GUÍA DE BUENAS PRÁCTICAS Y CÓDIGO LIMPIO

EQUIPO DE TECNOLOGÍA LOGYCA

Liberación 1.0, Ratificado, Junio 2022

TABLA DE CONTENIDO

Buenas prácticas y código limpio
Best practices and clean code

Objetivo: definir un estándar básico de desarrollo para nuestro equipo LOGYCA que permita hablar un mismo idioma, de manera simple, fácil de entender, fácil de mantener, y con esto garantizar un grado de calidad considerable en nuestras creaciones.

Mitos que no son buenas prácticas

- Reducir el código al mínimo
- Comentar todo
- Utilizar regions para ordenar
- Utiliza nomenclatura húngara
- Manejar patrones de diseño en toda la arquitectura

Buenas prácticas

se refieren a los estándares que ya existen dentro del código para garantizar:

- Rendimiento
- Lectura
- Tamaño o líneas de código
- Mantenibilidad
- Eliminar deuda técnica
- Seguridad

Cuando hacemos refinamiento o refactoring de código, sintaxis o arquitectura va enfocado en mejorar los aspectos anteriores.

Consideraciones:

Nombramiento / Naming

Variables

Mal:

```
int d;
```

Bien:

```
int DaySinceModification;
```

Métodos

Mal:

```
public List<Users> getUsers()
```

Bien:

```
public List<Users> GetActiveUsers()
```

Clases

Mal:

```
public  class ClassUser2
```

Bien:

```
public class User
```

Reduce código

Antes:

```
if(active == true)
{
    return CurrentActiveUsers;
}
```

Después:

```
if(active) return CurrentActiveUsers;
```

Antes:

```
foreach ( var item in Users )
{
    if(item.Name == UserName) return true;
}
```

Después:

```
return Users.Any(item => item.Name == UserName);
```

Utiliza las mejoras de C#

Antes:

```
string = “El resultado es “ + result;
```

Después:

```
string = $”El resultado es {result}”;
```

Antes:

```
if(CurrentDate != null)
{
    if(CurrentDate.Year > 20) return true;
```

```
    }
```
Después:
```
    if(CurrentDate?.Year > 20) return true;
```

Evita muchos parámetros (máximo 4 incluyendo el parámetro opcional)

Antes:
```
    public void CreateUser(string name, string lastname, int Id, string phone)
```
Después:
```
    public void CreateUser(User newUser)
```

Clases con una sola responsabilidad

Antes:
```
    public void SendEmailToListOfClients(string[] clients)
    {
        foreach(var client in clients)
        {
            var clientRecord = db.Find(client);
            if(clientRecord.IsActive())
            {
                Email(client);
            }
        }
    }
```
Después:
```
    public void SendEmailToListOfClients(string[] clients)
    {
        var activeClients = Clients.GetActiveClients(clients);
        foreach(var client in activeClients)
        {
            Email(client);
        }
    }
    public List<Client> GetActiveClients(string[] clients) => return db.Find(clients).Where(s => s.Status ==
    “Active”);
```

Encapsular Condiciones

Antes:
```
    if(article.state == “published”)
    {
        //...
    }
```

Después:

```
if(article.IsPublished())
{
        //...
}
```

Comentarios (Comentar solo aquellas cosas que nos ayudan a explicar para que se usa si no es suficiente con el nombre de la variable o método)

Evitar en lo posible cosas como está:

```
//The User Class.
public  class User
{
        //...
}
```

Evitar Try Catch

No es recomendable utilizar excepciones para gestionar el flujo normal de nuestras aplicaciones

Ejemplo:

```
[Benchmark]
public int ConversionOk() => return ConvertWithTryCatch("1");

[Benchmark]
public int ConversionError() => return ConvertWithTryCatch("hola");

private int ConvertWithTryCatch(string value)

{

        try {

                return Convert.ToInt32(value);

        } catch (Exception)

        {

                return 0;

        }

}
```

cuando se produce la excepción, en un escenario simple como el que estamos planteando en la prueba, el tiempo de ejecución es mil veces superior:

Method	Mean	Error	StdDev	Gen 0	Gen 1	Gen 2	Allocated
ConversionOk	13.42 ns	0.076 ns	0.071 ns	-	-	-	-
ConversionError	13,895.49 ns	43.435 ns	40.629 ns	0.0610	-	-	536 B

Principios SOLID

- **S**ingle responsability (responsabilidad única)
- **O**pen-closed (abierto para extensiones – cerrado para cambios)
- **L**iskov substitution (substitución …)
- **I**nterface segregation (segregación de interfaz)
- **D**ependency inversion (inversión de dependencia)

Single responsability (responsabilidad única)

Este principio indica que cada clase o función debe realizar solamente una actividad.

Por ejemplo. el método insertar de una clase categoría solo debe insertar no debe tener parámetro por ejemplo acción el cual indique que la función debe insertar o modificar una categoría. O una clase que por ejemplo guarde en base de datos una factura e imprima una factura, se debe tener 2 clases una para comunicarse con la base de datos y otra para imprimir.

Open-closed (abierto para extensiones – cerrado para cambios)

Tú código debe ser abierto y extendido por otros programadores para adecuarlos y debe ser cerrado para que no necesites cambiar tu código para adecuarlo a las necesidades de los demás programadores.

Por ejemplo, puedes tener una clase que calcule los impuestos de acuerdo al tipo de producto. Unos productos tienen un impuesto donde debes pagar el 16% del valor del producto como impuesto, otro impuesto puede ser de forma fija por ejemplo cada que compres el producto debes pagar $4.00 de impuesto. El código podría ser el siguiente:

¿Cómo conseguir un mejor código?

- Revisar el código con el equipo / CodeReview
- Definir estándares
- Estudiar la sintaxis del lenguaje
- Ser cuidadoso con los nombramientos y ortografía
- Escribir código en inglés
- Apoyarse de los IDEs

Referencias:

1. Libro 'Código Limpio' de Robert C. Martin
2. https://www.espai.es/blog/2021/11/buenas-practicas-en-c/
3. https://www.youtube.com/watch?v=8eejKMW4S4M
4. https://www.variablenotfound.com/2020/03/usar-trycatch-es-malo-para-el.html#:~:text=Por%20tanto%2C%20atendiendo%20a%20estos,si%20no%20se%20producen%20excepciones.&text=Aunque%20con%20una%20importante%20diferencia%20de%20velocidad%20entre%20.
5. https://abi.gitbook.io/net-core/3.-servicios-rest/4.7-refactorizando-tu-codigo