ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Grado

Aprendiendo a aplicar Código Limpio a través de un ejercicio práctico.
(Learning to apply Clean Code through a practical exercise.)

Para acceder al Título de

Graduado en

Ingeniería de Tecnologías de Telecomunicación

Autor: Juan Adrián Pérez Lagunas

Marzo 2023

TÍTULO	**Código Limpio: Cómo aprenderlo a través de un ejercicio práctico**			
AUTOR	**Juan Adrián Pérez Lagunas**			
DIRECTOR	**Juan Antonio Saiz Ipiña**			
TITULACIÓN	**GRADUADO EN INGENIERIA DE TECNOLOGIAS DE TELECOMUNICACIÓN**	**FECHA**	05-2023	**TOMO I DE I**

Contenido

Agradecimientos: 6

Introducción / Motivación 6

Especificación 7

Estado del Arte 8

Modelos de Arquitectura 8

Arquitectura en capas 8

Capa de presentación: 9

Capa de negocio: 9

Capa de acceso a datos: 9

Arquitecturas Limpias 9

Principios SOLID 11

Principio de responsabilidad única (Single responsability principle): 11

Principio de sustitución de Liskov (Liskov substitution principle): 13

Principio de segregación de la interfaz (Interface segregation principle): 14

Principio de inversión de la dependencia (Dependency inversion principle): 15

Código Legacy 16

Errores de código 17

A. Código duplicado: 17

B. Métodos demasiado largos: 17

C. Clases demasiado largas: 18

D. Largas Listas de parámetros: 18

E. Campos temporales: 19

F. Falta de convecciones de estilo: 19

G. Convecciones de estilo propias: 19

Errores de estrategia 20

A. Falta de respeto a la estructura de capas: 20

B. Carencia de tests: 20

C. Uso de framework propio a nivel de empresa: 20

D. Falta de respeto a los principios SOLID: 20

Testing Automatizado 20

Pruebas Unitarias 21

Pruebas de integración 21

Pruebas de funcionales 21

Test Driven Development (TDD) 22
xUnit.Net 22
Hechos 23
Teorías 24
FluentAssertions 24
Reglas de estilo de estándar 26
Nombres de identificadores 27
Reglas de nomenclatura 27
Convenciones de nomenclatura 27
Convenciones de nomenclatura 27
Convenciones de Diseño 28
Desarrollo 28
Ejercicio formador: 28
Parte 1 28
Descripción 28
Inputs-Outputs 28
Ejemplo de resolución 29
Recurring 29
Objetivo 30
Problemas Esperados 30
Parte 2 31
Descripción 31
Inputs-Outputs a añadir 31
Ejemplo de resolución 31
Recurring 31
Recurring weekly 31
Objetivo 32
Problemas Esperados 32
Parte 3 33
Descripción 33
Inputs-Outputs a añadir 33
Recurring monthly 33
Ejemplo de resolución 33
Recurring Monthly 33
Problemas Esperados 34
Parte 4 34

Descripción 34
Objetivo 34
Problemas Esperados 34
Posible extensión del trabajo: 34
Tabla de Ilustraciones: 36
Referencias: 37

Agradecimientos:

Quiero expresar mi más sincero agradecimiento por la ayuda y orientación recibidas durante la redacción de mi proyecto de fin de grado.

En primer lugar, quiero agradecer a mi director de proyecto, Juan Saiz Ipiña, por su dedicación y orientación en todo momento. Su vasta experiencia en el campo me ha proporcionado una perspectiva única y valiosa que me ha permitido mejorar mi trabajo y llevarlo a un nivel superior.

También quiero agradecer a la empresa Semicrol S.L., por haberme brindado la oportunidad de llevar a cabo este proyecto y por su colaboración constante. Gracias a su respaldo, he tenido la posibilidad de aplicar mis habilidades y conocimientos en un entorno profesional.

Asimismo, quiero agradecer a mi tutor académico, Pablo Sánchez Espeso, por su asesoramiento técnico en la elaboración de mi proyecto. Sus comentarios y sugerencias han sido muy valiosos para definir mi enfoque y encontrar la mejor vía a la consecución final del trabajo.

Además, me gustaría expresar mi gratitud a la Universidad de Cantabria y a la Escuela Técnica Superior de Ingenieros de Telecomunicación por haberme brindado la formación necesaria para llevar a cabo este proyecto.

Por último, pero no menos importante, quiero agradecer a mi familia y pareja por su paciencia y apoyo incondicional durante todo el proceso. Su amor y comprensión han sido fundamentales para mi éxito en este proyecto y en mi carrera académica.

A todos ustedes, gracias por su dedicación y compromiso en el éxito de mi proyecto. Sin su ayuda, respaldo y orientación, no habría sido posible alcanzar este logro.

Introducción / Motivación

EL presente trabajo nace de la experiencia profesional en la que una revisión profunda de código legacy ha sido aplicada. Cuando una empresa se propone la actualización y transformación de su core, las circunstancias suelen ser poco favorecedoras debido al gran peso de la tradición y formas de programar.

Es aquí en donde las técnicas y metodologías para lograr el cambio se convierten en un factor clave. Para lograr que un equipo de programadores, que supera la veintena, sea consciente y asuma el cambio, es preciso establecer una estrategia que en ocasiones puede resultar temporalmente involucionista y ardua. Todo se basa en la formación y la asunción de los nuevos principios de la programación avanzada y limpia, sobre estos conceptos profundizaremos a lo largo del trabajo.

En el presente documento se describen minuciosamente las etapas por las que una compañía ha de transitar para lograr una profunda actualización. Comenzaremos por establecer qué fundamentos en la programación actual son irrenunciables. A continuación, trataremos de identificar los errores habituales que se han cometido en el código legacy al que no hemos enfrentado y que han de ser corregidos progresivamente, sin la opción de empezar de nuevo.

Esta visión es muy interesante pues se abordan problemas y se detectan anomalías comunes y muy habituales en la mayor parte de aquellos desarrollos de software que provienen de una larga y extensa vida sin vigilancia.

A partir de este trabajo de identificación se propone una visión de como inyectar aquellas mejoras para reconducir las malas prácticas, por ello se describen los conocimientos más sobresalientes en la nueva era de la programación.

EL trabajo en su parte final plantea un ejercicio práctico con una intuición pedagógica concreta. Partiendo de un planteamiento sencillo su resolución, que iremos describiendo, conlleva pasar por todas y cada una de las etapas de los principios SOLID [1]que serán descritos previamente en el trabajo.

Este trabajo surge de la necesidad de formar adecuadamente a los desarrolladores de una manera acorde a los planteamientos y exigencias actuales que demanda el desarrollo de software hoy en día. Este ejercicio ha sido diseñado por varios motivos, el primero es eliminar todo tipo de dependencias que pueda tener el desarrollador fruto de proyectos antiguos en los que haya podido trabajar. Entender que es muy fácil que los proyectos se compliquen, sobre todo cuando un sistema escala. Ver que es muy fácil cometer errores y que lo aparentemente sencillo puede tornarse complicado y tedioso si no se realiza de la manera adecuada.

Este reto no consiste en el que desarrollador complete perfectamente el ejercicio si no en que se enfrente al problema, diseñe, se dé cuenta de los errores cometidos, rediseñe, rehaga las veces que sean necesarias y aprenda de cada decisión que tome.

Especificación

Como se ha comentado anteriormente el trabajo pivota en la realización de un ejercicio práctico y singular que permitirá abordar un amplio espectro de principios irrenunciables en programación de última generación. A continuación, se describen una serie de premisas y normas básicas a cumplir, que ayudarán de forma progresiva a comprender la intención pedagógica del planteamiento.

- Es independiente a la tecnología utilizada. Puede desarrollarse en el lenguaje, plataforma e ide que se desee. En este documento se aborda desde C# y Visual Studio, pero es igual de valido para el resto de lenguajes y plataformas.
- Tiene que poder probarse a través de una interfaz, programa de consola o cualquier medio que el participante considere.
- El participante puede abandonar cuando lo desee, es él el que decide si le gusta, motiva o le aporta valor. Pero uno de los objetivos es el esfuerzo ante la desmotivación provocada por la falta de resultados.
- Se puede buscar información, copiar código similar y adaptarlo, buscar aplicaciones similares para basar la solución, pero hay que tener en cuenta de que el ejercicio va a escalar.
- El ejercicio es básicamente individual pero está abierto al trabajo en equipo por parejas.

[1] SOLID: En ingeniería de software, SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion) es un acrónimo mnemónico introducido por Robert C. Martin a comienzos de la década del 2000 que representa cinco principios básicos de la programación orientada a objetos y el diseño.

- El instructor decidirá si el alumno puede pasar a la siguiente fase o si debe mejorar su fase actual.

Estado del Arte

Modelos de Arquitectura

Arquitectura en capas

A la hora de diseñar software hay que tener en cuenta diferentes factores, por un lado el avance de las tecnologías en el propio desarrollo de software y por otro; quizás el más importante, la posible escalabilidad a largo plazo del producto. En el mundo de la programación hay un término que define en gran medida la estrategia de lo que debe ser y será el producto a gran escala: estamos hablando de la arquitectura.

Desarrollar una buena arquitectura es fundamental de cara a no tener en el futuro problemas que obliguen a complejas refactorizaciones. Sobre este tema hay mucho escrito, pero quizás el concepto más extendido en la actualidad es la construcción ordenada mediante una arquitectura por capas.

Esta arquitectura aglutina en tres grandes bloques conceptos y rangos de código con el fin de independizar y lograr una mayor comprensión de lo que hace el código y como lo presenta.

Las tres capas se denominan: Presentación, Negocio o Dominio y Acceso a Datos o Persistencia. Su ordenación es jerárquica de tal manera que cada capa solo podrá acceder a la capa inmediatamente inferior. Este desacoplamiento entre capas va a permitir que si en un futuro nuestra aplicación debe cambiar de interfaz de usuario o de proveedor de base de datos lo podrá hacer sin grandes quebraderos de cabeza.

Veamos esta jerarquía a continuación

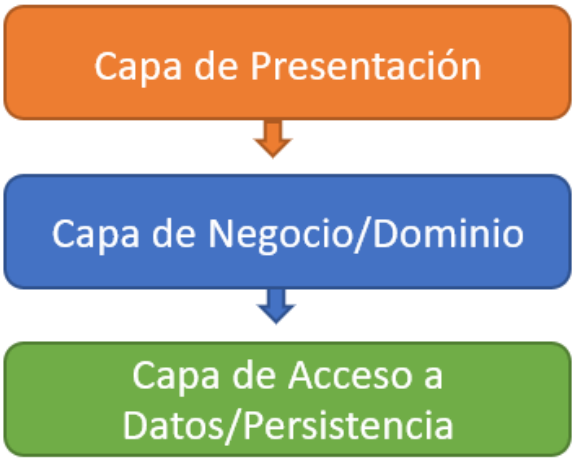

Ilustración 1: Estructura de capas

Veamos a continuación los contextos en los que interviene cada una de las capas señaladas:

Capa de presentación:

Es la responsable de manejar la interfaz gráfica de usuario, una aplicación de Windows Forms, web, móvil, etc… Es la capa donde se localiza el código encargado de interactuar con el usuario, se encarga de recopilar la información de entrada necesaria para efectuar los procedimientos declarados en la capa de negocio y mostrar la respuesta obtenida al usuario.

Capa de negocio:

Es el núcleo de la aplicación, en ella está toda la lógica que define el funcionamiento del programa. En ella se establecen todas las normas que debe cumplir el programa, recibe de la capa de presentación la información sobre las peticiones que realiza el usuario y se encarga de procesarlas, si es necesario comunicarse con la capa de datos para obtener o persistir datos se encargará también de ello, y devolver la información de los resultados de los procedimientos como respuesta a la capa de aplicación para que se los presente al usuario.

Capa de acceso a datos:

Es la capa proveedora de datos de persistencia. Es la capa donde se establece la conectividad con la base de datos y se encarga de proveer a la capa de negocio de los métodos necesarios para poder guardar y obtener los datos contenidos en la base de datos.

Arquitecturas Limpias

Si bien ha sido generalizado el modelo donde las capas únicamente podían acceder a la capa inmediatamente inferior, en algunos de los libros más emblemáticos del mundo del software podemos encontrar diversas opiniones y planteamientos diferentes como:

> *"A veces, las capas se organizan de modo que la capa de dominio oculte completamente la fuente de datos de la presentación. Más a menudo, sin embargo, la presentación accede directamente al almacén de datos. Si bien esto es menos puro, tiende a funcionar mejor en la práctica. La presentación puede interpretar un comando del usuario, usar la fuente de datos para extraer los datos relevantes de la base de datos y luego dejar que la lógica del dominio manipule esos datos antes de presentarlos en el cristal"* (Fowler)

o el planteamiento Domain-Driven Design (Evans, 2003).

Esta estructura de capas ha ido evolucionando con el tiempo y de todas las evoluciones, para este trabajo, es imperativo nombrar la Arquitectura Limpia (C.Martin, 2012). C.Martín propone la creación de una arquitectura estándar apoyándose en algunas ideas de arquitecturas ya existentes como la de cebolla o la hexagonal incorporando además partes de patrones de diseño como MVC[2] o EBI[3].

---

[2] MVC: Modelo Vista Controlador es un estilo de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.

[3] EBI: Entity Boundary Iterator es un estilo de arquitectura similar a MVC donde el negocio está abstraído del resto y no sabe desde que medio se está accediendo a ellos.

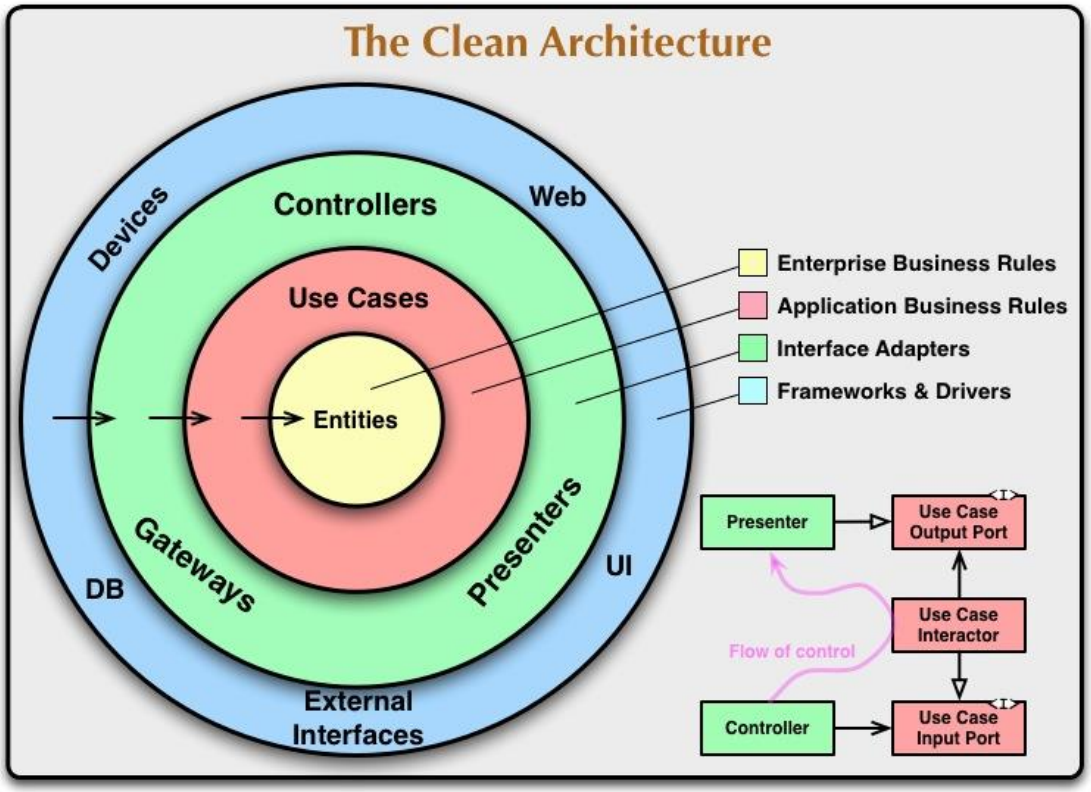

Ilustración 2: The Clean Architecture

Siguiendo esta arquitectura se pueden desarrollar sistemas que cumplan las principales características que debe tener hoy en día un proyecto de software:

- Independencia de frameworks.
- Testable
- Independencia de la interfaz de usuario
- Independencia de la base de datos
- Independencia de cualquier organismo externo.

La regla primordial que hace que esta arquitectura funcione es la Regla de la Independencia, la mejor explicación posible de esta regla es la que da el propio autor

> *"La regla primordial que hace que esta arquitectura funcione es la regla de la dependencia. Esta regla dice que las dependencias del código fuente sólo pueden apuntar hacia adentro. Nada en un círculo interior puede saber nada en absoluto sobre algo en un círculo exterior. En particular, el nombre de algo declarado en un círculo exterior no debe ser mencionado por el código en el círculo interior. Esto incluye funciones, clases, variables o cualquier otra entidad de software con nombre.*
>
> *Del mismo modo, los formatos de datos utilizados en un círculo externo no deben ser utilizados por un círculo interno, especialmente si esos formatos son generados por un marco en un círculo externo. No queremos que nada en un círculo externo afecte a los círculos internos." (C.Martin, 2012)*

En conclusión, debemos de basarnos en una buena arquitectura para beneficiarnos de lo que nos ofrece: bajo acoplamiento, alta cohesión, reutilización, mayor flexibilidad, facilidad de mantenimiento, mayor escalabilidad y agilidad el desarrollo.

Principios SOLID

Si descendemos ya a la tarea de la codificación pura, actualmente existen reglas de obligado cumplimiento. Cuando el código es extenso y se prevé una interacción por diferentes desarrolladores gran parte del éxito del crecimiento radica en que la ordenación de lo programado siga unos principios asumidos por todos y desde la primera línea de código escrita.

Como referencia actual diremos que en el año 2000 Robert C. Martin (Uncle bob), en su libro Agile Software Develpment, Principles, Patterns and Practices, describió 5 principios básicos que deben seguirse en la programación orientada a objetos:

1. **Principio de responsabilidad única (Single responsability principle)**

2. **Principio de abierto/cerrado (Open/closed principle)**

3. **Principio de sustitución de Liskov (Liskov substitution principle)**

4. **Principio de segregación de la interfaz (Interface segregation principle)**

5. **Principio de inversión de la dependencia (Dependency inversión principle)**

Siguiendo estos 5 principios, con cualquier lenguaje orientado a objetos, conseguiremos escribir un código limpio, flexible a los cambios, escalable, reutilizable, testable y fácilmente mantenible.

Veamos a continuación los cinco principios con ejemplos prácticos para una mejor comprensión.

Principio de responsabilidad única (Single responsability principle):

Es el principio de la minimización de la atomización de lo programado. De forma simple se puede decir que los objetos deben hacer únicamente una cosa, y por delegación que los métodos de los objetos deben tener también una única responsabilidad.

Veamos ahora un típico ejemplo en el que un sistema que realiza una orden de compra y luego envía una notificación al cliente

---

```
public class OrderService
{
    private readonly Client _client;

    public OrderService(Client client)
    {
        _Client = client;
    }

    public async Task Add(Order order)
    {
        // Código para crear la orden


        var message = new MailMessage ("sales@admin.com", order.ClientEmail)
        {
            Subject = "Se le asignó una compra",
```

```
            Body = "Estimado,\n Hemos creado su nueva orden de compra .."
        };

        await this.SendCustomerNotification(message);
    }

    public async Task SendCustomerNotification(MailMessage message)
    {
        await _smtpClient.SendMailAsync(message);
    }
}
```

Ilustración 3: Fragmento de Código 1: Ejemplo Single Responsability 1

Si analizamos esta clase veremos que tiene dos responsabilidades, generar la orden de compra y enviar la notificación al cliente, por lo que estaría incumpliendo el principio de responsabilidad única.

Refactorizando de manera correcta el código deberíamos tener dos clases, una encargada de generar la orden de compra y otra encargada del envío de la notificación.

```
public class NotificationService
{
    private readonly Client _client;

    public NotificationService(Client client)
    {
        _client = client;
    }

    public async Task SendEmail(MailMessage message)
    {
        await _smtpClient.SendMailAsync(message);
    }
}

public class OrderService
{
    private readonly NotificationService _notificationService;

    public OrderService(NotificationService notificationService)
    {
        _notificationService = notificationService;
    }

    public async Task Add(Order order)
    {
        // Código para crear la orden
        var message = new MailMessage ("sales@admin.com", order.ClientEmail)
        {
            Subject = "Se le asignó una compra",
            Body = "Estimado,\n Hemos creado su nueva orden de compra .."
        };
        await this._notificationService.SendEmail(message);
    }
}
```

Ilustración 4:Fragmento de Código 2: Ejemplo Single Responsability 2

Principio de abierto/cerrado (Open/closed principle): Una entidad de software debería estar abierta a extensión, pero cerrada a modificación. Esto quiere decir que deberíamos poder extender la funcionalidad de una clase sin tener que modificar código, únicamente para corregirla. Esto ayudará a que nuestro software escale sin afectar al código ya existente.

Cómo ejemplo típico tenemos una clase que envía una notificación con un método que notifica dependiendo del tipo de la notificación:

```
public async Task Send(List<Notification> notifications)
{
        foreach (var notification in notifications)
        {
                if (notification.Type.Equals("sms"))
                {
                        await SendbySMS(notification.PhoneNumber, notification.Subject);
                }

                if (notification.Type.Equals("email"))
                {
                        await SendbyEmail(notification.Email, notification.Subject);
                }
        }
}
```

Ilustración 5: Fragmento de Código 3: Ejemplo Open/closed 1

Si posteriormente a la creación de esta clase necesitáramos añadir algún tipo de notificación diferente, por ejemplo, WhatsApp, deberíamos de ir al método y modificarlo para añadir un nuevo *if* con la implementación en caso de ser tipo WhatsApp por lo que estaríamos rompiendo el principio *Open/closed*.

Cómo solución a esto lo que se propone es la creación de una interfaz que defina a los tipos y una clase para cada tipo que la cumplan:

```
public interface INotification
{
    Task Notifiy();
}
```

Ilustración 6:Fragmento de Código 4: Ejemplo Open/Closed 2

Ahora el método recibirá la interfaz y solo tendrá que llamar al método de notificación que implementará cada tipo, lo que nos permitirá a posteriori, crear nuevas clases para nuevos tipos que podamos necesitar:

```
public async Task Send(List<INotification> notifications)
{
        foreach (var notification in notifications)
        {
                await notification.Notifiy();
        }
}
```

Ilustración 7: Fragmento de Código 5:Ejemplo Open/Closed 3

Principio de sustitución de Liskov (Liskov substitution principle):

Este principio tiene mucha relevancia y fue introducido por Barbara Liskov en 1988 en la conferencia magistral "*Data abstraction and hierarchy*." La propuesta determina que cada clase que hereda de otra debe poder sustituir a su padre sin necesidad de conocer las diferencias entre ellas. Este principio ayuda y garantiza el uso de la herencia de manera correcta.

Es habitual encontrarnos casos en los que usamos una herencia en la que en alguna clase hija nos quede algún método del padre que no podamos utilizar y que lo dejemos vacío o lancemos

una excepción cuando se utilice, en este caso estaremos rompiendo el principio de sustitución de Liskov ya que no podríamos sustituir al hijo por el padre al no implementar este todos sus métodos y esto podría dar lugar a que la aplicación pueda lanzar esas excepciones o tener un comportamiento inesperado.

Como ejemplo pongamos una clase abstracta animal que tenga los métodos andar, correr y saltar:

```
public class Animal
{
    public virtual void Run() {

    }

    public virtual void Walk() {

    }

    public virtual void Jump() {
    }
}
```

Ilustración 8:Fragmento de Código 6: Ejemplo Liskov 1

A su vez podemos tener varias clases que hereden de Animal, por ejemplo, perro, gato y elefante. En el caso del perro y el gato podremos implementar sin problemas todos sus métodos, pero llegado el caso del elefante resulta que los elefantes no pueden saltar por lo que no podríamos implementar sobre escribir el método saltar para darle funcionalidad y deberíamos dejarlo vacío o lanzar una excepción. Esto rompe el principio de sustitución de Liskov y por lo tanto nos está indicando de que nuestro diseño de abstracciones no es el correcto.

Como solución podríamos tener una clase abstracta Animal con los métodos que tienen en común, andar y correr, y luego otra que herede de Animal que añada el comportamiento de saltar LightAnimal así perro y gato podrían heredar de LigthAnimal y elefante únicamente de Animal. De este modo salvamos el problema y adecuamos el código al principio de sustitución.

Principio de segregación de la interfaz (Interface segregation principle):

"Los clientes de un programa dado solo deberían conocer aquellos métodos que realmente usan, y no aquellos que no necesitan usar" (Wikipedia, s.f.).

Puede ser habitual, en programas que no han seguido los principios *SOLID*, encontrarse con "fat interfaces", interfaces demasiado grandes que pueden obligar a las clases que las implementan a dejar propiedades o métodos sin implementar o lanzando una excepción lo que puede suponer un problema ya que una clase externa podría llamar a esa propiedad o método y al no hacer nada tener un comportamiento inesperado o peor aún, recibir una excepción. Para cumplir este principio será necesario separar la interfaz en interfaces más pequeñas, llamadas "Interfaces de rol ".

Veamos como ejemplo el caso de un reloj simple y un Smartwatch:

Tenemos una interfaz para relojes.

```
public interface IWatch
{
        string GetTime();
        string GetEmailNotifications();
```

```
}
```

Ilustración 9:Fragmento de Código 7: Ejemplo Interface segregation 1

Y queremos implementar un reloj simple que solo da la hora. Al hacerle heredar de la interfaz que hemos implementado para los relojes tenemos que implementarla en su totalidad obligándonos a dejar sin implementación el método que nos da las notificaciones del email.

```
public class SimpleWatch : IWatch
{
        public string GetTime() => DateTime.Now;
        public string GetEmailNotifications() => throw new NotImplementedException();
}
```

Como solución deberemos crear interfaces más pequeñas.

```
public interface IWatch
{
        string GetTime();
}

public interface ISmartWatch: Iwatch
{
        string GetEmailNotifications()
}

public class SimpleWatch : IWatch
{
        public string GetTime() => DateTime.Now;
}
public class SmathWatch : ISmartWatch
{
        public string GetEmailNotifications() => "you have 2 emails"
}
```

Ilustración 10:Fragmento de Código 8: Ejemplo Interface segregation 2

Principio de inversión de la dependencia (Dependency inversion principle):

Este principio define dos reglas maestras que son las siguientes:

1. Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.
2. Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

Los problemas aquí comienzan cuando un módulo depende de otro, ya sea clase, proyecto o incluso framework, las clases que lo necesiten instancian los objetos de los otros módulos y los utilizan.

Este tipo de construcciones generan dependencias en el módulo hacia los otros módulos y no van a permitir realizar cambios de manera sencilla. La forma más sencilla de verlo es cuando realizamos test, imaginemos que tenemos una clase con un método que utiliza un servicio web para obtener unos datos, pero ese servicio web únicamente es accesible desde el entorno del cliente, si queremos hacer un test de esa clase en el entorno de desarrollo el test se romperá porque el servicio web no estará accesible.

```
public class maker
{
```

```
        public static void DoSomething()
        {
                var data = WebService.GetData()
                //DoSomething with data.
        }
}
```

Ilustración 11:Fragmento de Código 9: Ejemplo Dependency inversion 1

Como solución a este problema lo más habitual es utilizar la inyección de dependencias que consiste en la utilización de interfaces para pasar las instancias de los módulos externos como argumento y así no tener dependencia de ninguno de ellos.

```
public interface IWebService
{
        Data GetData()
}

public class maker
{
        public static void DoSomething(IWebService webService)
        {
                var data = webService.GetData()
                //DoSomething with data.
        }
}
```

Ilustración 12:Fragmento de Código 10: Ejemplo Dependency inversion 1

Con esta solución el método *DoSomething* ya no depende de un servicio web en concreto y, en este caso, podremos probarlo en nuestros test sin ningún problema utilizando un objeto *mock* por ejemplo.

La gran virtud de seguir este principio es que lograremos un código muy versátil, que no dependerá de desarrollos externos y que será muy sencillo de actualizar.

Código Legacy

Con lo anterior hemos sentados las bases de cómo ha de programarse en la actualidad. Si a la arquitectura por capas bien definida le añadimos los principios SOLID y además establecemos unas reglas de estilo actuales el producto será de primer nivel. Sin embargo, no siempre nos encontramos con escenarios en los que estas buenas prácticas puedan ser implantadas de forma máxima. Las tecnologías en programación avanzan rápidamente y en muchas ocasiones las compañías de software no pueden realizar las actualizaciones necesarias para estar al día. Esta falta de vigilancia se ve agravada si las conductas a lo largo del tiempo han desatendido los principios de calidad básicos. Esta circunstancia se da más a menudo de lo que uno puede pensar, y en la mayor parte de los casos se debe a que el día a día y las repuesta rápida a cliente deshabilita las buenas prácticas imprescindibles en toda empresa de software moderna.

En este apartado vamos a realizar un repaso a los grandes males que una vez juntos identifican los mayores problemas de un código viejo y desatendido, lo que viene a llamarse código legacy.

El término Legacy Code tiene una definición clara, por lo general se puede identificar como código fuente antiguo, difícil de entender y modificar, a veces heredado de antiguos desarrolladores que ya no forman parte del equipo.

También tenemos definiciones de autores consagrados, "*Legacy Code is code without test*" (Feathers, 2004) o "Legacy code is any code that significantly and continually resists the reflection of on-going requirements change." de Anthony Lauder.

Errores de código

Veamos ahora las patologías más habituales que muestra un código legacy de la mano de Martin Fowler que en su libro "Refactoring Improving the Design of Existing Code" enumera algunos de los errores más comunes que suelen cometer los desarrolladores a la hora de escribir su código y que indican que debe ser refactorizado, aquí algunos de los que consideró más importantes:

A. Código duplicado:

Este es un error muy común y sobre todo si el número de programadores que intervienen en el código es amplio y poco intercomunicado. Tener una visión sistémica de todo el código es una de las claves. Esta visión permitirá adquirir políticas de agrupamiento funcionales que identifiquen las partes de código que hacen lo mismo es una tarea imprescindible y muy necesaria.

Tengamos en cuenta que, si una clase tiene la misma estructura de código en varias partes del código, la limpieza y simplificación nos llevaría a ser extraída y representada por un único método.
Por otro lado, si el código duplicado está en clases hermanas [4](explicar hermanas), debería ser añadida a la clase madre.

Si nos centramos en los métodos y detectamos que existen varios métodos que realizan la misma acción con diferentes algoritmos, lo oportuno será determinar cuál de ellos es el más claro y eliminar el otro. Y si el código duplicado está en clases no relacionadas deberá valorarse el extraer la funcionalidad a una nueva clase e invocarla desde las clases originales. Como puede verse todo radica en evitar la duplicidad y la reutilización intensiva de piezas de código únicas.

B. Métodos demasiado largos:

Pero los problemas no se quedan en que tengamos secciones de código duplicadas, el problema puede ser que los métodos usados presenten una extensión que provoquen una incomprensión permanente a la hora de comprender su función y estructura.

Cuánto más largo es un método, más complicado es de entender. Los métodos de los objetos de una buena aplicación deben ser pequeños, tener únicamente una responsabilidad y contar con un nombre lo más descriptivo posible sobre lo que hacen. Si encontramos un método excesivamente grande la mejor práctica es analizarlo y dividir las funcionalidades en otros métodos más pequeños, pudiendo incluso convertir el principal en orquestador.

```
public static void EnrolStudent(string groupName, int groupNumber, Student student)
{
        var group = new group
        {
                name = groupName,
                number = groupNumber
```

[4] Clases Hermanas: Clases que heredan de una misma clase madre.

```
        };
        if(student.language == Languages.Spanish)
        {
                group.language == Languages.English;
        }
        var enrolment = new Enrolment(student,DateTime.Now);
        enrolment.Save();
        group.AddEnrolment(enrolment);
        group.Save();
}
```

Ilustración 13:Fragmento de Código 11: Código Legacy 1

Refactoring:

```
public static void EnrolStudent(string groupName, int groupNumber, Student student)
{
        var group = CreateGroup(groupName,groupNumber,student);
        var enrolment = CreateEnrolment(student);
        group.AddEnrolment(enrolment);
        group.Save();
}
private static group CreateGroup(string groupName, int groupNumber, Student student)
{
        var group = new Group
        {
                name = groupName,
                number = groupNumber,
        };
        if(student.language == Languages.Spanish)
        {
                group.language == Languages.Spanish;
        }else
        {
                group.language == Languages.English;
        }
        return group;
}
private static Enrolment CreateEnrolment(Student student)
{
        var enrolment = new Enrolment(student, DateTime.Now);
        enrolment.Save();
        return enrolment;
}
```

Ilustración 14:Fragmento de Código 12: Código Legacy 2

C. *Clases demasiado largas:*

En el mismo orden que los métodos, las clases demasiado largas son muy complicadas de entender y manejar en un mantenimiento ordinario. Además, este tipo de clases por lo general no cumplen el principio de responsabilidad única, algo absolutamente necesario y marcado por los principios de calidad estándar en nuestros días.

Una de las labores de corrección ante este tipo de problemas es la sucesiva subdivisión de las clases en unas nuevas clases y subclases mucho más manejables y comprensibles. También las clases con muchas variables pueden ser problemáticas y a veces se pueden unificar dentro de nuevas entidades. La solución de este problema a través de tareas de refactorización es compleja y hasta cierto punto arriesgada, pues la interdependencia en el código Legacy suele ser una constante.

D. *Largas Listas de parámetros:*

Debido a que los datos globales suelen ser problemáticos se puede tender a largas listas de parámetros en métodos con todo lo que necesitan para su rutina, pero esto sean más

complicado de entender. En programación orientada a objetos estos parámetros se pueden encapsular dentro de una clase y el método pasar de recibir los datos que necesita a recibir el objeto y coger de él lo necesario.

```
public void DoSomething(string param1, string param2, int param3, int param4, bool param5,
bool param6)
{
        ...
}
```

Refactoring:

```
public void DoSomething(DoSomethingData data)
{
        ...
}

public class DoSomethingData()
{
        public param1 {get;set;}
        public param2 {get;set;}
        public param3 {get;set;}
        public param4 {get;set;}
        public param5 {get;set;}
        public param6 {get;set;}
}
```

Ilustración 15:Fragmento de Código 13: Código Legacy 3

E. Campos temporales:

En ocasiones se pueden ver variables de instancia que no siempre se usan. Esto puede dar grandes quebraderos de cabeza a la hora de entender el código ya que generalmente esperamos que un objeto necesite de todas sus variables para funcionar. Puede ser complicado averiguar cuando y por qué se usan esas variables.

F. Falta de convecciones de estilo:

En el desarrollo de una aplicación es muy común que participen varios programadores y, además, que en su ciclo de vida algunos salgan y entren otros nuevos en el proyecto. Si cada programador programa siguiendo sus propios gustos nos encontraríamos con un código muy difícil de leer donde en unas clases nos encontraríamos por ejemplo las variables privadas con mayúsculas, en otra comenzando por _ y en otra en minúscula. Establecer normas de estilo es imprescindible para que todos los desarrolladores “hablen el mismo idioma” dentro del aplicación.

G. Convecciones de estilo propias:

Ligado a lo anterior, es muy común, que encontremos empresas que tengan sus propias normas de estilo. Esto, a pesar de que puede parecer una buena idea, no lo es. Cuando llega un integrante nuevo al equipo, debe perder tiempo en aprender las normas propias de estilo de la empresa cuando existen normas de estilo estandarizadas para cada lenguaje que harían que cualquiera pueda desarrollar y entender nuestra aplicación sin necesidad de aprendizaje nuevo.

Errores de estrategia

Lo anterior define un grupo de problemas acumulados por la desatención en la acción de la programación, son problemas cercanos a la indisciplina de las buenas prácticas básicas. Pero a todo lo anterior el escenario de un código legacy pude verse agravado por una falta de visión o estrategia. En este caso estaríamos hablando de arquitectura, de herramientas y de búsqueda de la calidad da través de los test. Así podemos enumerar lo siguiente:

A. *Falta de respeto a la estructura de capas:*

No siempre es trivial determinar qué debe ir en cada capa y por ello se pueden cometer errores que no respeten la correcta estructura de capas de nuestra aplicación haciendo que a posteriori pueda ser una tarea muy tediosa realizar cualquier cambio en la misma. *Queries*[5] de acceso a datos en clases de negocio o incluso de presentación o lógica de negocio en clases de presentación son dos de los errores más comunes que se pueden cometer.

B. *Carencia de tests:*

Un código sin testar es un código en el que no se puede cambiar nada teniendo la seguridad de que nuestro cambio no va a afectar a otras partes de la aplicación. Todo código debe ser testado, hoy en día existen varios sistemas muy buenos de automatización de pruebas y debemos emplearas para codificar pruebas tanto unitarias como de integración de nuestra aplicación. Probar a mano es ineficiente e inefectivo.

C. *Uso de framework 6propio a nivel de empresa:*

Otro de los errores más comunes que se comenten es la creación de un framework propio dentro de las empresas y su uso para el desarrollo de todas las aplicaciones de esta. Cada proyecto tiene sus propias características y necesidades, si un mismo código se convierte en la base de todos cuando algún proyecto necesite de alguna particularidad está deberá servir para todos los demás puesto que este código es común. Esto se vuelve especialmente problemático si necesitamos que una aplicación esté o use tecnologías más modernas o anticuadas que otras y no haya compatibilidad entre las mismas.

D. *Falta de respeto a los principios SOLID:*

Los principios SOLID se han desarrollado para siguiéndolos se consiga un código fácil de mantener y ampliar con el tiempo. No seguirlos hará que tengamos que perder mucho tiempo en entender, desacoplar y refactorizar nuestro código, pudiendo llegar incluso a decidir rehacerlo por completo ante la imposibilidad de realizar una correcta refactorización.

Testing Automatizado

Es aquí en donde nos encontramos con una labor ardua pero absolutamente necesaria, la confianza de un programador en que su construcción no va a fallar ha de ir acompañada de la

[5] Queries: Consultas.

[6] Framework: Esquema o marco de trabajo que ofrece una estructura base para elaborar un proyecto con objetivos específicos

validación robotizada que garantice procesalmente su diseño. En el desarrollo de software siempre ha existido la necesidad de probar que todo funciona de la forma esperada.

Para testar una aplicación se deben definir pruebas sobre los casos de uso funcional de la misma y aquellas pruebas que defina el desarrollador para probar que el código es robusto frente a los errores. Para el testeo manual de estas pruebas lo deseable es que, a parte del propio desarrollador, sean pasadas por otros desarrolladores ajenos al proyecto y por algún departamento ajeno al desarrollo, lo cual implica un montón de tiempo.

Otra desventaja que tiene el testeo manual es que cuando la aplicación escala, cada vez que se realiza algún cambio, se debería de volver a probar que no hemos tocado nada que afecte al resto de la aplicación, es decir, volver a pasar las pruebas, que cada vez serán más y cada vez llevarán más tiempo.

Para solucionar esta problemática surgieron las pruebas automatizadas, que hacen trabajar a la máquina simulando aquellos casos y condiciones presupuestadas por nosotros y que deben ser infalibles pase lo que pase. Esto tiene la ventaja de que las pruebas quedan escritas en código y se asegura que todas serán evaluadas en cada compilación. Pero el gran poder de esta técnica laboriosa es defendernos ante cambios ya que podemos ejecutarlas y verificar que no hemos afectado al funcionamiento de la aplicación y todo ello a gran velocidad.

Las pruebas de test pueden ser de amplio espectro y además las hay de diferentes Tipologías. Veamos los tres tipos de pruebas más importantes:

Pruebas Unitarias

Las pruebas unitarias son aquellas que prueban componentes o métodos de software individuales, conocidos como “unidad de trabajo”. Probaran una única clase, comprobando sus funcionalidades de forma muy concreta. Que un método lance una excepción si un argumento es NULL o que devuelva lo esperado según los argumentos, son posibles casos de pruebas unitarias. La intensidad y densidad de estas pruebas es clave puede se debe tender a cubrir el 100% de los casos que puedan ocurrir en una clase.

Pruebas de integración

Las pruebas de integración son aquellas en las que se prueba la capacidad de dos o más unidades de trabajo para funcionar juntas. Se verifica que la integración de las diferentes clases y servicios que componen nuestra aplicación funcionan correctamente. Con este tipo de pruebas se busca verificar la correcta conexión de todas las capas de nuestra aplicación. Así probaremos la coherencia de una capa de datos y su conexión a la base de datos real con los datos que precisemos en el caso bajo test.

Pruebas de funcionales

Las pruebas funcionales son aquellas en las que se prueba la funcionalidad de la aplicación completa. Son aquellas que verifican que la aplicación cumple con los requisitos especificados y que hay fiabilidad en cuanto a rendimiento y carga.

Un ejemplo básico podría ser la comprobación de que cuando un usuario mete su usuario y contraseña correctamente accede correctamente a la aplicación plenamente identificado.

Test Driven Development (TDD)

En 1992, Kent Beck, en su libro "Test-Driven Development: By Example" describió una metodología de desarrollo basada en el testing que consiste básicamente en realizar los test antes del código.

Podrían enumerarse los siguientes pasos:

- Estudio de la funcionalidad a desarrollar dividiéndola en pequeñas tareas.
- Diseño de los test que van a garantizar el funcionamiento de dicha funcionalidad
- Desarrollo de los test.
- Desarrollo del código de la funcionalidad cumpliendo los test.
- Refactorización del código cumpliendo de calidad.

Como principales ventajas podemos enumerar las siguientes:

- Confianza en el correcto funcionamiento del código.
- Menor tiempo
- Se evita la creación de código innecesario.
- Metodología Ágil, flujo continuo de tareas, análisis y correcciones.
- Refactorización que permite cumplir estándares de calidad antes de la finalización del desarrollo.

En cuanto a las principales desventajas:

- Es difícil de implementar para el desarrollo de interfaces de usuario.
- Puede generar una falsa sensación de seguridad ya que pasen todos los test no quiere decir que no se nos hayan quedado posibles errores por identificar.
- El desarrollo de los test puede complicarse si la aplicación tiene llamadas a APIs externas o acceso a datos.
- La curva de aprendizaje es bastante pronunciada lo que puede conducir a pérdida de productividad.

xUnit.Net

Se trata del framework de testing más utilizado hoy en día y es OpenSource.

Dentro de este apartado vamos a explicar el funcionamiento básico necesario para la ejecución del ejercicio de formación objeto de este trabajo, para más información y documentación visitar la página web del proyecto: https://xunit.net/

Su funcionamiento es bastante sencillo, la clase de pruebas debe ser pública, cada método será una prueba e irá adornado con el atributo [Fact] si es un hecho o [Theory] si es una teoría, xUnit.net buscará los métodos que contengan esos atributos y tomará como de testing sus clases, ejecutándolos de manera estática si son estáticos o generando una instancia que al finalizar será liberada.

Los test en Xunit.Net se dividen en tres partes:

- Arrange: Definición de variables necesarias para los test.
- Act: Ejecución de métodos a testar y obtención de resultados
- Assert: Comprobación de que el resultado es el esperado. Dentro del framework de xUnit tenemos las siguientes

- Equal
- NotEqual
- Same
- NotSame
- Contains
- DoesNotContain
- InRange
- NotInRange
- IsAssignableFrom
- Empty
- NotEmpty
- True
- False
- IsType<T>
- Null
- NotNull
- Throws<T>

Hechos

Un hecho es una prueba única, se ejecutará una sola vez. Por ejemplo, si tenemos un método que multiplica dos números

```
public static class Math
{
	public static void Multiply(int a, int b)
	{
		return a * b;
	}
}
```

Ilustración 16:Fragmento de Código 14: xUnit.Net

Para probar la funcionalidad de este método podríamos tener varios hechos que prueben, por ejemplo, la multiplicación de 2 enteros, si uno de ellos es cero o si uno de ellos es negativo.

```
[Fact]
public void Multiply_two_int()
{
	//Arrange
	int a = 2;
	int b = 3;

	//Act
	var result = Math.Multiply(a,b);

	//Assert
	Assert.Equal(6,result)
}

[Fact]
public void Multiply_two_int_being_one_cero()
{
	//Arrange
	int a = 0;
	int b = 3;
```

```
        //Act
        var result = Math.Multiply(a,b);

        //Assert
        Assert.Equal(0,result)
}
```

Ilustración 17:Fragmento de Código 15: xUnit.Net Fact

Teorías

Las teorías son pruebas múltiples, como se puede observar en el ejemplo anterior, el código de ambas pruebas es prácticamente el mismo, cambiando únicamente los valores de entrada al método a probar. Para no tener que repetir código existe el atributo Theory que ejecutara la misma prueba tantas veces como valores le asignes, asignación que se hace mediante el atributo InlineData pasandole como argumentos los argumentos que le queremos pasar al método.

Los test del ejemplo anterior quedarían de la siguiente manera:

```
[Theory]
[InlineData(2,3,6)]
[InlineData(0,3,6)]
public void Multiply_two_int_being_one_cero(int a, int b, int
expectedResult)
{
        //Act
        var result = Math.Multiply(a,b);

        //Assert
        Assert.Equal(expectedResult,result)
}
```

Ilustración 18:Fragmento de Código 16: xUnit.Net Theory

FluentAssertions

En xUnit es posible crear nuestras propias aserciones, pero esto no tiene por qué ser necesario, en este caso presentamos FluentAssertions, una librería open source, además disponible como paquete nugget, con métodos de extensión para poder realizar las aserciones en las pruebas de una manera más clara utilizando sintaxis fluida.

Url del proyecto: https://fluentassertions.com/

Url del proyecto en GitHub: https://github.com/fluentassertions/fluentassertions

Url de documentación del proyecto: https://fluentassertions.com/introduction

La principal ventaja que nos ofrece es la capacidad de escribir nuestras aserciones indicando que es lo que se está afirmando y el resultado esperado con una sintaxis muy cercana al lenguaje hablado.

Ponemos como ejemplo el Fragmento de código 16 para realizar la aserción utilizando FluentAssertions:

```
[Theory]
[InlineData(2,3,6)]
```

```
[InlineData(0,3,6)]
public void Multiply_two_int_being_one_cero(int a, int b, int
expectedResult)
{
      //Act
      var result = Math.Multiply(a,b);

      //Assert
      result.Should().Be(expectedResult)
}
```

Ilustración 19:Fragmento de Código 17: FluentAssertions

Como se puede observar, la lectura, en inglés, de la aserción es rápidamente entendible.

Para utilizar estos recursos se debe llamar al método de extensión Should() seguido de cualquiera de los métodos que esta librería nos ofrece, como opciones más frecuentes tenemos:

- Be(expected)
- NotBe(expected)
- BeNull()
- NotBeNull()
- BeOfType<T>()
- BeOfType(type)
- BeSameAs(expected)
- NotBeSameAs(expected)
- Match(expresion)

Para tipos booleanos:

- BeTrue()
- BeFalse()
- NotBeTrue()
- NotBeFalse()

Para tipos numéricos:

- BeGreatherThan(expected)
- BeLessThan(expected)
- BeGreaterOrEqualTo(expected)
- BeLessOrEqualTo(expected)
- BeInRange(min,max)
- NotBeInRange(min,max)
- BePositive()
- BeNegative()

Para cadenas de texto:

- BeEmpty()
- NotBeEmpty()
- BeNullOrWhiteSpace()
- NotBeNullOrWhiteSpace()
- Contain(expected)

- NotContain(expected)
- StartWith(expected)
- NotStartWith(expected)
- EndWith(expected)
- NotEndWith(expected)

Por último, quería comentar el caso de las excepciones ya que se trata también de un caso frecuente de uso.

Si queremos validar si un determinado método lanza o no una excepción y validar sus propiedades existe el método Invoking al que se le puede pasar lo mismo un método que un Action.

Como ejemplo que nos da la propia documentación tenemos la validación de que el método Foo lanza un InvalidOperationException con un mensaje determinado.

```
subject.Invoking(y => y.Foo("Hello"))
    .Should().Throw<InvalidOperationException>()
    .WithMessage("Hello is not allowed at this moment");
```

Ilustración 20:Fragmento de Código 18: FluentAssertions Exceptions 1

```
Action act = () => subject.Foo2("Hello");

act.Should().Throw<InvalidOperationException>()
    .WithInnerException<ArgumentException>()
    .WithMessage("whatever");
```

Ilustración 21:Fragmento de Código 19:FluentAssertions Exceptions 2

Existen multitudes de opciones para estos tipos de datos y para varios otros como fechas, colecciones, diccionarios, enumeraciones, por suerte se trata de una librería muy bien documentada y si se desea ampliar la información descrita en este trabajo se puede consultar en la url indicada anteriormente, lo cual recomiendo encarecidamente.

Reglas de estilo de estándar

Los estándares de estilo son una serie de normas aceptadas por los desarrolladores para permitir a los lectores del código entender sin dificultad software ajeno a ellos disminuyendo el tiempo de aprendizaje necesario para comenzar a desarrollar en cualquier proyecto.

Existen muchas convenciones, pero para C# [7]recomendamos seguir el estilo de codificación establecido por la propia Microsoft. Aquí vamos a nombrar algunas de las principales, pero se puede obtener información completa en la url de la Especificación del lenguaje C#:https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/language-specification/readme

[7] C#: Lenguaje de programación orientado a objetos desarrollado por Microsoft.

Nombres de identificadores

"Un identificador es el nombre que se asigna a un tipo (clase, interfaz, estructura, registro, delegado o enumeración), miembro, variable o espacio de nombres." (Microsoft)

Microsoft en su documentación,

https://docs.microsoft.com/es-es/dotnet/csharp/fundamentals/coding-style/identifier-names establece una serie reglas y convenciones de nomenclatura

Reglas de nomenclatura

Para que un identificador sea válido debe seguir las siguientes reglas:

- Los identificadores deben comenzar con una letra o carácter de subrayado (_).
- Los identificadores pueden contener caracteres Unicode
- Los identificadores pueden contener palabras reservadas de C# usando como prefijo "@"

Para ampliar la información consultar el apartado sobre estructura léxica de Microsoft https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/language-specification/lexical-structure#643-identifiers

Convenciones de nomenclatura

Como objetivos de las convenciones de codificación Microsoft, en su apartado de convenciones de código C#: https://docs.microsoft.com/es-es/dotnet/csharp/fundamentals/coding-style/coding-conventions

Nos enumera cuatro principales:

- Crear una apariencia coherente de código para favorecer la atención en su contenido.
- Aumentar la velocidad de comprensión del código.
- Facilitar la copia, cambio y mantenimiento del código.
- Mostrar los procedimientos recomendados de C#.

En este mismo apartado expone las convenciones de nomenclatura, diseño, comentarios y lenguaje para el código C#.

Convenciones de nomenclatura

Como convenciones de nomenclatura destacamos dos:

- Uso de la grafía Pascal case ("PascalCasing") para asignar nombres a clases, records y estructuras. También en el caso de las interfaces, pero con la I delante.
- Uso de la grafía Camel Case ("camelCasing") para asignar nombres a campos private o internal y prefijos con (_) y para métodos.
- Prefijo s_ para campos private/internal static y t_ para subprocesos estáticos.

Para ampliar información: https://github.com/dotnet/runtime/blob/main/docs/coding-guidelines/coding-style.md

Convenciones de Diseño

Microsoft también define convenciones de diseño con el objetivo de un código más fácil de leer en el que destaque la estructura del código, como principales define las siguientes:

- Utilizar la configuración del Editor de código predeterminado.
- Escribir solo una instrucción por línea.
- Escribir solo una declaración por línea
- Si a las líneas de continuación no se les aplica la sangría automática, se debe hacer con una tabulación.
- Agregar al menos una línea en blanco entre definiciones de método y propiedad
- Utilizar paréntesis para que las cláusulas de expresión resulten evidentes.

Además, define convenciones para los comentarios, tipos de datos, delegados, try-catch, operadores && y ||, new, control de eventos, miembros estáticos, cláusulas de LINQ y seguridad que se pueden consultar aquí:
https://docs.microsoft.com/es-es/dotnet/csharp/fundamentals/coding-style/coding-conventions

Desarrollo

Es aquí en donde comenzamos la tarea de explicar cómo se deben aplicar las técnicas anteriormente desarrolladas. Para ello realizaremos un ejercicio práctico altamente estructurado y destinado a servir de ejemplo de aprendizaje. Los principios SOLID aquí serán claves.

Para que esta sección del TFG pueda servir de guía a los lectores plantearemos el desarrollo como un ejercicio en toda regla, de tal forma que habrá una parte correspondiente a la descripción de los requisitos para luego ir desarrollando la solución con aquellos aspectos comentarios propios de una evaluación cualificada según aportemos soluciones. Así vamos a simular la demanda de un cliente y a partir de ese momento iremos descubriendo las técnicas necesarias para su resolución optima.

Ejercicio formador:

Parte 1

Descripción

El cliente pide un pequeño programa que, a partir de una fecha determinada por el usuario (CurrentDate), dé, como salida, la siguiente fecha de ejecución y una breve descripción del proceso. El programa puede ser configurable de tal manera que se ejecute una sola vez (Once) o de manera recurrente (Recurring), además, se deberá configurar el periodo de fechas entre el cual se va a ejecutar (StartDate - EndDate). El sistema tendrá la posibilidad de ser habilitado o deshabilitado.

Inputs-Outputs

Once:

- Inputs:

    - Fecha actual (CurrentDate)
    - Tipo de ejecución (Type): Once
    - Próxima fecha de ejecución (DateTime)
    - Habilitado (Enabled)
    - Fecha de inicio (StartDate)
    - Fecha de fin (EndDate)
- Outputs:
    - Próxima fecha de ejecución (NextExecutionTime)
    - Descripción (Description)

Recurring

- Inputs:
    - Fecha actual (CurrentDate)
    - Tipo de ejecución (Type): Once
    - Días entre ejecuciones (Days)
    - Habilitado (Enabled)
    - Fecha de inicio (StartDate)
    - Fecha de fin (EndDate)
- Outputs:
    - Próxima fecha de ejecución (NextExecutionTime)
    - Descripción (Description)

Ejemplo de resolución

Once

- Inputs:

CurrentDate	04/01/2022 10:00
Type	Once
DateTime	08/01/2022 18:00
Enabled	True
StartDate	01/01/2022 00:00
EndDate	10/01/2022 00:00

- OutPuts:

NextExecutionTime	08/01/2022 18:00
Description	Ocurrs Once. Schedule will be used on 08/01/2022 at 18:00.

Recurring

- Inputs:

CurrentDate	08/01/2022 10:00
Type	Recurring
Days	1
Enabled	True

StartDate	01/01/2022 00:00
EndDate	

- Outputs:

NextExecutionTime	05/01/2022 10:00
	06/01/2022 10:00
	07/01/2022 10:00
	08/01/2022 10:00
Description	Ocurrs every day. Sechedule will be used on 05/01/2022 at 10:00 starting on 01/01/2022 at 00:00.

Objetivo

La complejidad de esta primera parte es baja, por lo que el principal objetivo es que el desarrollador diseñe su aplicación previendo que esta va a escalar y por ello deberá decantarse por un modelo adecuado de arquitectura, además de cumplir los principios SOLID y las reglas de estilo.

Otro objetivo importante de esta parte es que el desarrollador identifique que su aplicación no debe depender de la interfaz de usuario desde la cual se va a utilizar, por lo que la solución que adopte debe incluir la batería de pruebas de testing automatizado que cubran todos los posibles casos de uso, así como las validaciones de los parámetros de entrada de la aplicación.

En esta parte no serán excluyentes, pero hay que valorar aspectos relativos a la previsión de escalado que hace el desarrollador. Aquí me refiero a:

- Como realiza los inputs de fechas,
- Si tiene en cuenta que en un futuro pueda cambiar la cultura y por tanto el formato de fecha,
- Que los textos sean literales o prevea que la aplicación pueda necesitar traducción en un futuro
- Que el número de parámetros de entrada pueda aumentar considerablemente, etc.

Problemas Esperados

Una vez lanzado el ejercicio es muy probable que la respuesta de un desarrollador novel no sea la más adecuada. El error más importante que puede cometer el desarrollador consiste en que para realizar las pruebas necesarias durante el desarrollo de la aplicación, decida desarrollar una interfaz de usuario, lo más común puede ser encontrarnos un programa de consola o un formulario, que no tenga una arquitectura definida o que no se respeten los principios SOLID o las Reglas de Estilo requeridas..

Como errores de menor importancia podríamos detectar que el desarrollador no ha diseñado correctamente algunas de las partes de la aplicación, o no lo haya hecho con la adecuada previsión de escalado. También podremos encontrarnos, más específicamente, con que los DateTime son instanciados pasándoles la fecha como string, con textos literales.

Parte 2

Descripción

El cliente, habiendo funcionado un tiempo con el primer desarrollo, solicita algunos añadidos. Propone incluir un nuevo control para gestionar la frecuencia horaria sobre cada día. Esta opción no estará activa si la configuración diaria es Once.

Además de esto, le gustaría añadir un nuevo apartado para la gestión semanal, de tal manera que pueda seleccionar cada cuantas semanas se ejecuta la aplicación y que días de la semana.

Inputs-Outputs a añadir

Recurring:

- Número de Horas/Minutos/Segundos entre ejecuciones

Recurring Weekly:

- Selector múltiple de días, Lunes,Martes,Miércoles,Jueves,Viernes.
- Numero de semanas entre ejecuciones.

Ejemplo de resolución

Recurring

Inputs:

CurrentDate	08/01/2022 10:00
Type	Recurring
Days	1
Enabled	True
StartDate	01/01/2022 00:00
EndDate	
Hours Between Executions	12

Outputs:

NextExecutionTime	05/01/2022 10:00
	05/01/2022 22:00
	06/01/2022 10:00
	06/01/2022 22:00
Description	Ocurrs every day every 12 hours. Sechedule will be used on 05/01/2022 at 10:00 starting on 01/01/2022 at 00:00.

Recurring weekly

Inputs:

CurrentDate	02/01/2020 00:00
Type	Recurring
Days	1

Enabled	True
StartHour	04:00:00
EndHour	8:00.00
HourBetweenExecutions	2
WeeksBetweenExecutions	2
ExecutionDays	Monday, Thursday, Friday
StartDate	01/01/2020 00:00
EndDate	18/01/2020 00:00

Outputs:

NextExecutionTime	02/01/2022 04:00:00 (Thursday)
	02/01/2022 06:00:00
	02/01/2022 08:00:00
	03/01/2022 04:00:00 (Friday)
	03/01/2022 06:00:00
	03/01/2022 08:00:00
	13/01/2022 04:00:00 (Monday)
Description	Occurs every 2 weeks on Monday, Thursday, Friday every 2 hours between 04:00:00 and 08:00:00 starting on 01/01/2020 0:00:00

Objetivo

El objetivo de esta parte no es otro que hacer escalar el programa de tal manera que el desarrollador se dé cuenta por sí solo de si la solución que ha aportado a la primera parte está correctamente diseñada en términos de escalabilidad.

Por otro lado, la complejidad de la lógica se aumenta considerablemente, lo que hará al desarrollador darse cuenta de que le va a resultar mucho más sencillo el desarrollo si previamente analiza los casos de uso, desarrolla los test y después el código. La combinación de configuraciones tiene que derivar en un número elevado de test que cubran el máximo de casos de uso posibles.

Problemas Esperados

En esta parte es muy posible que el desarrollador tenga que rehacer su aplicación desde cero, lo cual no es malo, es parte del aprendizaje, tomó un camino equivocado y se ha dado cuenta pronto de que es mejor rehacerlo. Es posible también que veamos desarrolladores que a pesar de que código, al finalizar la primera parte, no nos pareciese del todo correcto, hayan buscado la manera de, dentro de un código poco escalable, encajar este nuevo funcionamiento, en manos del evaluador queda decidir si hacérselo ver o esperar que se tope más adelante con la realidad.

Por otro lado, habrá que prestar atención al desarrollo de los test ya su número ya que la complejidad de la lógica de esta parte ya se torna complicada y son la única garantía de que lo que se ha desarrollado funciona como se espera.

Parte 3

Descripción

El cliente propone llegar aún más allá y considera que para dar por concluida la funcionalidad necesita cubrir, por último, una configuración mensual que consistirá en poder elegir el primer, segundo, tercer o último lunes, martes, miércoles, jueves, viernes, día, día de la semana o día del fin de semana y el número de meses entre ejecuciones.

Inputs-Outputs a añadir

Recurring monthly

- Meses entre ejecuciones
- Ordinal de repetición (primer, segundo, tercer o último
- Días de ejecución (lunes, martes, miércoles, jueves, viernes, día, diade la semana o día del fin de semana.

Ejemplo de resolución

Recurring Monthly

Inputs:

CurrentDate	10/01/2022 03:00:00
Type	Recurring
Enabled	True
StartHour	03:00:00
EndHour	06:00.00
HourBetweenExecutions	1
MonthsBetweenExecutions	3
OrdinalDay	Third
ExecutionDays	WeekendDay
StartDate	01/01/2022 00:00
EndDate	18/01/2020 00:00

Outputs:

NextExecutionTime	22/01/2022 03:00:00
	22/01/2022 04:00:00
	22/01/2022 05:00:00
	22/01/2022 06:00:00
	09/04/2022 03:00:00
Description	Occurs the First WeekDay of every 3 months every 1 hours, 0 minutes, 0 seconds between 03:00:00 and 06:00:00 starting on 10/01/2022 0:00:00

Problemas Esperados

Como en el apartado anterior, en esta parte se complica aún más la lógica, aumentando considerablemente las posibles combinaciones, es por ello que las posibilidades de que el desarrollador que no ha hecho las cosas bien en las partes anteriores ya vea imposible continuar sin repensar y rehacer su solución son ya muy altas.

De nuevo hay que prestar mucha atención a los test y su número ya debe ser muy elevado, son garantía de funcionamiento correcto.

Parte 4

Descripción

A estas alturas, el cliente está tan satisfecho con la aplicación que pretende implementarla en sus oficinas internacionales, es por ello que solicita que, además de en español, la aplicación pueda funcionar en Ingles de UK y de US.

Hay que tener en cuenta de que además de los textos, el formato US altera el orden de la fecha a mm-dd-yyyy y qué quizá posteriormente pueden pedir nuevos idiomas.

Inputs-Outputs a añadir

- Cultura (en-En, en-US, es-Es)

Objetivo

EL objetivo de esta parte es que el desarrollador se dé cuenta de que una aplicación no solo puede escalar en funcionalidad si no que puede hacerlo en diferentes direcciones.

Se pretende que sea capaz de diseñar un sistema de traducción de textos escalable, no hace falta, aunque si quiere puede hacerlo, que utilice una base de datos, puede valernos con que la simule con diccionarios, pero en el desarrollo final no debería haber literales.

Tampoco debe haber literales a la hora de gestionar las fechas, se pretende que, si ha utilizado estos, los sustituya por instancias que no los usen, el objeto DateTime puede ser instanciado sin literales y gestiona los cambios de cultura perfectamente.

Problemas Esperados

Es posible que haya un amplio número de desarrolladores que no haya tenido en cuenta los cambios de cultura y tenga su desarrollo lleno de literales, deberán sustituirlos y dejarla limpia de los mismos y, a su vez, adaptar los test antiguos y generar nuevos que cubran los textos en diferentes culturas y el cambio de orden en las fechas. La única complicación que podemos encontrar en esta última parte es que el desarrollador no sea capaz de idear un sistema de traducciones, en tal caso podremos darle alguna pista, el objetivo no es que sea un sistema perfecto y complejo si no que se dé cuenta de las diferentes direcciones de escalabilidad.

Posible extensión del trabajo:

El trabajo asume una estructura propia de trabajo fin de grado como es obligado; sin embargo, propone directrices solidas encaminadas a la formación que pudieran ser extendidas y ampliadas con el propósito de articular un curso de formación para programadores. La

recolección inicial de aquellas técnicas y métodos irrenunciables en el ámbito de la programación moderna podría ser ampliada estableciendo un guion de aprendizaje con ejemplos prácticos que deberían desembocar en la resolución del ejercicio final evaluable en base a los principios aprendidos.

A su vez la estructura en partes del ejercicio permite escalarlo y añadir partes nuevas en las que pueda interesar su aprendizaje, como por ejemplo persistencia de datos, control de código de fuente, integración continua y despliegue automatizado (CI/CD), etc.

Tabla de Ilustraciones:

Ilustración 1: Estructura de capas 8
Ilustración 2: The Clean Architecture 10
Ilustración 3: Fragmento de Código 1: Ejemplo Single Responsability 1 12
Ilustración 4:Fragmento de Código 2: Ejemplo Single Responsability 2 12
Ilustración 5: Fragmento de Código 3: Ejemplo Open/closed 1 13
Ilustración 6:Fragmento de Código 4: Ejemplo Open/Closed 2 13
Ilustración 7: Fragmento de Código 5:Ejemplo Open/Closed 3 13
Ilustración 8:Fragmento de Código 6: Ejemplo Liskov 1 14
Ilustración 9:Fragmento de Código 7: Ejemplo Interface segregation 1 15
Ilustración 10:Fragmento de Código 8: Ejemplo Interface segregation 2 15
Ilustración 11:Fragmento de Código 9: Ejemplo Dependency inversion 1 16
Ilustración 12:Fragmento de Código 10: Ejemplo Dependency inversion 1 16
Ilustración 13:Fragmento de Código 11: Código Legacy 1 18
Ilustración 14:Fragmento de Código 12: Código Legacy 2 18
Ilustración 15:Fragmento de Código 13: Código Legacy 3 19
Ilustración 16:Fragmento de Código 14: xUnit.Net 23
Ilustración 17:Fragmento de Código 15: xUnit.Net Fact 24
Ilustración 18:Fragmento de Código 16: xUnit.Net Theory 24
Ilustración 19:Fragmento de Código 17: FluentAssertions 25
Ilustración 20:Fragmento de Código 18: FluentAssertions Exceptions 1 26
Ilustración 21:Fragmento de Código 19:FluentAssertions Exceptions 2 26

Referencias:

C.Martin, R. (13 de August de 2012). *The Clean Code Blog*. Obtenido de https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Evans, E. (2003). *Doman-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional.

Feathers, M. (2004). *Working Effectively with Legacy Code.* Addison Wesley.

Fowler, M. (s.f.). *Refactoring Improving the Design of Existing Code.* Addison Wesley.

Microsoft. (s.f.). *Docs.Microsoft.com.* Obtenido de Docs.Microsoft.com: https://docs.microsoft.com/es-es/dotnet/csharp/fundamentals/coding-style/identifier-names

Wikipedia. (s.f.). *Wikipedia. La enciclopedia libre.* Obtenido de https://es.wikipedia.org/wiki/Principio_de_segregaci%C3%B3n_de_la_interfaz