22127270 - Nguyễn Quang Minh

> Report about principle: `Single Responsibility Principle` and `Don't Repeat Yourself`

Introduction

Modern software engineering emphasizes maintainable, extensible, and robust code. Two key principles that contribute to these qualities are:

- Single Responsibility Principle (SRP)
  Ensures that a class or module has one reason to change, focusing on one responsibility. This reduces complexity and improves maintainability.
- Don't Repeat Yourself (DRY)
  Promotes reusability by avoiding code duplication. When code is repeated in multiple places, it increases the chance for errors and makes changes more difficult.

A> Single Responsibility Principle

> SRP is one of the five SOLID principles of object-oriented design

- It stated that: `A class should have only one reason to change`
  => A class/modules/function should have `one and only one responsibility`
  => If a class has multiple responsibilities, changes in one responsibility might impact others, making the code harder to maintain and understand.
- Benefits:
  - Maintainability: Changes in one area of functionality affect only one class.
  - Testability: Smaller, focused classes are easier to test.
  - Extensibility: New features can be added without impacting unrelated functionalities

B> Don't Repeat Yourself

> Every piece of knowledge must have a single unambiguous, authoritative representation within a system.

- In practice, this means avoiding duplicate code.
- Benefits:
  - Reduced Errors: Fixing a bug in one location fixes it for all uses.
  - Easier Refactoring: Changes need to be made in only one place
  - Cleaner Code Base: Code is more readable and easier to maintain.
- Common Pitfall

- Repeating similar code across modules leads to inconsistency and extra work when changes are required.

C> Example

C.1. Example for SRP:

Example 1: Logging and Business Logic Separation:

- Before (Violating SRP)

```
public class PaymentMethod {
    public bool ProcessPayment(decimal amount) {
        if (amount < 0) {
            Log("Invalid payment amount")
            return false;
        }
        Log("Valid payment amount");
        return true;
    }
    private void Log(string message) {
        Console.WriteLine(message);
    }
}
```

- Applying SRP

```
public class PaymentProcessor {
    private readonly ILogger _logger;
    public PaymentProcessor(ILogger logger) {
        _logger = logger
    }
    public bool ProcessPayment(decimal amount) {
        if (amount < 0) {
            _logger.Log("Invalid amount")
            return false;
        }
        _logger.Log("Valid amount")
        return true;
    }
```

```
}
public interface ILogger {
    void Log(string message);
}
public class ConsoleLogger : ILogger {
    public void Log(string message) {
        Console.WriteLine(message);
    }
}
```

> In the refactored code, the `PaymentProcessor` class is solely responsible for processing payments.
> `Logging` has been extracted into a separate interface `ILogger` and its implementation `ConsoleLogger`
> => This separation makes class easier to maintain, test and modify independently.

Example 2: A Class that both calculate total and mail to customer

- Before (Violating SRP):

```
public class Order
{
    public List<OrderItem> Items { get; set; }

    public Order(List<OrderItem> items)
    {
        Items = items;
    }

    public decimal CalculateTotal()
    {
        decimal total = 0;
        foreach (var item in Items)
        {
            total += item.Price * item.Quantity;
        }
        return total;
    }

    // Phương thức này gửi email xác nhận đơn hàng
```

```
    public void SendConfirmationEmail(string email)
    {
        decimal total = CalculateTotal();
        // Giả lập gửi email (ví dụ in ra console)
        Console.WriteLine($"Gửi email tới {email}: Đơn hàng của bạn có tổng
tiền {total}.");
    }
}

public class OrderItem
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    public int Quantity { get; set; }
}
```

- After (Apply SRP)

```
public class Order {
    public List<OrderItem> Items {get; set;}
    public Order(List<OrderItem> items) {
        Items = items;
    }
    public decimal CalculateTotal() {
        decimal total = 0;
        for (var item in Items) {
            total += item.Price * item.Quantity
        }
        return total;
    }
}
public class OrderItem() {
    public tring Name {get;set;} = "";
    public string Price {get;set;} = "";
    public int Quantity {get;set;} = 0;
}
public class EmailService() {
    public void SendOrderConfirmation(Order order, string email) {
        decimail total = order.CalculateTotal();
```

```
        Console.WriteLine($"Gửi email tới {email}: Đơn hàng của bạn có tổng
tiền {total}.");
    }
}
var items = new List<OrderItem> {
    new OrderItem { Name = "ProductA", Price=100, Quantity=1
    new OrderItem { Name = "ProductB", Price=100, Quantity=1 }
}
Order order = new Order(items);
EmailService emailService = new EmailService();
emailService.SendOrderConfirmation(order, "khachhang@example.com");
```

> In this Separation:
>
> - Order only take responsibility for manage Order Detail and calculate Total
> - EmailService class will take responsibility for email the customer.

C.2. Example for DRY

Example 1: Read data

- Suppose that we have many similar code but todo only one same thing:

```
public string ReadDataFile1()
{
    try
    {
        return File.ReadAllText("data1.txt");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Lỗi khi đọc data1.txt: {ex.Message}");
        return null;
    }
}

public string ReadDataFile2()
{
    try
    {
        return File.ReadAllText("data2.txt");
```

```
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Lỗi khi đọc data2.txt: {ex.Message}");
        return null;
    }
}
```

- Improve

```
public string ReadDataFile(string filename)
{
    try
    {
        return File.ReadAllText(filename);
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Lỗi khi đọc {filename}: {ex.Message}");
        return null;
    }
}

// Sử dụng:
string data1 = ReadDataFile("data1.txt");
string data2 = ReadDataFile("data2.txt");
```

Example 2: Logging

- Logging structure repeat redundancy in many place:

```
...
    Console.WriteLine($"Error: {message} - Exception: {ex.Message}");
...
    Console.WriteLine($"Error: {message} - Exception: {ex.Message}");
```

- Make a logger function

```
public static class Logger
{
    public static void LogError(string message, Exception ex)
    {
        // Giả lập ghi log lỗi, có thể thay thế bằng việc ghi vào file hoặc hệ
thống log thực tế
        Console.WriteLine($"Error: {message} - Exception: {ex.Message}");
    }
}
```

C3. Apply both SRP and DRY

```
using System;
using System.Collections.Generic;
using System.IO;
using Newtonsoft.Json;

namespace OrderManagement
{
    // Lớp Order chỉ đảm nhận dữ liệu đơn hàng và tính toán tổng tiền
    public class Order
    {
        public List<OrderItem> Items { get; set; }

        public Order(List<OrderItem> items)
        {
            Items = items;
        }

        public decimal CalculateTotal()
        {
            decimal total = 0;
            foreach (var item in Items)
            {
                total += item.Price * item.Quantity;
            }
            return total;
        }
```

```
    }

    public class OrderItem
    {
        public string Name { get; set; }
        public decimal Price { get; set; }
        public int Quantity { get; set; }
    }

    // Lớp EmailService chịu trách nhiệm gửi email, không liên quan đến logic
của Order
    public class EmailService
    {
        // DRY: Sử dụng một hàm tiện ích chung để gửi email và xử lý ngoại lệ
        public void SendEmail(string to, string subject, string body)
        {
            try
            {
                // Giả lập gửi email: in ra console
                Console.WriteLine($"Gửi email tới: {to}");
                Console.WriteLine($"Subject: {subject}");
                Console.WriteLine($"Body: {body}");
            }
            catch (Exception ex)
            {
                Logger.LogError("Lỗi gửi email", ex);
            }
        }

        public void SendOrderConfirmation(Order order, string email)
        {
            string subject = "Xác nhận đơn hàng";
            string body = $"Đơn hàng của bạn có tổng tiền:
{order.CalculateTotal()}";
            // Sử dụng lại phương thức SendEmail đã đóng gói xử lý lỗi (DRY)
            SendEmail(email, subject, body);
        }
    }

    // Lớp Logger dùng chung cho toàn bộ ứng dụng (DRY: tránh lặp lại logic
```

```
ghi log)
    public static class Logger
    {
        public static void LogError(string message, Exception ex)
        {
            // Ví dụ ghi log đơn giản: In ra console, có thể ghi file hoặc sử
dụng hệ thống logging khác
            Console.WriteLine($"[ERROR] {message} - Exception: {ex.Message}");
        }
    }

    // Lớp FileHelper: Một ví dụ khác của DRY khi xử lý thao tác đọc/ghi file
JSON
    public static class FileHelper
    {
        // DRY: Hàm chung để serialize object sang JSON và ghi ra file
        public static void WriteJsonToFile(string filename, object obj)
        {
            try
            {
                string json = JsonConvert.SerializeObject(obj,
Formatting.Indented);
                File.WriteAllText(filename, json);
            }
            catch (Exception ex)
            {
                Logger.LogError($"Lỗi ghi file {filename}", ex);
            }
        }
    }

    // Ứng dụng sử dụng các lớp trên
    public class Program
    {
        public static void Main(string[] args)
        {
            var items = new List<OrderItem>
            {
                new OrderItem { Name = "Sản phẩm A", Price = 100, Quantity = 2
},
```

```
                new OrderItem { Name = "Sản phẩm B", Price = 200, Quantity = 1
}
            };

            // Tạo đơn hàng (chỉ chứa dữ liệu và logic tính toán)
            Order order = new Order(items);

            // Gửi email xác nhận (được tách riêng ra để gửi email)
            EmailService emailService = new EmailService();
            emailService.SendOrderConfirmation(order,
"khachhang@example.com");

            // Ghi lại đơn hàng ra file JSON (sử dụng lại hàm tiện ích của
DRY)
            FileHelper.WriteJsonToFile("order.json", order);
        }
    }
}
```

Single Responsibility Principle (SRP):

- **Order and OrderItem:** Only contain data and the logic for calculating the total amount.
- **EmailService:** Solely responsible for sending order confirmation emails. Any changes related to how emails are sent only need to be adjusted here.
- **Logger:** Handles error logging, preventing code for exception handling from being repeated in multiple places.
- **FileHelper:** Manages the logic for writing data to a JSON file, keeping it separate from other classes.

Don't Repeat Yourself (DRY):

- **SendEmail Method:** Encapsulates the logic for sending emails and handling exceptions, making it reusable in other email-sending methods without repeating error-handling code.
- **FileHelper.WriteJsonToFile:** A utility function for writing JSON to a file, which avoids repeating the code for converting and writing data across multiple locations.
- **Logger:** Provides consistent error logging in a maintainable way.

D> Conclusion

- Key Takeaways:

    - `SRP:` Each class should have one and only one responsibility. This improve code clarity, maintainability and testability
    - `DRY:` Avoid repeating code. Encapsulate common logic into reusable functions or classes, reducing the potential for errors and making codebase more maintainable.
- Final Thoughts:
    - Adopting SRP and DRY helps in building robust and scalable applications. They are part of a larger family of principles (such as SOLID) that encourage developers to write cleaner and more maintainable code. In practice, applying these principle leads to a more modular design, easier debugging and more straight forward enhancements.
    - This report provides a detailed explanation of SRP and DRY along with practical `C#` examples. These guidelines serve as a foundation for writing better software and are invaluable as projects scale in complexity.