Ejercicios de Principios de Diseño (KISS, DRY, YAGNI, SOLID)

Instrucciones Generales

1. Crea una solución que contenga una aplicación **de consola en C#** por cada uno de los principios a implementar.
2. Cada ejercicio debe estar en su propio archivo y namespace.
3. Refactoriza el código para cumplir con el principio correspondiente.
4. Sube tu solución a **GitHub** y organiza el código en carpetas.

Ejercicio 1: Aplicando KISS (Keep It Simple, Stupid)

Enunciado:

Un restaurante necesita calcular el total a pagar por los clientes. Se deben sumar los precios de los platos y agregar una propina opcional. Actualmente, el código es innecesariamente complicado y difícil de entender.

Objetivo:
Refactoriza el código para aplicar **KISS**, eliminando la complejidad innecesaria y haciéndolo más claro y fácil de entender.

Código inicial (No sigue KISS):

```csharp
public class RestaurantBill
{
    public decimal CalculateTotal(decimal[] prices, decimal? tipPercentage)
    {
        decimal total = 0;
        for (int i = 0; i < prices.Length; i++)
        {
            total += prices[i];
        }

        if (tipPercentage.HasValue)
        {
            total += total * (tipPercentage.Value / 100);
        }
        else
        {
            total += total * 0.10m; // Propina por defecto del 10%
        }

        return total;
    }
}
```

Requerimientos:

- El usuario ingresará los precios de los platos (separados por comas).
- Se preguntará si desea agregar una propina personalizada o usar la predeterminada del **10%**.
- El sistema debe calcular y mostrar el total a pagar.

Ejemplo de ejecución:

Ingrese los precios de los platos (separados por comas): 200,150,300

¿Desea agregar una propina personalizada? (s/n): n

Total a pagar (con propina del 10%): 715

Ejercicio 2: Aplicando DRY (Don't Repeat Yourself)

Enunciado:

Una empresa necesita calcular el salario de sus empleados de manera eficiente. Actualmente, hay código duplicado en los cálculos de impuestos y bonificaciones para empleados de tiempo completo y medio tiempo.

Objetivo:
Refactoriza el código para aplicar **DRY**, eliminando la duplicación y reutilizando la lógica común en un método.

Código inicial (No sigue DRY):

```csharp
public class Payroll
{
    public decimal CalculateSalaryForFullTime(decimal baseSalary)
    {
        decimal tax = baseSalary * 0.18m;
        decimal bonus = baseSalary * 0.05m;
        return baseSalary - tax + bonus;
    }

    public decimal CalculateSalaryForPartTime(decimal hourlyRate, int hoursWorked)
    {
        decimal salary = hourlyRate * hoursWorked;
        decimal tax = salary * 0.18m;
        decimal bonus = salary * 0.05m;
        return salary - tax + bonus;
    }
}
```

Requerimientos:

- El usuario ingresará el tipo de empleado (**"1" para tiempo completo, "2" para medio tiempo**).
- Para empleados de **tiempo completo**, se ingresará el salario base.
- Para empleados de **medio tiempo**, se ingresará el sueldo por hora y las horas trabajadas.
- Se calculará el salario neto aplicando un **18% de impuestos** y un **5% de bono**.

Ejemplo de ejecución:

Seleccione el tipo de empleado (1: Tiempo completo, 2: Medio tiempo): 1

Ingrese el salario base: 50000

Salario neto después de impuestos y bono: 43250

Ejercicio 3: Aplicando YAGNI (You Aren't Gonna Need It)

Enunciado:

Un sistema de gestión de productos permite agregar y eliminar productos. Sin embargo, el código contiene un método para generar reportes que aún no es necesario.

Objetivo:
Refactoriza el código para eliminar la funcionalidad innecesaria y aplicar **YAGNI**.

Código inicial (No sigue YAGNI):

```
public class ProductService
{
    public void AddProduct(string name, decimal price)
    {
        Console.WriteLine($"Product {name} added with price {price}.");
    }

    public void DeleteProduct(int productId)
    {
        Console.WriteLine($"Product {productId} deleted.");
    }

    public void GenerateProductReport() // Este método no es necesario ahora
    {
        Console.WriteLine("Generating product report...");
    }
}
```

Requerimientos:

- El usuario podrá **agregar productos** ingresando su nombre y precio.
- Podrá **eliminar productos** ingresando el ID.
- El método innecesario **GenerateProductReport()** debe ser eliminado.

Ejemplo de ejecución:

Seleccione una opción:

1. Agregar producto

2. Eliminar producto

Ingrese el nombre del producto: Laptop

Ingrese el precio: 750

Producto 'Laptop' agregado con éxito.

Ejercicio 4: Aplicando SOLID

Enunciado:

Una empresa necesita un sistema de notificaciones que pueda enviar correos electrónicos y SMS, pero la clase actual viola el **Principio de Responsabilidad Única (SRP)** porque también maneja el registro de logs.

Objetivo:
Refactoriza el código dividiendo responsabilidades en diferentes clases para cumplir con **SRP (Single Responsibility Principle)** dentro del marco de **SOLID**.

Código inicial (No sigue SOLID - SRP):

csharp

```csharp
public class NotificationService
{
    public void SendEmail(string email, string message)
    {
        Console.WriteLine($"Sending Email to {email}: {message}");
    }

    public void SendSMS(string phoneNumber, string message)
    {
        Console.WriteLine($"Sending SMS to {phoneNumber}: {message}");
    }

    public void LogNotification(string message)
    {
        Console.WriteLine($"Logging notification: {message}");
    }
}
```

Ejemplo de ejecución:

Seleccione el tipo de notificación:

1. Email

2. SMS

Ingrese el mensaje: Bienvenido a nuestro servicio.

Enviando Email: Bienvenido a nuestro servicio.

Notificación registrada en logs.