Minimización de escritura y Maximización de reutilización de código fuente mediante el uso de programación en capas y buenas prácticas de programación

RESUMEN: En la construcción de software se busca que funcione y se construya de forma escalable, reutilizable y mantenible. El desarrollar el software en capas permite la reutilización de código para varias aplicaciones, hacer uso de patrones de diseño mejora considerablemente la correcta implementación del código, así mismo el usar técnicas como SOLID y CLEAN CODE permiten una mejor construcción, entendimiento y documentación por y para los programadores. Haciendo uso de Visual C# e implementando lo antes mencionado mediante el uso de Interfaces, Clases Genéricas, Herencia, Polimorfismo y bibliotecas Standard permitirá que el desarrollo de aplicaciones de escritorio (WinForms y WPF), Web (Asp.Net) y Móviles (iOS y Android) se realicen en tiempos muy cortos así como reducir el número de líneas de código, el mantenimiento y depuración del mismo consiguiendo maximizar las funcionabilidades, la estandarización, validación y destino de las aplicaciones. De esta forma resulta fácil crear aplicaciones de escritorio y migrarlas fácilmente a web o aplicaciones móviles nativas de apoyo. Aún más, al hacer uso de Frameworks permitirá crear aplicaciones de escritorio multiplataforma y al aplicarlo en las herramientas de desarrollo para el Internet de las Cosas, tales como Raspberry Pi entre otras.

PALABRAS CLAVE: Clean Code, Patrones de Diseño, SOLID

Colaboración
Espinoza Galicia Carlos Arturo; Martínez Endonio Alberto; Escalante Cantú Mario, Instituto Tecnológico Superior De Huichapan

ABSTRACT: In the construction of software it is sought that it works and is built in a scalable, reusable and maintainable way. Developing the layered software allows code reuse for several applications, making use of design patterns considerably improves the correct implementation of the code, as well as using the techniques like SOLID and CLEAN CODE allow a better Construction, understanding and documentation by and for programmers. Using Visual C# and implementing the aforementioned through the use of Interfaces, generic classes, inheritance, polymorphism and Standard libraries will allow the development of desktop applications (WinForms and WPF), Web (Asp.Net) and mobile (IOS and Android) are done in very short times as well as reduce the number of lines of code, the maintenance and debugging of the same by maximizing the Funcionabilidades, standardization, validation and destination of the applications. This makes it easy to create desktop applications and easily migrate them to Web or native mobile support applications. Moreover, by making use of frameworks will allow to create multiplatform desktop applications and to apply it in the development tools for the Internet of things, such as Raspberry Pi among others.

KEYWORDS: Clean Code, Design Patterns, SOLID

INTRODUCCIÓN

En el mercado del software, existe gran cantidad de software similar, la diferencia muchas veces radica en como está construido, la forma como se escribe tiene mucho que ver con la experiencia de los desarrolladores y los líderes del proyecto. Se entiende que un software debe hacer las cosas para lo que fue

diseñado, eso no debe quedar duda, pero también debe cumplir con estándares de calidad del lado de los desarrolladores, es por tal que el uso de un estándar de desarrollo que permita reutilizar código, mantenerlo de una manera más fácil, reducir tiempos de desarrollo y mantenimiento y sobre todo que permita fácilmente integrar cambios y nuevas acciones es la diferencias entre un software de "escuela" y un software de "grandes ligas".

En el presente documento se muestra una metodología para el desarrollo de proyectos de software basado en tecnologías .Net con el lenguaje Visual C#. Esta metodología emplea el uso de clases abstractas, interfaces, clases genéricas, patrones de diseño, buenas prácticas y desarrollo en capas [1].

Se muestra un ejemplo de cómo desarrollar cada capa, poniendo énfasis en las capas donde se puede reutilizar mayor parte de código como las capas COMMON y DAL [2]

MATERIAL Y MÉTODOS

A continuación, se describe la metodología que se emplea para el desarrollo de proyectos basados en Tecnologías .Net de Microsoft, se enlista cada una de las etapas a seguir (cabe tomar en cuenta, que antes y después pueden existir más etapas, sin embargo, aquí solo se realiza el enfoque a la parte de codificación del software):

1.Tipo título Diseño de arquitectura de software. Como primer parte, se tiene que observar cual será la arquitectura por usar, es decir que tipo de servidor de base de datos usar: Relacional, NoSQL [3] u otra y de que forma se van a trasportar los datos mediante las capas; algo a tomar en cuenta es si otras tecnologías intervendrán en el proyecto, para en su caso diseñar una capa de transporte, pero básicamente se propone cualquiera de los modelos mostrados en las figuras 1 y 2.

La figura 1, muestra una arquitectura en capas [4] que tiene la capa de servicio en la parte superior y está destinada a solo exponer lo mínimo necesario (posiblemente solo lo implementado por hardware); esta es especialmente útil cuando la tecnología de base de datos es totalmente soportada por todas las capas de interfaz de usuario realizadas en .Net, tal es el caso de MongoDB [3], ya que su driver implementado en bibliotecas .Net Standard pueden ser consumidas por cualquier aplicación desarrollada en .Net incluyendo aplicaciones Android y iOS desarrolladas en Xamarin [1].

La figura 2, muestra una arquitectura similar a la anterior pero que la capa de transporte se encuentra entre la capa DAL y la capa BIZ, de modo que esta implementa una API RestFull [5] que soportara a otras tecnologías compatibles con una API RestFull, tecnologías como Angular, JavaScript, PHP, Java y otras, la ventaja de esta implementación es la creación de microservicios que pueden ser usadas por otras tecnologías.

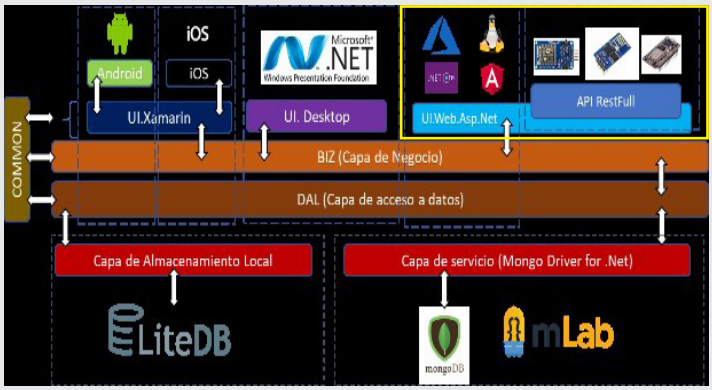

Figura 1: Propuesta de arquitectura en capas donde solo la capa de servicios solo se usa para comunicar métodos explícitos (Construcción Propia)

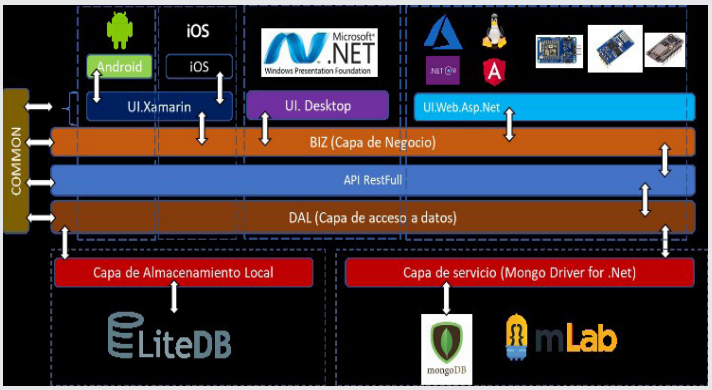

Figura 2: Propuesta de arquitectura en capas donde la capa de Transporte o Servicios implementa microservicios (Construcción propia)

En si ambas formas pueden ser soporte de otras tecnologías, la diferencia entre ellas es que la primera está orientada a proporcionar solo los servicios necesarios para otras tecnologías (principalmente que son accedidas desde hardware como Arduino, NodeMCU [6] o Raspperry Pi [7]), reduciendo notablemente la implementación y código de esta mientras que la segunda se tienen microservicios para todas las entidades y repositorios, el modo de decisión

2.Tipo título Diseño de entidades y/o base de datos. Aquí se propone diseñar las entidades con una base de datos NoSQL, pensando siempre en una gran cantidad de datos, además que estas son más rápidas que las relacionales, en la figura 3 se muestra un ejemplo de un diagrama de clases usando el concepto de DTO [8].

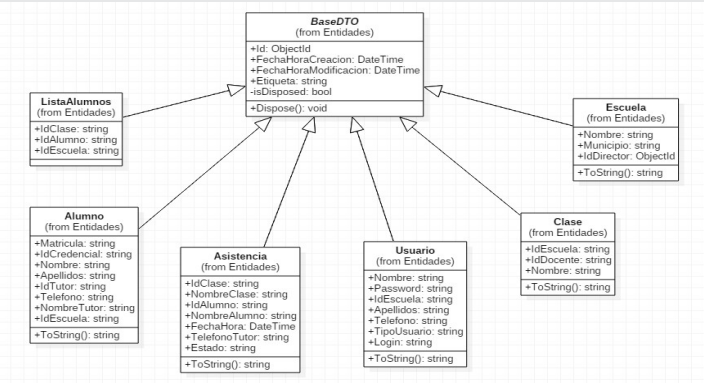

Figura 3: Ejemplo de Diagrama de clases que implementa clases abstractas y DTO (Construcción propia)

3. Tipo título Desarrollo de Capas Base

a. Diseño de capa COMMON, aquí se propone crear las siguientes carpetas (figura 4):

(a) Entidades: que contiene las clases que definen las entidades (tablas) de la base de datos, aquí se puede hacer mediante un ORM como entity framework, otro o incluso de forma manual, aquí se propone el uso de DTO y una clase base abstracta que pueda ser heredada por otras entidades y que por consiguiente ya tengan todas las propiedades de la clase base, ejemplo de esto se muestra en la figura 3. Con esto se observa que ya empezamos a reutilizar código, en fases posteriores se observa la gran ventaja de usar estas clases bases.

(b) Interfaces, en esta se definen todas las interfaces a implementar por los repositorios en la capa DAL y BIZ [2], aquí se pueden implementar clases genéricas. En la figura 5, se muestra el código de la interfaz para los repositorios genéricos y en la figura 6 se muestra el código de una interfaz que implementaría un método Manager [9] en la capa BIZ.

(c) Modelos, en esta carpeta se crearán todos los modelos que serán ocupados en las distintas interfaces de usuario, especialmente los usados por los patrones MVVM [10] y MVC [5], ya que en la mayoría de las ocasiones estos modelos pueden compartirse, en la figura 7 se muestra un ejemplo de un modelo Login que es común tanto para la aplicación móvil como la web y la de escritorio.

(d) Tools, esta carpeta es para crear métodos de apoyo para las capas superiores como por ejemplo métodos de cifrado u otros.

(e) Validadores, esta carpeta almacena el código de las clases que validan las entidades antes de ser insertadas en la base de datos [11], haciendo uso de genéricos y abstractos también se pueden reutilizar código, la figura 8 y 9 muestran los códigos de un validador genérico y uno especifico donde se observa reutilización de código.

Figura 4: Propuesta de carpetas de la capa COMMON (Construcción Propia)

```
– referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 2 cambios
public interface IGenericRepository<T> where T:BaseDTO
{
    – referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 1 cambio | 0 excepciones
    bool Resultado { get; set; }
    – referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 1 cambio | 0 excepciones
    string Error { get; set; }
    – referencias | MasterCarlosEspinoza, Hace 108 días | 1 autor, 1 cambio | 0 excepciones
    T Create(T entidad);
    – referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 2 cambios | 0 excepciones
    IQueryable<T> Read { get; }
    – referencias | MasterCarlosEspinoza, Hace 108 días | 1 autor, 1 cambio | 0 excepciones
    T Update(T entidad);
    – referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 1 cambio | 0 excepciones
    bool Delete(ObjectId id);
    – referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 1 cambio | 0 excepciones
    T SearchById(ObjectId id);
    – referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 2 cambios | 0 excepciones
    IQueryable<T> Query(Expression<Func<T, bool>> predicado);
    – referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 1 cambio | 0 excepciones
    bool DeleteAll(Expression<Func<T, bool>> predicado);
    – referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 1 cambio | 0 excepciones
    IQueryable<T> CreateAll(IQueryable<T> entidades);
}
```

Figura 5: Código de la interface genérica propuesta para los repositorios.

```
21 referencias | Carlos Espinoza, Hace 19 días | 2 autores, 2 cambios
public interface IGenericManager<T> where T:BaseDTO
{
    5 referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 1 cambio | 0 excepciones
    bool Resultado { get; set; }
    5 referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 1 cambio | 0 excepciones
    string Error { get; set; }
    2 referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 1 cambio | 0 excepciones
    T Insertar(T entidad);
    4 referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 1 cambio | 0 excepciones
    IQueryable<T> Listar { get; }
    2 referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 1 cambio | 0 excepciones
    T Actualizar(T entidad);
    2 referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 1 cambio | 0 excepciones
    bool Eliminar(ObjectId id);
    2 referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 1 cambio | 0 excepciones
    T BuscarPorId(ObjectId id);
    1 referencia | MasterCarlosEspinoza, Hace 46 días | 1 autor, 1 cambio | 0 excepciones
    bool EliminarTodos(Expression<Func<T, bool>> predicado);
    2 referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 1 cambio | 0 excepciones
    IQueryable<T> InsertarTodos(IQueryable<T> entidades);
    1 referencia | Carlos Espinoza, Hace 19 días | 1 autor, 1 cambio | 0 excepciones
    IQueryable<T> ListarPaginadoPersonalizado(int omitir=-1, int mostrar=-1);
    1 referencia | Carlos Espinoza, Hace 19 días | 1 autor, 1 cambio | 0 excepciones
    IQueryable<T> ListarPaginado(int tamanioPagina, int numeroPagina);
}
```

Figura 6: Código de la interfaz genérica para los Managers (Creación Propia)

```
0 referencias | Carlos Espinoza, Hace 35 días | 1 autor, 1 cambio
public class ModelInicioSesion
{
    0 referencias | Carlos Espinoza, Hace 35 días | 1 autor, 1 cambio | 0 excepciones
    public string Email { get; set; }
    0 referencias | Carlos Espinoza, Hace 35 días | 1 autor, 1 cambio | 0 excepciones
    public string Constrasenia { get; set; }
}
```

Figura 7: Código del Modelo Login, que puede ser usado en los patrones MVVM y MVC en las distintas interfaces graficas de usuario.

```
18 referencias | MasterCarlosEspinoza, Hace 108 días | 1 autor, 1 cambio
public abstract class GenericValidator<T>:AbstractValidator<T> where T:BaseDTO
{
    0 referencias | MasterCarlosEspinoza, Hace 108 días | 1 autor, 1 cambio | 0 excepciones
    public GenericValidator()
    {
        RuleFor(p => p.Id).NotNull().NotEmpty();
        RuleFor(p => p.FechaHoraCreacion).NotNull();
        RuleFor(p => p.FechaHoraModificacion).NotNull();
    }
}
```

Figura 8: Código de un validador genérico para entidades que heredas de BaseDTO (Construcción Propia

```
2 referencias | MasterCarlosEspinoza, Hace 108 días | 1 autor, 1 cambio
public class AvisoAlertaValidator:AbstractValidator<AvisoAlerta>
{
    1 referencia | MasterCarlosEspinoza, Hace 108 días | 1 autor, 1 cambio | 0 excepciones
    public AvisoAlertaValidator()
    {
        RuleFor(p => p.Destinatario).NotNull().NotEmpty();
        RuleFor(p => p.DireccionAviso).NotNull().NotEmpty();
        RuleFor(p => p.IdAlerta).NotNull().NotEmpty();
        RuleFor(p=>p.IdFormadeAviso).NotNull().NotEmpty();
        RuleFor(p=>p.IdLectura).NotNull().NotEmpty();
        RuleFor(p=>p.IdRegla).NotNull().NotEmpty();
        RuleFor(p=>p.IdSensor).NotNull().NotEmpty();
        RuleFor(p=>p.Texto).NotNull().NotEmpty();
    }
}
```

Figura 9: Código de un validador de la entidad AvisoAlerta, la cual hereda del validador genérico mostrado en la figura 8 (Construcción Propia)

b. Diseño de capa DAL. Esta capa es la que accede directamente a la base de datos y que implementa las operaciones CRUD [12] para cada entidad/tabla, aquí la idea es generar una sola clase que implemente la interfaz genérica de la capa common correspondiente, en la figura 10 se observa como implementa la interfaz mencionada.

```
19 referencias | Carlos Espinoza, Hace 19 días | 2 autores, 3 cambios
public class GenericRepository<T> : IGenericRepository<T> where T : BaseDTO
{
    private MongoClient client;
    private IMongoDatabase db;
    18 referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 1 cambio | 0 excepciones
    public GenericRepository(AbstractValidator<T> validator)
    {
        Validator = validator;
        client = new MongoClient(new MongoUrl(@"mongodb://"));
        db = client.GetDatabase("");
        Resultado = false;
        Error = "";
    }
    8 referencias | MasterCarlosEspinoza, Hace 46 días | 1 autor, 1 cambio | 0 excepciones
    private IMongoCollection<T> Collection()
    {
        return db.GetCollection<T>(typeof(T).Name);
    }
```

Figura 10: Fragmento de clase genérica que implementa la interfaz para los métodos CRUD para cada entidad (Construcción Propia)

c. Diseño de capa BIZ, esta capa es la encargada de implementar los métodos de negocio del software, aquí se genera una clase abstracta que ya implemente los métodos CRUD (Figura 11) y una clase por cada entidad, estas tienen que implementar su respectiva interfaz definida en la capa COMMON (figura 12), en esta se puede observar que prácticamente ya no contiene código.

```
19 referencias | Carlos Espinoza, Hace 19 días | 1 autor, 3 cambios
public class GenericManager<T> : IGenericManager<T> where T : BaseDTO
{
    protected internal IGenericRepository<T> Repository;
    18 referencias | Carlos Espinoza, Hace 39 días | 1 autor, 1 cambio | 0 excepciones
    public GenericManager(IGenericRepository<T> repository)
    {
        Repository = repository;
    }

    4 referencias | Carlos Espinoza, Hace 39 días | 1 autor, 1 cambio | 0 excepciones
    public IQueryable<T> Listar => Repository.Read;

    5 referencias | Carlos Espinoza, Hace 39 días | 1 autor, 1 cambio | 0 excepciones
    public bool Resultado
    {
        get { return Repository.Resultado; }
        set { Repository.Resultado = value; }
    }
    5 referencias | Carlos Espinoza, Hace 39 días | 1 autor, 1 cambio | 0 excepciones
    public string Error
    {
        get { return Repository.Error; }
        set { Repository.Error = value; }
    }
```

Figura 11: Fragmento de la clase Genérica que implementa la interfaz para consumir un repositorio genérico (Construcción Propia)

```
3 referencias | Carlos Espinoza, Hace 14 días | 1 autor, 1 cambio
public class CarolisRobotManager : GenericManager<CarolisRobot>, ICarolisRobotManager
{
    1 referencia | Carlos Espinoza, Hace 14 días | 1 autor, 1 cambio | 0 excepciones
    public CarolisRobotManager(IGenericRepository<CarolisRobot> repository) : base(repository)
    {
    }
}
```

Figura 12: Clase que implementa la interfaz de la capa COMMON y que hereda de la clase genérica (Construcción Propia)

4.Tipo título Diseño de capa de transporte: para el caso de Asp.Net, el crear una API es mediante la creación de controladores, estos controladores deben tener métodos para cada verbo HTTP [5], la figura 13 muestra una clase genérica que implementa estos métodos y la figura 14 muestra un controlador (prácticamente sin código) que genera el ENDPOINT necesario para cada entidad.

```
[Produces("application/json")]
[Route("api/GenericApi")]
19 referencias | Carlos Espinoza, Hace 19 días | 1 autor, 2 cambios
public abstract class GenericApiController<T> : Controller where T:BaseDTO
{
    IGenericManager<T> manager;
    18 referencias | Carlos Espinoza, Hace 35 días | 1 autor, 1 cambio | 0 excepciones
    public GenericApiController(IGenericManager<T> genericManager)
    {
        manager = genericManager;
    }
    // GET: api/GenericApi
    [HttpGet]
    0 referencias | Carlos Espinoza, Hace 35 días | 1 autor, 1 cambio | 0 solicitudes | 0 excepciones
    public IEnumerable<T> Get()
    {
        return manager.Listar;
    }

    // GET: api/GenericApi/5
    [HttpGet("{id}")]
    0 referencias | Carlos Espinoza, Hace 35 días | 1 autor, 1 cambio | 0 solicitudes | 0 excepciones
    public IActionResult Get(ObjectId id)
    {
        if (string.IsNullOrEmpty(id.ToString()))
        {
            return BadRequest("El Id es requerido");
        }
```

Figura 13: Fragmento de clase genérica para implementar los verbos HTTP de una API (Construcción Propia)

```
3 referencias | Carlos Espinoza, Hace 14 días | 1 autor, 1 cambio
public class CarolisRobotManager : GenericManager<CarolisRobot>, ICarolisRobotManager
{
    1 referencia | Carlos Espinoza, Hace 14 días | 1 autor, 1 cambio | 0 excepciones
    public CarolisRobotManager(IGenericRepository<CarolisRobot> repository) : base(repository)
    {
    }
}
```

Figura 14: Clase de un controlador para implementar una API, hereda de la clase mostrada en la figura 14 y se puede observar que prácticamente no tiene código (Construcción Propia)

5. Diseño de Capas de Interfaz de Usuario: estas capas, depende de cada patrón utilizado, pero por ejemplo para una aplicación de escritorio WPF, se pueden usar controles que permitan generar un editor de catálogos, para esto, se puede usar la interfaz propuesta en la figura 15, en la figura 16 se muestra un editor de catálogos que implementa los métodos de la interfaz de la figura 15. Solo resta crear una clase como la mostrada en la figura 17 que implemente los métodos de la interfaz de la figura 15 y se exponga a la interfaz de usuario de la figura 16. De esta forma solo se crea una clase por entidad solo con los controles necesarios para poder editar la entidad y solo se indica los objetos a trabajar, ahorrando así tiempo de desarrollo, codificación y reutilizando mucho código.

```
45 referencias | MasterCarlosEspinoza, Hace 112 días | 2 autores, 3 cambios
public interface IRepositorioPanel
{
    /// <summary> Obtiene el nombre de la Entidad de la cual se esta creando el cata ...
    19 referencias | Carlos Espinoza, Hace 132 días | 1 autor, 1 cambio
    string NombreDeEntidad { get; }
    /// <summary> Obtiene el error ocurrido en las operación de Guardar, Modificar o ...
    20 referencias | Carlos Espinoza, Hace 132 días | 1 autor, 1 cambio
    string Error { get; }
    /// <summary> Guarda el registro
    19 referencias | Carlos Espinoza, Hace 132 días | 1 autor, 1 cambio
    bool Guardar();
    /// <summary> Modifica el registro
    19 referencias | Carlos Espinoza, Hace 132 días | 1 autor, 1 cambio
    bool Modificar();
    /// <summary> Elimina el registro
    19 referencias | Carlos Espinoza, Hace 132 días | 1 autor, 1 cambio
    bool Eliminar();
    /// <summary> Cancela el modo de modificación, limpia los controles e inhabilita ...
    20 referencias | Carlos Espinoza, Hace 132 días | 1 autor, 1 cambio
    void CancelarModificacion();
    /// <summary> Asigna el datagrid del control al panel para que pueda interactuar
    19 referencias | Carlos Espinoza, Hace 132 días | 1 autor, 1 cambio
    DataGrid AsignarDataGrid { set; }
    19 referencias | MasterCarlosEspinoza, Hace 112 días | 1 autor, 1 cambio
    ComboBox AsignarComboFiltro { set; }
    /// <summary> Carga los datos iniciales cuando se muestra el panel, por ejemplo ...
    19 referencias | Carlos Espinoza, Hace 132 días | 1 autor, 1 cambio
    void CargarDatosIniciales();
    /// <summary> Manda a imprimir el reporte del control
```

Figura 15: Fragmento de la interfaz IRepositorioPanel la cual permite implementar métodos genéricos para un editor de catálogos. (Construcción Propia)

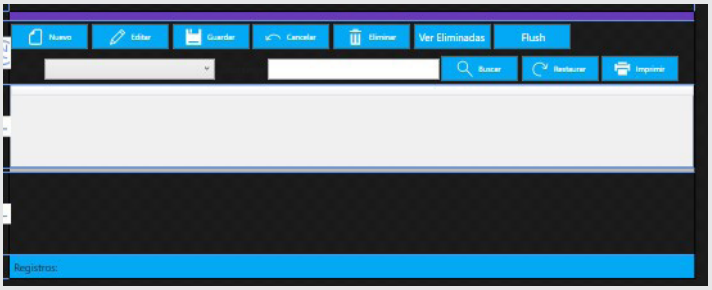

Figura 16: Diseño de un editor de catálogos genérico para implementar con la interfaz de la figura 15 (Construcción propia)

```
19 referencias | Carlos Espinoza, Hace 19 días | 1 autor, 3 cambios
public class GenericManager<T> : IGenericManager<T> where T : BaseDTO
{
    protected internal IGenericRepository<T> Repository;
    18 referencias | Carlos Espinoza, Hace 39 días | 1 autor, 1 cambio | 0 excepciones
    public GenericManager(IGenericRepository<T> repository)
    {
        Repository = repository;
    }

    4 referencias | Carlos Espinoza, Hace 39 días | 1 autor, 1 cambio | 0 excepciones
    public IQueryable<T> Listar => Repository.Read;

    5 referencias | Carlos Espinoza, Hace 39 días | 1 autor, 1 cambio | 0 excepciones
    public bool Resultado
    {
        get { return Repository.Resultado; }
        set { Repository.Resultado = value; }
    }
    5 referencias | Carlos Espinoza, Hace 39 días | 1 autor, 1 cambio | 0 excepciones
    public string Error
    {
        get { return Repository.Error; }
        set { Repository.Error = value; }
    }
```

Figura 17: Fragmento de una clase que hereda de un UserControl e implementa la interfaz de la figura 15 (Construcción Propia)

Con las otras capas de interfaz de usuario, se puede hacer algo similar, donde se reutilice el código todo mediante el uso de clases genéricas, abstractas e interfaces.

RESULTADOS

Después de realizar la arquitectura y las interfaces iniciales, siguiendo esta metodología, los proyectos son muy fáciles de codificar con un mínimo de líneas.

Esta misma metodología se ha usado para realizar proyectos que sus resultados ya han sido publicados en revistas indexadas, como:

- Implementación de plataforma web y aplicaciones móviles mediante buenas prácticas usando tecnología .NET [2]
- Evaluador de calidad de escritura de código fuente [13]
- Implementación de plataforma para el internet de las cosas en un ambiente de nube pública. [14]

Estos proyectos implementan también metodologías KISS [15], CleanCode [16] y SOLID [17] para el desarrollo.

Resultados como tal se pueden observar directamente en la sección de Metodología, donde se observa en las figuras la reutilización de código, minimización de escritura de código y maximización de funcionabilidad, el usar estas técnicas requiere entender muy bien cada concepto, pero el usarlas también ayuda mucho a la comprensión a profundidad de estos.

Aplicando esta metodología, proyectos completos (en su fase preliminar) se han terminado en tiempo récord; estos mismos principios se han presentado a alumnos de 1ro y 3er semestre de la Ingeniería en Sistemas Computacionales y has servido para comprender de forma más que adecuada los conceptos

de Programación a Objetos, entregando proyectos altamente escalables, reutilizables y mantenibles.

Si bien al principio esta metodología pudiera resultar enredada y tediosa con el uso y el paso del tiempo a resultado de gran beneficio, ya que por ejemplo, los validadores, como se pudo observar en la figura 10, estos son implementados directamente en la capa DAL, por lo que no es necesario validar en cada Interfaz de usuario, de modo que los mismos errores de validación (los textos) se dan en todas las interfaces de usuarios ganando estandarización en la implementación, dejando a los programadores preocuparse por lo importante como implementar los métodos en la capa BIZ.

Otra gran ventaja del trabajo en capas es que una vez que se tiene una interfaz de usuario funcionando, migrar a otro contexto, por ejemplo de escritorio a web o de web a móviles, simplemente se generan la nueva interfaz de usuario ya que los métodos generados ya funcionan y que además de que con alguna modificación las modificaciones hechas en la capa BIZ estarán disponibles para implementar fácilmente en todas las interfaces de usuario ganando velocidad de desarrollo y disminución de tiempo de depuración.

Algo importante (que no se tocó en este documento) es el uso de pruebas unitarias y automatizadas, con el uso de este tipo de codificación las pruebas unitarias también se simplifican, por ejemplo, en los repositorios, una sola clase funciona para cualquier número de entidades, por lo que depurando una única clase es funcional para todas nuestras entidades.

CONCLUSIONES

Si bien el construir un software es un arte, también se deben tener técnicas que hagan más productivos a los equipos de trabajo, como se pudo observar usando interfaces, clases genéricas, abstractas, implementando patrones de diseño y buenas prácticas de programación se gana en:

- Escalabilidad
- Reutilización de código
- Disminución de tiempo de desarrollo
- Disminución de tiempo de depuración
- Estandarización de escritura de código
- Disminución de tiempo de capacitación de nuevos recursos humanos.

Si bien al principio esta metodología pudiera resultar enredada y tediosa con el uso y el paso del tiempo a resultado de gran beneficio.

BIBLIOGRAFÍA

[1] D. Esposito y A. Saltarello, Microsoft .Net: Architecting Applications for the Enterprise, Whashington: Microsoft Press, 2015.

[2] C. Espinoza Galicia, A. Martínez Endonio, M. Escalante Cantu y R. Martinez Rangel, «Implementación de Plataforma Weby Aplicaciones Móviles Mediante Buenas Practicas Usando Tecnología .Net,» Revista de Tecnologias de la información y Comunicaciones, pp. 42-49, 2017.

[3] D. Cyrus y N. Amol, MongoDB Coockbook, Birmingham, UK.: Packt Publishing, 2016.

[4] Microsoft Corporation, Microsoft Application Architecture Guide, Patterns & Practices, 2nd Edition, Redmond: Microsoft Corporation, 2009.

[5] Microsoft Corporation, «Información general sobre ASP.NET MVC,» [En línea]. Available: https://msdn.microsoft.com/es-es/library/dd381412(v=vs.108).aspx.

[6] NodeMCU Team, «NodeMcu: Connect Things EASY,» 2014. [En línea]. Available: http://www.nodemcu.com/index_en.html.

[7] Programming for the Intener of Things, Using Windows 10 IoT Core and Azure IoT Suite, Redmond, Washington, US: Microsoft Press, 2017.

[8] Microsoft Corporation, «Data Transfer Object,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/ff649585.aspx.

[9] L. González Bañales, «SG BUZZ Conocimiento para crear Software Grandioso,» 8 Julio 2016. [En línea]. Available: https://sg.com.mx/content/view/213. [Último acceso: 2 Agosto 2016].

[10] Microsoft Corporation, «El patrón MVVM,» [En línea]. Available: https://msdn.microsoft.com/es-mx/library/hh848246.aspx.

[11] J. Bishop, C# 3.0 Design Patterns, Sebastopol, CA: O'Reilly Media, Inc., 2008.

[12] 1&1 Internet Inc, «CRUD: la base de la gestión de datos,» [En línea]. Available: https://www.1and1.mx/digitalguide/paginas-web/desarrollo-web/crud-las-principales-operaciones-de-bases-de-datos/.

[13] C. Espinoza Galicia, W. Gómez López y R. Reyes López, «Evaluador de calida de escritura de código fuente,» Revista de Tecnológia Informática, pp. 30-35, 2017.

[14] C. Espinoza Galicia, W. Gómez López y R. F. Guillén Mallete, «Implementación de plataforma para el Internet de las cosas en un ambiente de nube pública,» Tecnologia Educatíva, pp. 63-69, 2016.

[15] S. Rossel, «KISS – One Best Practice to Rule Them All,» 19 Agosto 2015. [En línea]. Available: https://simpleprogrammer.com/2015/08/19/kiss-one-best-practice-to-rule-them-all/.

[16] M. Robert C., Clean Code, A Handbook of Agile Software Craftsmanship, Boston, MA: Pearson Education, Inc., 2009.

[17] A. Gaurav Kumar, SOLID Principles Succinctly, Morrisville, NC: Syncfusion Inc., 2016.