EASTERN INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTING AND INFORMATION TECHNOLOGY

Practice Assignment – Quarter 2, 2024-2025

Course Name: Coding Practice

Course Code: CSE 422

Student's Full Name:

Student ID:

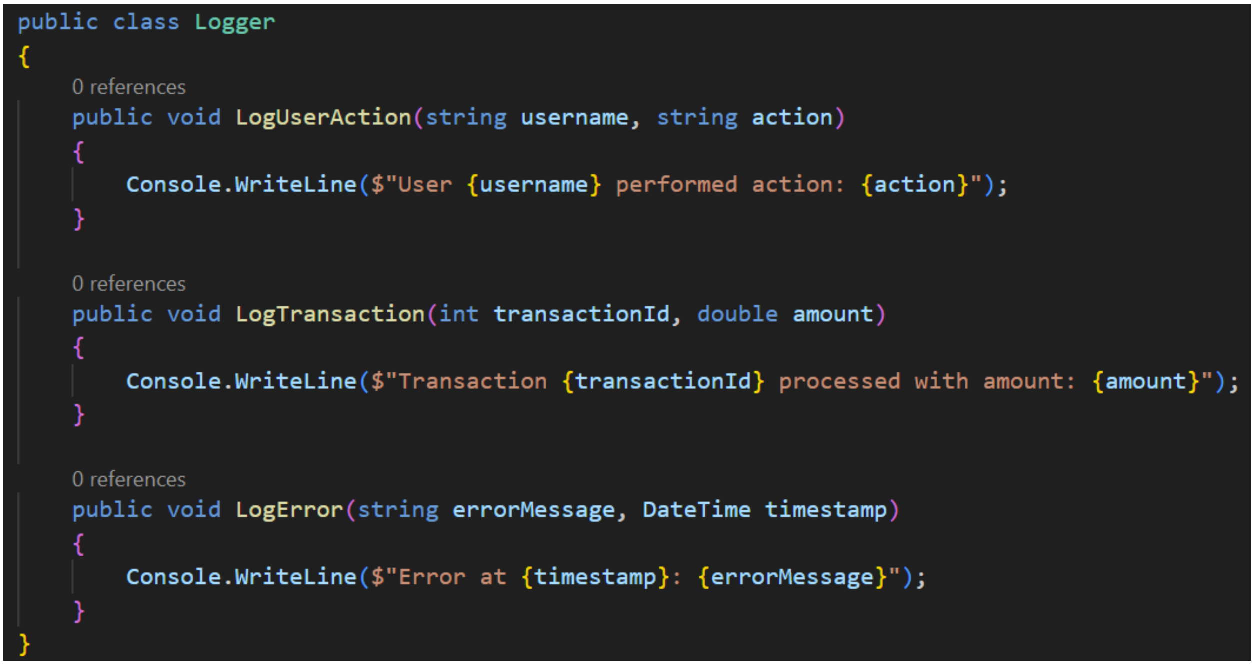
Practice Assignment 6

Design Pattern

I.DON'T REPEAT YOURSELF (DRY)

Exercise 1: Applying "**Reflection"** to Reduce Redundancy in Logging

You have a logging system as follows:



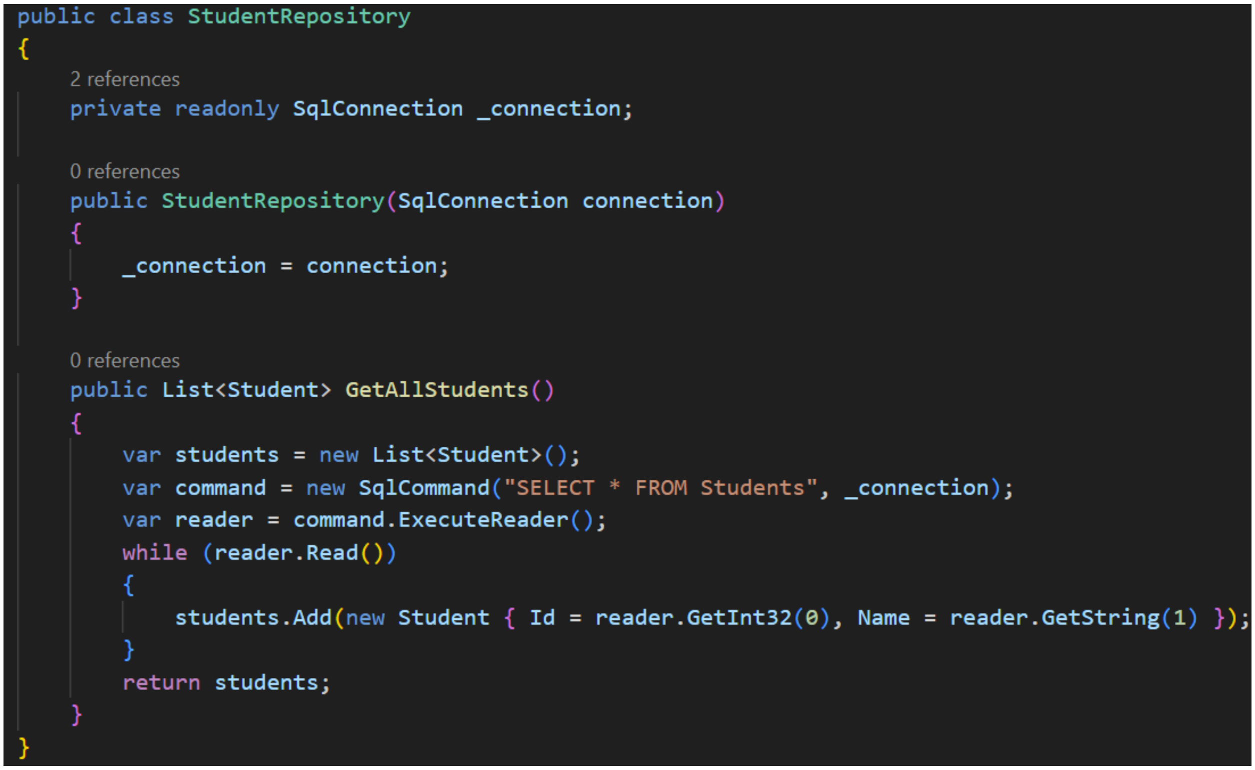
```
public class Logger
{
    0 references
    public void LogUserAction(string username, string action)
    {
        Console.WriteLine($"User {username} performed action: {action}");
    }

    0 references
    public void LogTransaction(int transactionId, double amount)
    {
        Console.WriteLine($"Transaction {transactionId} processed with amount: {amount}");
    }

    0 references
    public void LogError(string errorMessage, DateTime timestamp)
    {
        Console.WriteLine($"Error at {timestamp}: {errorMessage}");
    }
}
```

Requirements:

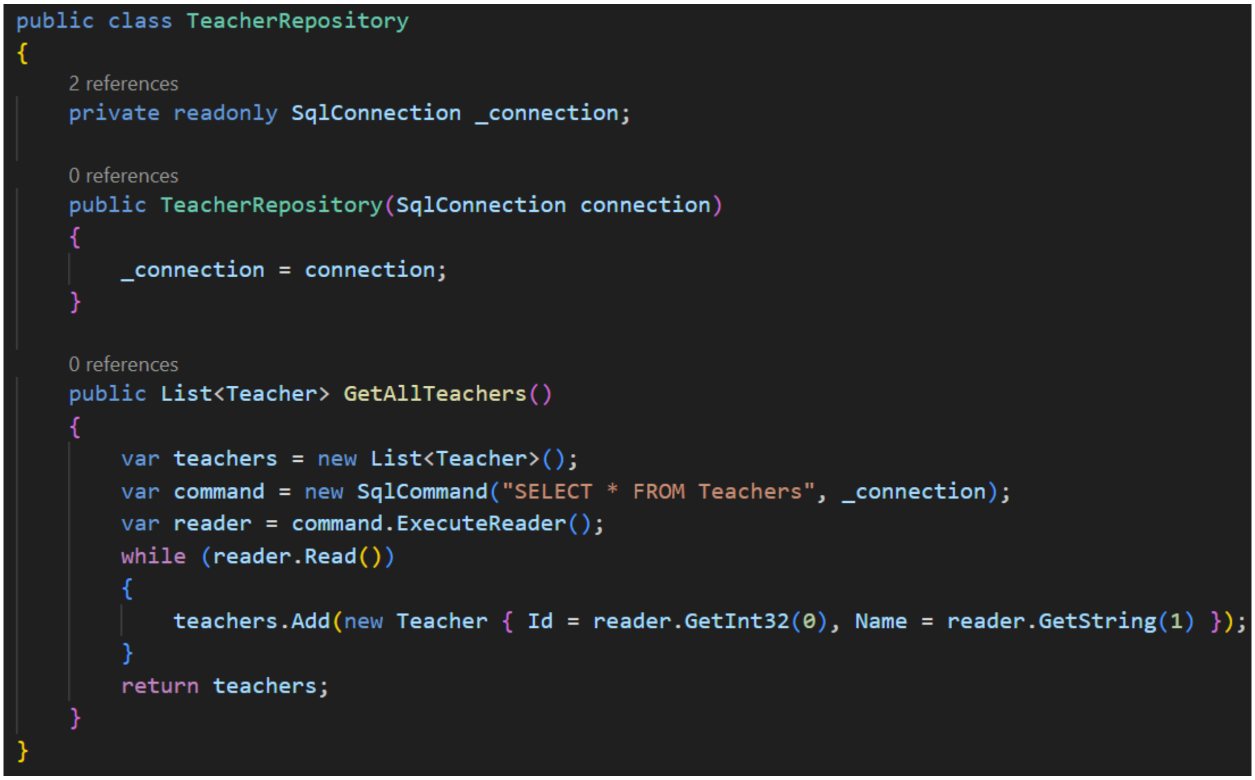
1. What are **Reflection** and **Expression Trees**?
2. Analyze the redundancy in the above code.
3. Rewrite the code using **Reflection** or **Expression Trees** to eliminate redundancy in logging.

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

You have two repositories storing data from SQL Server:



```
public class StudentRepository
{
    2 references
    private readonly SqlConnection _connection;

    0 references
    public StudentRepository(SqlConnection connection)
    {
        _connection = connection;
    }

    0 references
    public List<Student> GetAllStudents()
    {
        var students = new List<Student>();
        var command = new SqlCommand("SELECT * FROM Students", _connection);
        var reader = command.ExecuteReader();
        while (reader.Read())
        {
            students.Add(new Student { Id = reader.GetInt32(0), Name = reader.GetString(1) });
        }
        return students;
    }
}
```

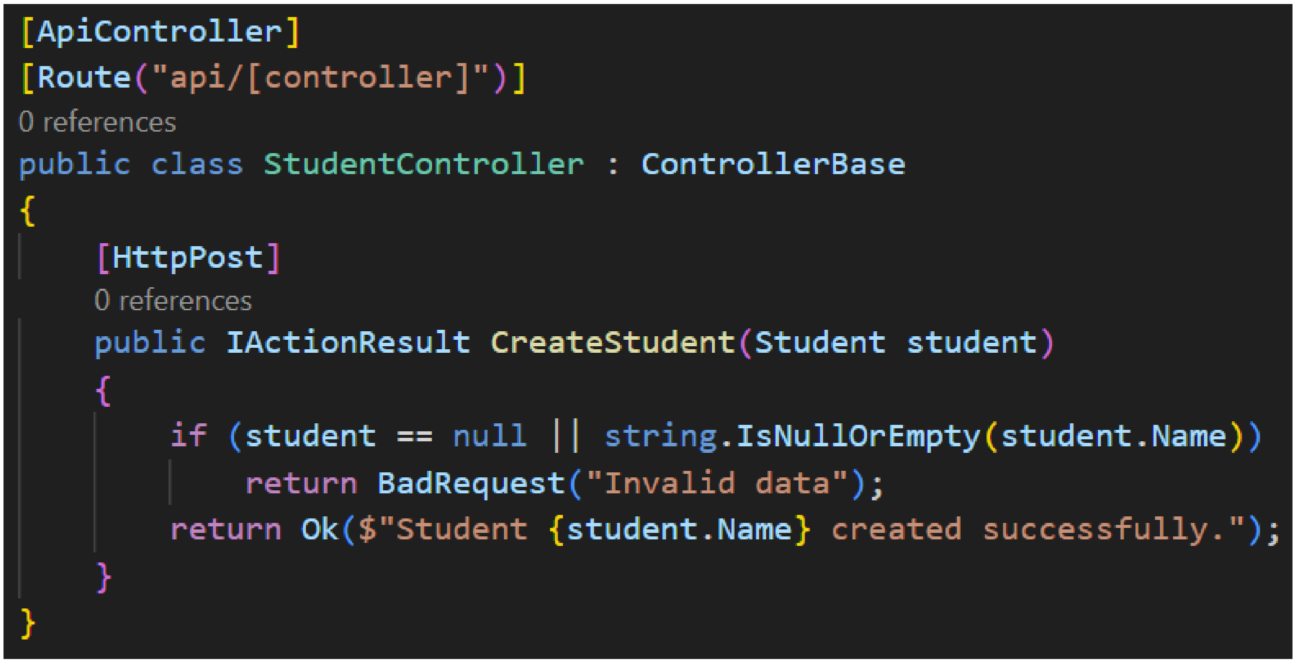


```
public class TeacherRepository
{
    2 references
    private readonly SqlConnection _connection;

    0 references
    public TeacherRepository(SqlConnection connection)
    {
        _connection = connection;
    }

    0 references
    public List<Teacher> GetAllTeachers()
    {
        var teachers = new List<Teacher>();
        var command = new SqlCommand("SELECT * FROM Teachers", _connection);
        var reader = command.ExecuteReader();
        while (reader.Read())
        {
            teachers.Add(new Teacher { Id = reader.GetInt32(0), Name = reader.GetString(1) });
        }
        return teachers;
    }
}
```

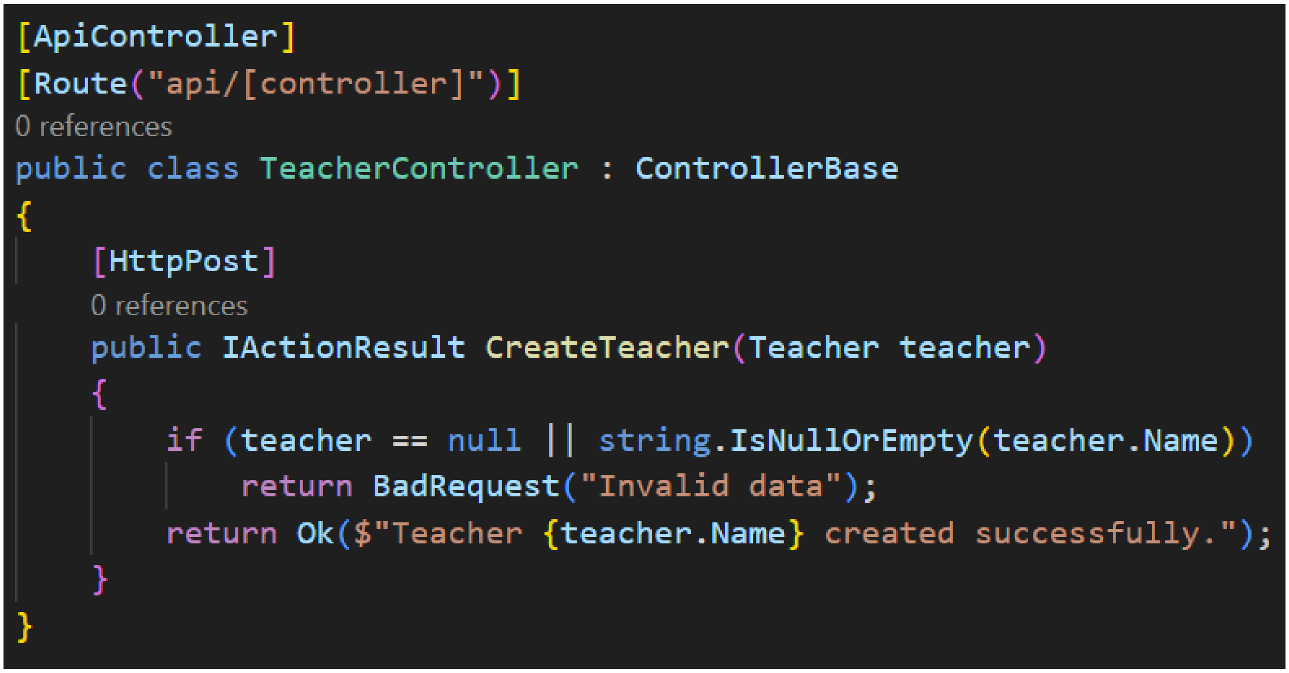
Requirements:

1. Identify DRY violations in the above code.
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
You have an API Controller handling two different object types as follows:



```
[ApiController]
[Route("api/[controller]")]
0 references
public class StudentController : ControllerBase
{
    [HttpPost]
    0 references
    public IActionResult CreateStudent(Student student)
    {
        if (student == null || string.IsNullOrEmpty(student.Name))
            return BadRequest("Invalid data");
        return Ok($"Student {student.Name} created successfully.");
    }
}
```

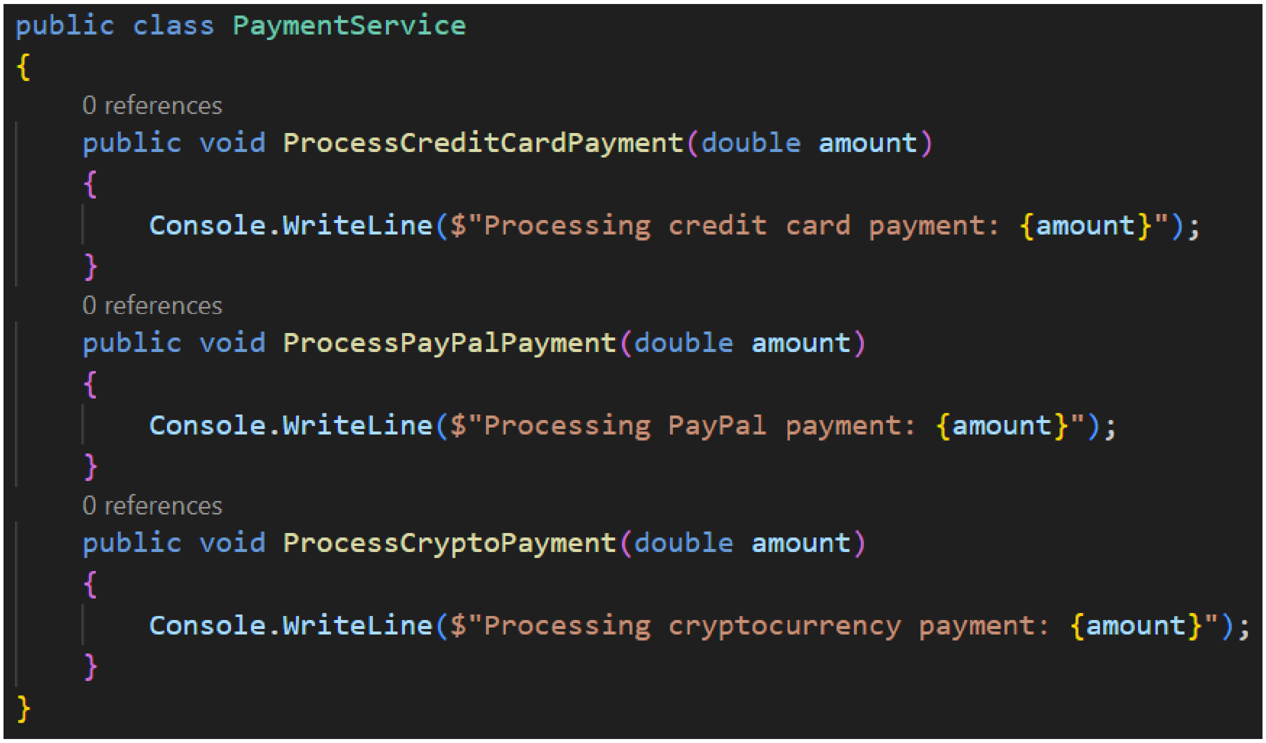


```
[ApiController]
[Route("api/[controller]")]
0 references
public class TeacherController : ControllerBase
{
    [HttpPost]
    0 references
    public IActionResult CreateTeacher(Teacher teacher)
    {
        if (teacher == null || string.IsNullOrEmpty(teacher.Name))
            return BadRequest("Invalid data");
        return Ok($"Teacher {teacher.Name} created successfully.");
    }
}
```

Requirements*:*

1. Identify redundancy in the above code.
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
You have a payment system that supports multiple payment methods:



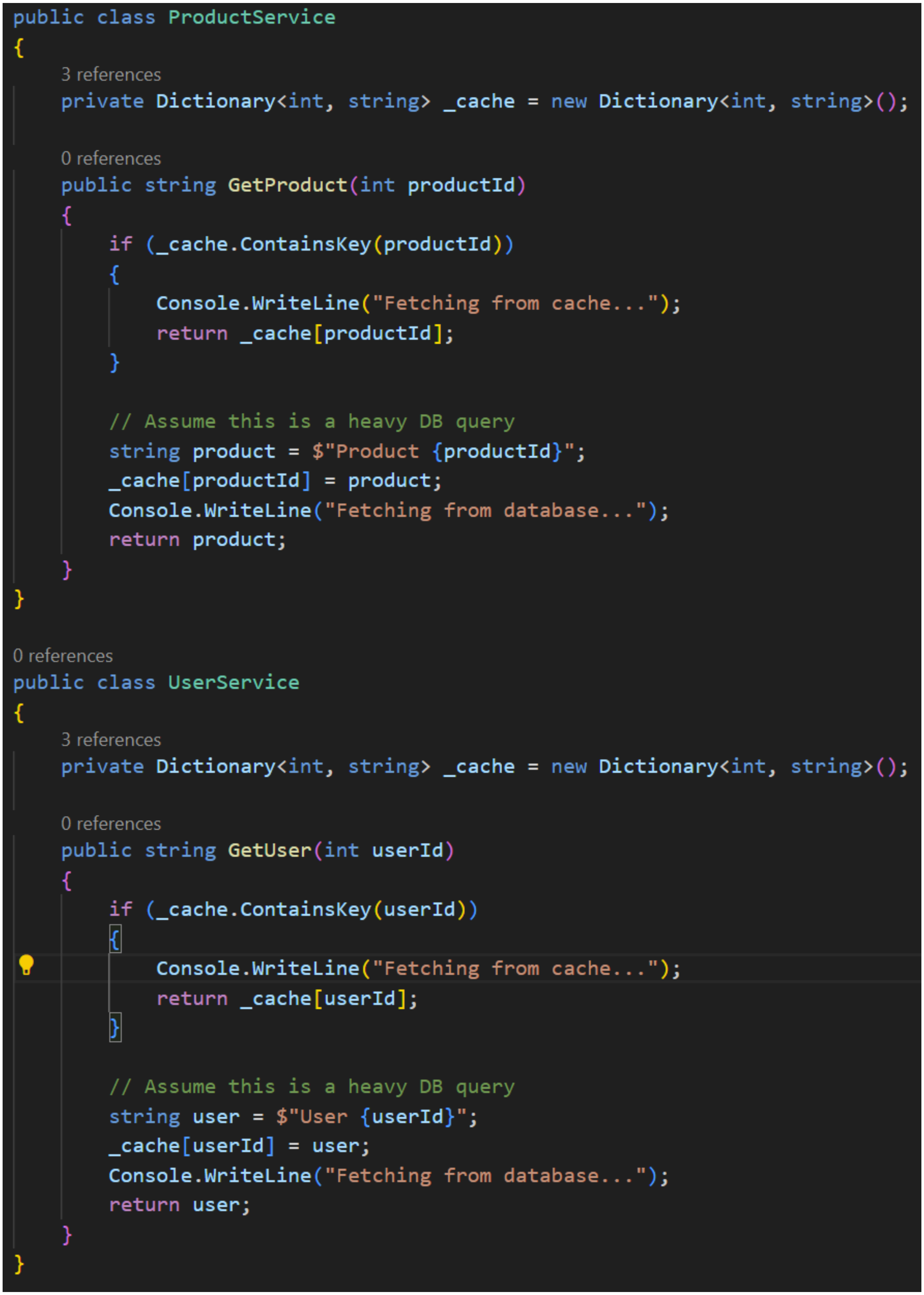
```
public class PaymentService
{
    0 references
    public void ProcessCreditCardPayment(double amount)
    {
        Console.WriteLine($"Processing credit card payment: {amount}");
    }
    0 references
    public void ProcessPayPalPayment(double amount)
    {
        Console.WriteLine($"Processing PayPal payment: {amount}");
    }
    0 references
    public void ProcessCryptoPayment(double amount)
    {
        Console.WriteLine($"Processing cryptocurrency payment: {amount}");
    }
}
```

Requirements:

1. What are **Strategy Pattern** and **Factory Pattern**?
2. Identify DRY violations in the above code.
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:



```
public class ProductService
{
    3 references
    private Dictionary<int, string> _cache = new Dictionary<int, string>();

    0 references
    public string GetProduct(int productId)
    {
        if (_cache.ContainsKey(productId))
        {
            Console.WriteLine("Fetching from cache...");
            return _cache[productId];
        }

        // Assume this is a heavy DB query
        string product = $"Product {productId}";
        _cache[productId] = product;
        Console.WriteLine("Fetching from database...");
        return product;
    }
}

0 references
public class UserService
{
    3 references
    private Dictionary<int, string> _cache = new Dictionary<int, string>();

    0 references
    public string GetUser(int userId)
    {
        if (_cache.ContainsKey(userId))
        {
            Console.WriteLine("Fetching from cache...");
            return _cache[userId];
        }

        // Assume this is a heavy DB query
        string user = $"User {userId}";
        _cache[userId] = user;
        Console.WriteLine("Fetching from database...");
        return user;
    }
}
```

Requirements:

1. Identify redundancy in cache handling.

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.