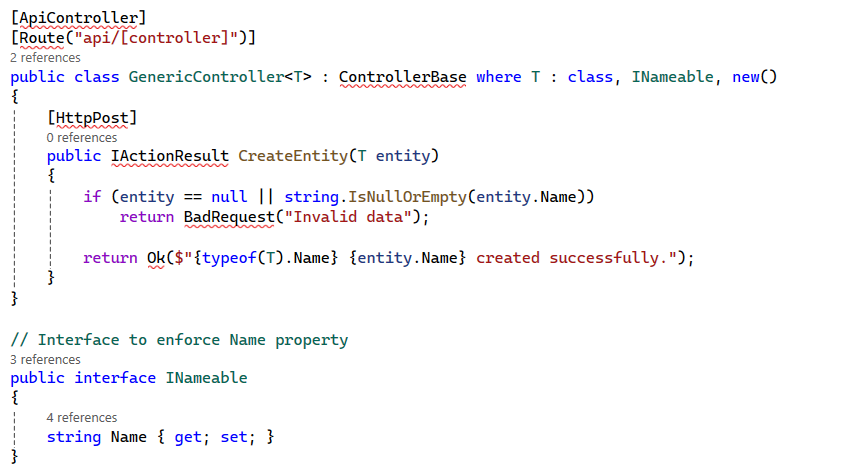
AI-generated content may be incorrect.

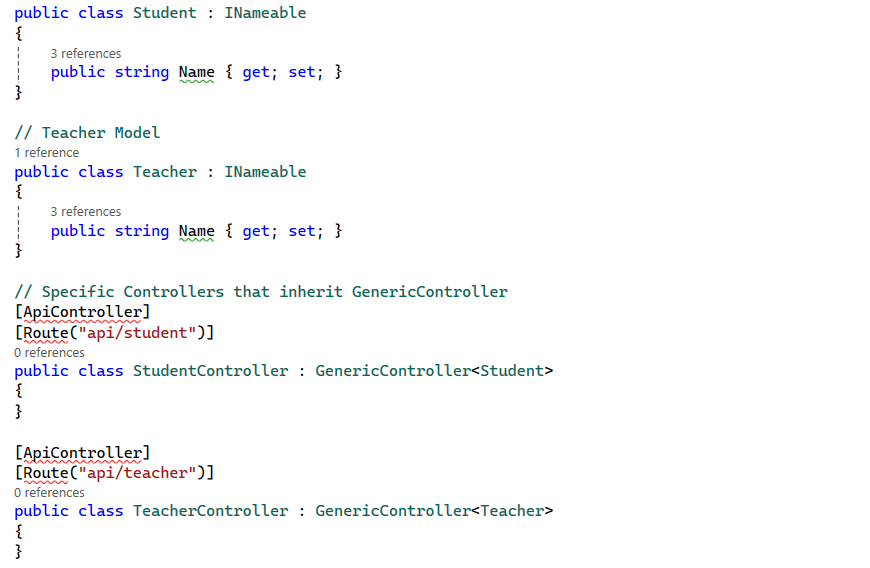
Exercise 3:

1. Identify redundancy in the above code.

* **Duplicate Code Structure**: Each controllers manage a different type of object, but they all follow a similar pattern, which leads to code duplication.

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





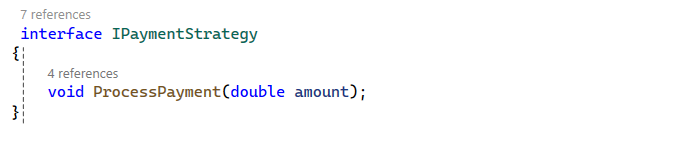
Exercise 4:

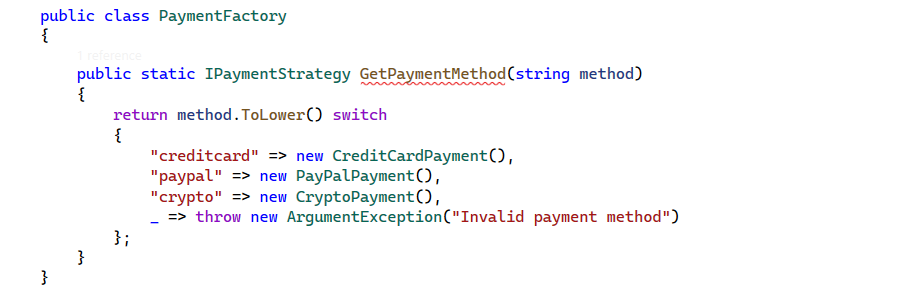
**1. What are Strategy Pattern and Factory Pattern?**

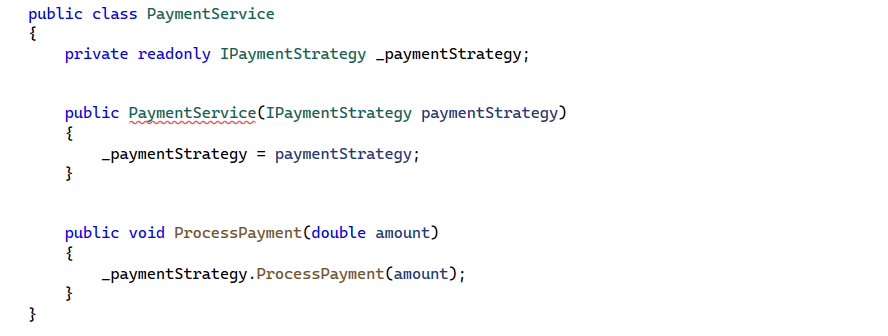
* **Strategy Pattern**: It defines a family of algorithms (strategies), encapsulates each one, and makes them interchangeable at runtime. It allows selecting a payment method dynamically without modifying the existing code.
* **Factory Pattern**: It provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects created. It helps centralize object creation logic.

**2. Identifying DRY Violations in the Given Code**

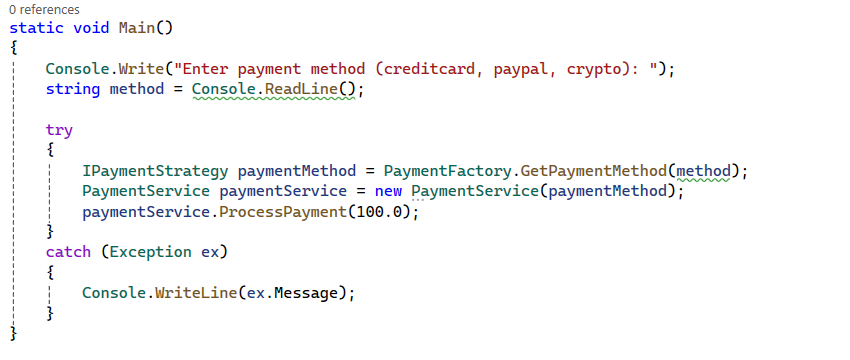
* The code **violates the DRY (Don't Repeat Yourself) principle** because:
  1. **Duplicate Method Logic**: Each method does the same thing—logs a message—with only minor differences in text.
  2. **Tightly Coupled Code**: The PaymentService class is tightly coupled with specific payment types. Adding a new payment method requires modifying the class, which breaks the **Open/Closed Principle**.
  3. **Refactored Code Using Strategy Pattern + Factory Pattern**









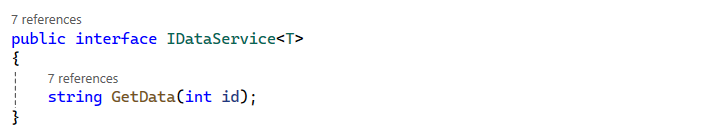


Exercise 5:

**1. Identified Redundancy in Cache Handling**

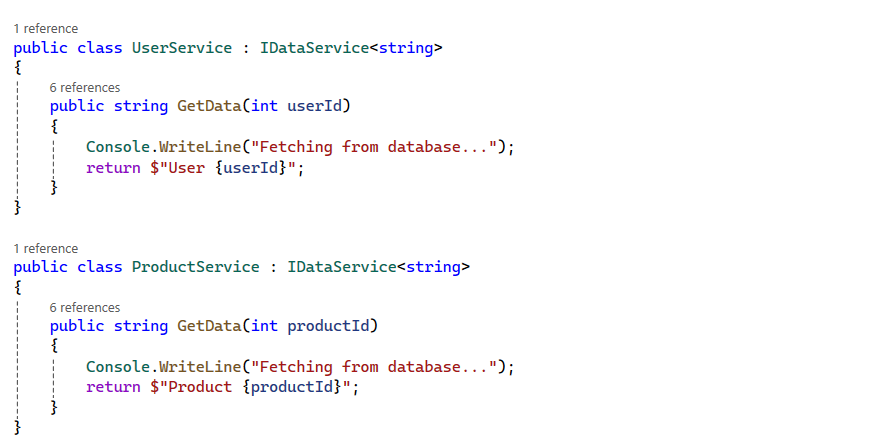
* **Both UserService and ProductService contain identical caching logic.**
* **The \_cache dictionary and its lookup logic (checking if a key exists, fetching from the cache or database) are duplicated.**
* **Code Duplication: If another service needs caching, we must rewrite the same logic again.**
* **Violates Single Responsibility Principle (SRP): Services should focus on fetching users/products, not handling caching.**

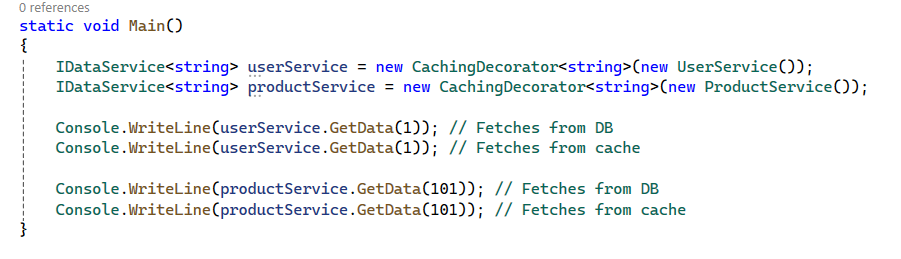
### 2. **Refactored Code Using the Decorator Pattern**



A computer screen shot of a program

AI-generated content may be incorrect.





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.

**Minimum Number of Projects Required**

A clean and layered architecture requires at least six core projects:

1. LibraryManagement.Core (Domain Layer)
2. LibraryManagement.Application (Application Layer)
3. LibraryManagement.Infrastructure (Infrastructure Layer)
4. LibraryManagement.Persistence (Data Layer)
5. LibraryManagement.API (Presentation Layer)
6. LibraryManagement.Tests (Testing Layer)

**Project Breakdown & Responsibilities**

**1. LibraryManagement.Core (Domain Layer)**

* Contains **business entities, domain models, and interfaces**.
* Defines core entities like Book, Member, Loan, and Librarian.
* Contains **domain-driven business logic** and **rules**.

**2. LibraryManagement.Application (Application Layer)**

* Contains **use cases, business logic, and service interfaces**.
* Implements **CQRS (Command-Query Responsibility Segregation)**.
* Uses **DTOs (Data Transfer Objects)** for communication.

**3. LibraryManagement.Infrastructure (Infrastructure Layer)**

* Implements **third-party services, logging, authentication, and caching**.
* Provides services like **email notifications, cloud storage, or external APIs**.

**4. LibraryManagement.Persistence (Data Layer)**

* Handles **database interactions** using **Entity Framework Core**.
* Manages **Repositories & Unit of Work Pattern**.
* Stores **Books, Members, Loans, and Librarians**.

**5. LibraryManagement.API (Presentation Layer)**

* Exposes the system as **RESTful APIs**.
* Uses **ASP.NET Core MVC/Web API**.
* Implements **Controllers for Book, Member, and Loan Management**.

**6. LibraryManagement.Tests (Testing Layer)**

* Contains **unit tests and integration tests**.
* Uses **Moq, xUnit/NUnit** to test business logic.