

**Tema**

Arquitectura Tradicional en Capas – C#

**Tutor**

Ing. Eduardo Mauricio Campaña Ortega

MIS.MDU.CCNA.CCIA.

PhD. (c) Ingeniería de Software

PhD. (c) Seguridad Información

Fecha

26/08/2023

ÍNDICE

[1. INTRODUCCIÓN 4](#_Toc129501136)

[2. OBJETIVO 4](#_Toc129501137)

[2.1 OBJETIVO GENERAL 4](#_Toc129501138)

[2.2 OBJETIVOS ESPECÍFICOS 4](#_Toc129501139)

[3. MARCO TEÓRICO 5](#_Toc129501140)

[3.1 DOTNET Y ARQUITECTURA EN CAPAS. 5](#_Toc129501141)

[3.2 USOS DE LA ARQUITECTURA EN CAPAS. 6](#_Toc129501142)

[3.3 POO Y SU RELACIÓN DE LA ARQUITECTURA EN CAPAS. 6](#_Toc129501143)

[3.4 DOMAIN-DRIVEN DESIGN (DDD) 7](#_Toc129501144)

[3.5 DOMAIN-DRIVEN DESIGN Y ARQUITECTURA EN CAPAS. 8](#_Toc129501145)

[3.6 PROGRAMACION ORIENTADA A OBJETOS (POO) 9](#_Toc129501146)

[3.7 VALIDACIONES 12](#_Toc129501147)

[3.8 VALIDACIONES EN LA ARQUITECTURA EN CAPAS. 12](#_Toc129501148)

[3.9 TÉRMINOS 13](#_Toc129501149)

[4. DESARROLLO 15](#_Toc129501150)

[4.1 CREACIÓN DEL PROYECTO Y GENERACIÓN DE SU ESTRUCTURA. 15](#_Toc129501151)

[4.1 CAPA DE ACCESO A DATOS. 17](#_Toc129501152)

[4.2 CAPA DE DOMINIO. 28](#_Toc129501153)

[**4.3 CAPA DE PRESENTACIÓN.** 35](#_Toc129501154)

[5. EJECUCIÓN DEL PROYECTO 43](#_Toc129501155)

[6. CONCLUSIONES 46](#_Toc129501156)

[7. RECOMENDACIONES 46](#_Toc129501157)

[8. BIBLIOGRAFÍA 46](#_Toc129501158)

**ÍNDICE DE IMÁGENES**

[Figura 1. Diagrama de arquitectura en capas. 5](#_Toc129467191)

[Figura 2. Un botón podría ser definida como objeto en capa de vista.. 7](#_Toc129467192)

[Figura 3. Diagrama de DDD. 8](#_Toc129467193)

[Figura 4. Características de POO. 11](#_Toc129467194)

[Figura 5. Ejemplo de validación en sitio web. 12](#_Toc129467195)

[Figura 6. Selección de tipo de aplicación. 17](#_Toc129467196)

[Figura 7. Selección de nombre y ubicación. 17](#_Toc129467197)

[Figura 8. Lista de capas. 18](#_Toc129467198)

[Figura 9. Carpetas AccesoDatos. 18](#_Toc129467199)

[Figura 10. Carpetas Dominio. 18](#_Toc129467200)

[Figura 11. Carpetas Presentación. 19](#_Toc129467201)

[Figura 12. Formas de agregar referencias. 19](#_Toc129467202)

[Figura 13. Creación de clase. 20](#_Toc129467203)

[Figura 14. Lista de archivos por carpeta. 20](#_Toc129467204)

[*Figura 15. Creación de clase.* 29](#_Toc129467205)

[*Figura 16. Lista de archivos por carpeta capa Dominio.* 30](#_Toc129467206)

[*Figura 17. Creación de clase.* 37](#_Toc129467207)

[*Figura 18. Creación de formulario* 38](#_Toc129467208)

[*Figura 19. Lista de archivos por carpeta capa Dominio.* 38](#_Toc129467209)

[*Figura 20. Interface de la vista.* 39](#_Toc129467210)

[Figura 21. Lista de registros 43](#_Toc129467211)

[Figura 22. Retiro de un registro. 43](#_Toc129467212)

[Figura 23. Edición de registro. 44](#_Toc129467213)

[Figura 24. Edición de registro. 44](#_Toc129467214)

[Figura 25. Mensajes de validación. 45](#_Toc129467215)

**ÍNDICE DE TABLAS**

[Tabla 1. Términos 15](#_Toc129467216)

[Tabla 2 Código IEmployeeRepository 21](#_Toc129467217)

[Tabla 3 Código IGenericRepository 22](#_Toc129467218)

[Tabla 4 Código Vista Employee 23](#_Toc129467219)

[Tabla 5 Código EmployeeRepository 24](#_Toc129467220)

[Tabla 6 Código MasterRepository 26](#_Toc129467221)

[Tabla 7 Código Repository 28](#_Toc129467222)

[*Tabla 8 Código EmployeeModel* 31](#_Toc129467223)

[*Tabla 9 Código EntityState* 35](#_Toc129467224)

[*Tabla 10 Código FormEmployee de eventos* 40](#_Toc129467225)

[*Tabla 11 Código DataValidation* 42](#_Toc129467226)

# INTRODUCCIÓN

Este trabajo de investigación y desarrollo tiene como objetivo diseñar e implementar un sistema CRUD utilizando una arquitectura tradicional en capas, programación orientada a objetos (POO), Domain-Driven Design (DDD), Patrones de Arquitectura Empresarial (PEAA) y validaciones.

La arquitectura en capas es un enfoque común para la construcción de aplicaciones empresariales escalables y mantenibles. La POO permite el desarrollo de código reutilizable y modular, lo que facilita el mantenimiento y la evolución del sistema. El DDD se enfoca en la modelización de dominios complejos y ayuda a garantizar que el software refleje fielmente el mundo real. Los patrones de arquitectura empresarial (PEAA) son un conjunto de patrones probados y recomendados para la construcción de sistemas empresariales. Finalmente, las validaciones son una parte crítica del proceso de desarrollo, ya que aseguran que los datos de entrada sean correctos y válidos.

Al combinar estas técnicas y metodologías, se espera crear un sistema robusto y escalable que cumpla con los requisitos funcionales y no funcionales establecidos. El trabajo de investigación y desarrollo se centrará en diseñar y desarrollar un sistema CRUD completo que permita realizar las operaciones básicas de creación, lectura, actualización y eliminación de datos en una base de datos relacional. Además, se implementarán validaciones de datos para garantizar que la información ingresada sea coherente y esté en conformidad con las reglas de negocio establecidas.

La estructura del trabajo constará de 3 secciones claves, siendo primero una introducción teórica sobre todo lo relacionado para comprender el tema fundamental de la arquitectura en capas, seguido de una guía detallada con códigos con la cual se podrá crear un proyecto de CRUD en capas, y por último una serie de conclusiones realizadas por los autores de este trabajo

.

# OBJETIVO

### OBJETIVO GENERAL

El objetivo general de este trabajo de investigación y desarrollo es diseñar y desarrollar un CRUD utilizando Arquitectura Tradicional en Capas, POO, DDD, Patrones de Diseño PEAA y técnicas de validación para demostrar cómo se pueden aplicar estas técnicas y metodologías en el desarrollo de una aplicación eficiente y escalable.

### OBJETIVOS ESPECÍFICOS

* Realizar una revisión teórica de la Arquitectura Tradicional en Capas, POO, DDD, Patrones de Diseño PEAA y técnicas de validación, para entender cómo aplicarlos en el desarrollo de la aplicación CRUD.
* Diseñar el modelo de datos y las diferentes capas de la aplicación (presentación, negocio y acceso a datos), aplicando las técnicas y metodologías seleccionadas.
* Documentar el proceso de diseño y desarrollo de la aplicación CRUD, detallando las decisiones tomadas y las técnicas utilizadas.

# 3. MARCO TEÓRICO

## 3.1 DOTNET Y ARQUITECTURA EN CAPAS.

La arquitectura en capas es [1] una de las formas más comunes de organizar una aplicación, ya que permite separar las diferentes responsabilidades de una aplicación en capas independientes y bien definidas. En el caso de .NET, la arquitectura en capas se basa en la separación de la aplicación en tres capas: la capa de presentación, la capa de negocio y la capa de acceso a datos.

La capa de presentación es la capa en la que se definen los componentes de la interfaz de usuario de la aplicación. Esta capa se encarga de interactuar con el usuario, presentando la información y recogiendo las acciones que este realiza. En .NET, la capa de presentación puede ser implementada a través de diferentes tecnologías, como Windows Forms, WPF o ASP.NET.

La capa de negocio es la capa en la que se definen las reglas y la lógica de la aplicación. Esta capa es responsable de procesar la información que recibe la aplicación desde la capa de presentación y de generar las respuestas y acciones necesarias para devolver el resultado al usuario. En .NET, la capa de negocio se implementa a través de clases que representan los objetos y las entidades del dominio de la aplicación.

La capa de acceso a datos es la capa en la que se accede a los datos almacenados en la aplicación. Esta capa se encarga de interactuar con la base de datos y de realizar las operaciones de consulta y modificación de los datos. En .NET, la capa de acceso a datos se implementa a través de ADO.NET, que proporciona una serie de clases para interactuar con diferentes tipos de bases de datos.

La arquitectura en capas en .NET permite una separación clara de las diferentes responsabilidades de la aplicación y una fácil escalabilidad y mantenimiento de la misma. Además, esta arquitectura se puede combinar con otras metodologías y técnicas, como el patrón MVC, para mejorar aún más la estructura y la funcionalidad de la aplicación

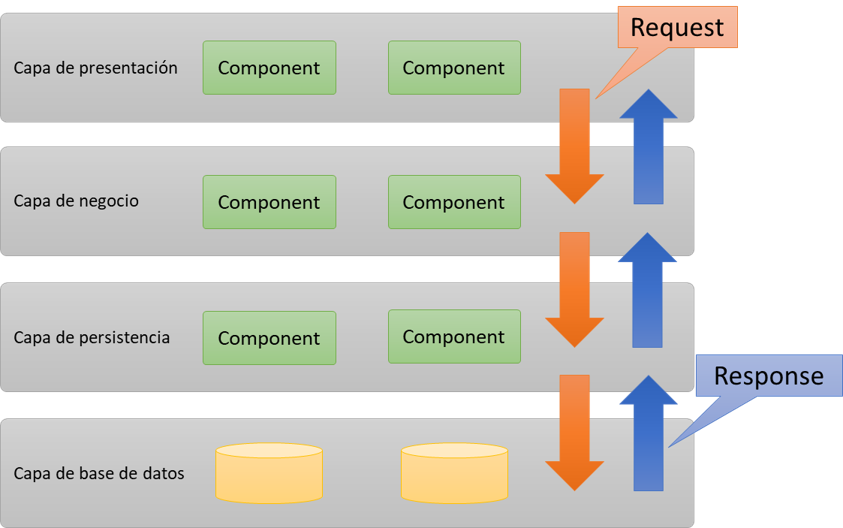


Figura 1. Diagrama de arquitectura en capas.

## USOS DE LA ARQUITECTURA EN CAPAS.

La arquitectura en capas [2] se utiliza en situaciones donde se necesita separar las responsabilidades y los componentes de una aplicación en diferentes capas, para mejorar la calidad, la escalabilidad y la mantenibilidad del software.

Un ejemplo real de la utilización de la arquitectura en capas se encuentra en el desarrollo de sistemas empresariales, como los sistemas de gestión de recursos humanos o los sistemas de gestión de inventario. En estos sistemas, es común utilizar la arquitectura en capas para separar la lógica de la interfaz de usuario, la lógica de negocio y el acceso a la base de datos.

En el caso de un sistema de gestión de recursos humanos, la capa de presentación podría estar formada por la interfaz de usuario de la aplicación, donde se presentan los datos y se recogen las acciones del usuario. La capa de negocio estaría compuesta por las reglas y la lógica de la aplicación, como por ejemplo el cálculo de los salarios o la gestión de las vacaciones. Por último, la capa de acceso a datos se encargaría de acceder a la base de datos para almacenar y recuperar la información necesaria.

La utilización de la arquitectura en capas en este ejemplo permitiría una separación clara de las responsabilidades de cada capa, lo que facilitaría el mantenimiento y la escalabilidad de la aplicación. Además, esta arquitectura permitiría reutilizar las capas en diferentes proyectos o sistemas, lo que puede reducir el tiempo de desarrollo y mejorar la calidad del software.

## POO Y SU RELACIÓN DE LA ARQUITECTURA EN CAPAS.

La programación orientada a objetos (POO) y la arquitectura en capas son dos conceptos diferentes pero complementarios que se utilizan en el desarrollo de software. La POO se refiere a un paradigma de programación que se basa en la creación de objetos que tienen un estado y un comportamiento, y que se comunican entre sí para realizar tareas específicas. Por otro lado, la arquitectura en capas se refiere a una estructura de diseño que separa el sistema en diferentes capas o niveles, cada una con una funcionalidad específica.

La combinación de estos dos conceptos permite crear aplicaciones de software más escalables, flexibles y mantenibles. En una arquitectura en capas, cada capa puede estar compuesta por objetos de diferentes clases, que se comunican entre sí mediante interfaces específicas. Por ejemplo, en una arquitectura en tres capas (presentación, negocio y datos), cada capa podría tener diferentes clases de objetos que representen sus diferentes componentes.

En la capa de presentación, se podrían crear objetos que representen los diferentes elementos de la interfaz de usuario, como botones, campos de entrada, etc. En la capa de negocio, se podrían crear objetos que representen las diferentes reglas de negocio de la aplicación, como el proceso de compra de un producto en una tienda en línea. En la capa de datos, se podrían crear objetos que representen las diferentes entidades de la base de datos, como los productos disponibles en la tienda.

Cada una de estas capas tendría su propia funcionalidad específica, y se comunicarían entre sí mediante interfaces bien definidas. Por ejemplo, la capa de presentación podría llamar a métodos de la capa de negocio para realizar una operación de compra, y la capa de negocio podría llamar a métodos de la capa de datos para obtener o modificar información en la base de datos.



Figura 2. Un botón podría ser definida como objeto en capa de vista..

## DOMAIN-DRIVEN DESIGN (DDD)

Domain-Driven Design (DDD) es [3] una metodología de diseño de software que se centra en el negocio y su modelo de dominio. La idea principal de DDD es que el software debe estar basado en el conocimiento y entendimiento del dominio de negocio, es decir, en la problemática específica que se quiere solucionar.

En DDD, se busca que el software sea más fácil de entender, de mantener y de extender, mediante la creación de un modelo rico en conceptos y una separación clara entre las diferentes capas de la arquitectura. Una de las principales ventajas de DDD es que, al centrarse en el negocio, el equipo de desarrollo y los stakeholders pueden comunicarse de manera más efectiva y comprender mejor los requisitos y objetivos del proyecto.

El modelo de dominio en DDD es la representación de los conceptos, reglas y relaciones que son importantes en el dominio de negocio. Este modelo debe ser entendido por todas las personas involucradas en el proyecto, desde los desarrolladores hasta los usuarios finales. El objetivo es que el modelo de dominio sea tan claro y preciso que pueda ser utilizado como una fuente única de verdad para todo el proyecto.

Para aplicar DDD, se pueden seguir una serie de pasos:

* Comprender el dominio: es fundamental que el equipo de desarrollo tenga un conocimiento profundo del dominio de negocio en el que está trabajando. Esto implica hablar con expertos en el tema, comprender los procesos de negocio, etc.
* Crear el modelo de dominio: a partir del conocimiento obtenido en el primer paso, se debe crear un modelo de dominio que refleje los conceptos y relaciones importantes del negocio.
* Establecer los límites del contexto: los límites del contexto son los límites dentro de los cuales el modelo de dominio es válido. Estos límites se establecen para evitar que el modelo de dominio se vuelva demasiado grande y complejo.
* Definir las capas de la arquitectura: se deben definir las diferentes capas de la arquitectura, como la capa de presentación, la capa de dominio y la capa de infraestructura.
* Aplicar patrones de diseño: se pueden aplicar patrones de diseño como el patrón Repository o el patrón Service para implementar el modelo de dominio de manera efectiva.
* Iterar y mejorar: DDD es un enfoque iterativo, por lo que es importante seguir mejorando y refinando el modelo de dominio y la arquitectura a medida que se avanza en el proyecto.

Un ejemplo de cómo se puede aplicar DDD sería en un proyecto de una tienda en línea. En este caso, el equipo de desarrollo debería comprender los procesos de negocio de la tienda, como la gestión de productos, pedidos, pagos, etc. A partir de esta comprensión, se podría crear un modelo de dominio que refleje los conceptos y relaciones importantes para la tienda, como productos, categorías, clientes, pedidos, etc. Luego se podría establecer los límites del contexto y definir las diferentes capas de la arquitectura. Por último, se podrían aplicar patrones de diseño para implementar el modelo de dominio e iterar y mejorar el modelo y la arquitectura a medida que se avanza en el proyecto.

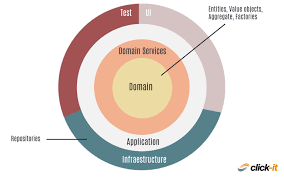


Figura 3. Diagrama de DDD.

## DOMAIN-DRIVEN DESIGN Y ARQUITECTURA EN CAPAS.

El Domain-Driven Design (DDD) y la arquitectura en capas [4] son dos enfoques importantes en el desarrollo de software que pueden trabajar juntos de manera efectiva para crear sistemas más escalables, mantenibles y centrados en el negocio.

En la arquitectura en capas, el software se divide en capas lógicas separadas, cada una con una responsabilidad clara y definida. Esto permite una separación de intereses y una mayor flexibilidad en el cambio y mantenimiento del sistema. Por otro lado, en DDD, se enfoca en el diseño de un modelo de dominio rico y bien definido que refleje el lenguaje utilizado por los expertos del negocio.

Estas dos metodologías se complementan de la siguiente manera: en DDD, la capa del dominio es considerada como el núcleo del sistema, ya que aquí es donde se definen las entidades, objetos de valor y reglas de negocio. Las otras capas, como la de aplicación, la de infraestructura, y la de presentación, se construyen alrededor de la capa del dominio para ofrecer soporte y facilitar su uso.

En la capa de aplicación, se definen los casos de uso y se coordinan las acciones en las diferentes capas del sistema para implementar las funcionalidades de negocio. En la capa de infraestructura, se manejan temas como la persistencia de datos y las integraciones con sistemas externos. En la capa de presentación, se maneja la interacción con el usuario y se proporciona una interfaz gráfica o de servicios.

Para concluir DDD y la arquitectura en capas trabajan juntos para proporcionar una estructura clara y flexible en el diseño y desarrollo de sistemas de software, centrándose en la comprensión y resolución de problemas del negocio en lugar de en los aspectos técnicos del software.

## PROGRAMACION ORIENTADA A OBJETOS (POO)

Programación Orientada a Objetos (POO, por sus siglas en inglés OOP, Object-Oriented Programming) es [5] un paradigma de programación que se basa en el concepto de "objetos", que representan entidades reales o virtuales con atributos y comportamientos.

En la programación orientada a objetos, los objetos interactúan entre sí a través de mensajes, y cada objeto es una instancia de una clase, que es una plantilla o modelo para la creación de objetos. Las clases pueden ser utilizadas para crear objetos similares y compartir atributos y comportamientos.

La programación orientada a objetos proporciona una abstracción y encapsulamiento de los datos y comportamientos de los objetos, lo que permite una mejor organización y reutilización del código. También promueve el modularidad y la separación de preocupaciones, lo que facilita la mantenibilidad y la escalabilidad del software.

Características de POO:

* **Abstracción:** Es un concepto clave que permite representar entidades reales o virtuales de manera simplificada y resumida. En la programación orientada a objetos, la abstracción se logra a través de la creación de clases, que son plantillas o modelos que describen los atributos y comportamientos de un objeto. Las clases permiten ocultar los detalles técnicos y proporcionar una representación más fácil y comprensible de las entidades.

La programación orientada a objetos se sabe que, de alguna manera, trata de "modelizar" los elementos del mundo real. En el mundo existe un universo de objetos que colaboran entre sí para realizar tareas de los sistemas. Llevado al entorno de la programación, también se debe programar una serie de clases a partir de las cuales se puedan instanciar objetos que colaboran entre sí para la resolución de problemas. Si se asume esto, a la vista de las situaciones que ocurren en el mundo real, se puede entender la abstracción.

Características:

* + *Simplificación*: Representa entidades reales o virtuales de manera más sencilla y resumida.
  + *Ocultamiento de detalles*: Oculta los detalles técnicos y permite una representación más fácil y comprensible de las entidades.
  + *Reutilización de código*: Permite la creación de clases que pueden ser reutilizadas en diferentes proyectos y situaciones.
  + *Mejor comprensión*: Permite una mejor comprensión y organización del código y las entidades representadas.
* **Encapsulamiento:** Se refiere a la protección de los datos y comportamientos de un objeto de su acceso externo. El encapsulamiento se logra a través de la creación de clases, que contienen atributos y métodos que describen los datos y comportamientos de un objeto. Estos atributos y métodos son accesibles solo dentro de la clase y no pueden ser modificados o accedidos desde fuera de la clase.

Características:

* + *Protección de datos*: Protege los datos y comportamientos de un objeto de su acceso externo, lo que promueve la seguridad y la integridad de los datos.
  + *Control de acceso*: Permite definir el nivel de acceso a los atributos y métodos de una clase, lo que facilita el control sobre la información.
  + *Mejor organización*: Permite una mejor organización y separación de preocupaciones, lo que facilita la mantenibilidad y escalabilidad del software.
  + *Modificación segura*: Protege los datos y comportamientos de un objeto de ser modificados por error o de manera accidental.
* **Herencia:** Permite a los programadores crear nuevas clases a partir de clases existentes. La herencia permite que las propiedades y comportamientos de una clase sean heredados por una clase hija, lo que facilita la reutilización y la organización del código

Características:

* + *Composición:* Es cuando unos elementos están compuestos de otros, o que unos elementos están presentes en otros. Por ejemplo, el sistema respiratorio y los pulmones, la nariz, etc. Se puede decir que los pulmones están dentro del sistema respiratorio, así como dentro de los pulmones se encuentran bronquios y alvéolos. En esta jerarquía de elementos se tiene composición porque donde unos forman parte de otros. En una factura también se puede decir que puede haber una jerarquía de composición. La factura tiene un cliente, varios conceptos facturables, un impuesto, etc.
  + *Clasificación*: Este tipo de jerarquización indica que unos elementos son una especialización de otros. Por ejemplo, los animales, donde se tiene vertebrados e invertebrados. Luego, dentro de los vertebrados se encuentran aves, reptiles, mamíferos, etc. En los mamíferos se encuentran perros, vacas, conejos... Éste es el tipo de jerarquización en que quiero que te fijes.
  + *Reutilización de código*: Permite la reutilización de código y funcionalidades de una clase padre en una clase hija, lo que ahorra tiempo y esfuerzo al desarrollador.
  + *Organización de clases*: Permite la organización de clases en una jerarquía, lo que facilita la comprensión y la estructuración del software.
* **Polimorfismo:** Permite a los objetos de diferentes clases comportarse de manera similar aunque tengan diferentes implementaciones internas. El polimorfismo permite que un objeto de una clase pueda ser tratado como un objeto de otra clase si cumple con ciertos requisitos de tipo o interfaz.

Características:

* + *Flexibilidad*: Proporciona una mayor flexibilidad y adaptabilidad al software, permitiendo que objetos de diferentes clases se comporten de manera similar.
  + *Interfaces*: Permite definir una interfaz común para objetos de diferentes clases, lo que facilita la interoperabilidad y el intercambio de información entre objetos.

Imagen que contiene Diagrama

Descripción generada automáticamente

Figura 4. Características de POO.

Ventajas de POO:

* **Separación de responsabilidades**: Permite la separación de preocupaciones (SoC), ya que cada capa tiene una sola responsabilidad.
* **Fácil de desarrollar**: Este estilo arquitectónico es especialmente fácil de implementar, además de que es muy conocido y una gran mayoría de las aplicaciones la utilizan.
* **Fácil de probar**: Debido a que la aplicación construida por capas es posible ir probando de forma individual cada capa, lo que permite probar por separada cada capa.
* **Fácil de mantener**: Debido a que cada capa hace una tarea muy específica, es fácil detectar el origen de un bug para corregirlo, o simplemente se puede identificar donde se debe aplicar un cambio.
* **Seguridad:** La separación de capas permite el aislamiento de los servidores en subredes diferentes, lo que hace más difícil realizar ataques.

## VALIDACIONES

Las validaciones en software son [6] mecanismos implementados en una aplicación para verificar que los datos proporcionados por el usuario o el sistema son correctos y cumplen con ciertas reglas o restricciones. Estas reglas pueden incluir comprobaciones de formato, longitud, tipo de datos, valores mínimos y máximos, entre otros criterios.

Las validaciones son una parte esencial del desarrollo de software, ya que ayudan a garantizar la integridad y precisión de los datos que se almacenan y procesan en una aplicación. Además, también pueden mejorar la experiencia del usuario al proporcionar mensajes claros y precisos sobre los errores de entrada de datos.

Algunos ejemplos comunes de validaciones en software incluyen:

* Validaciones de formato: verificar que los datos proporcionados por el usuario se ajusten a un formato específico (por ejemplo, una dirección de correo electrónico válida).
* Validaciones de rango: verificar que los valores proporcionados estén dentro de un rango específico (por ejemplo, una fecha de nacimiento debe estar en un rango de fechas válidas).
* Validaciones de unicidad: verificar que los valores proporcionados no estén duplicados (por ejemplo, un número de identificación único).
* Validaciones de integridad referencial: verificar que los datos relacionales estén vinculados correctamente entre sí (por ejemplo, un pedido solo puede tener un cliente válido).

En general, las validaciones son una parte importante de cualquier proceso de desarrollo de software, ya que pueden ayudar a evitar errores y mejorar la calidad del producto final..

}

Figura 5. Ejemplo de validación en sitio web.

## VALIDACIONES EN LA ARQUITECTURA EN CAPAS.

En una arquitectura en capas, las validaciones se [7] pueden agregar en varias capas del sistema para garantizar que los datos sean precisos y coherentes en todo momento. Algunas de las capas donde se pueden agregar validaciones son:

* Capa de presentación: Esta capa es responsable de interactuar con el usuario final y recopilar los datos ingresados por el usuario. Aquí se pueden agregar validaciones en el lado del cliente para garantizar que los datos ingresados cumplan con ciertas reglas y restricciones antes de enviarlos al servidor.
* Capa de servicio: Esta capa es responsable de procesar los datos y proporcionar la lógica de negocio de la aplicación. Aquí se pueden agregar validaciones para garantizar que los datos cumplan con las reglas y restricciones de negocio y que sean válidos antes de procesarlos.
* Capa de persistencia: Esta capa es responsable de interactuar con la base de datos y almacenar los datos en ella. Aquí se pueden agregar validaciones para garantizar que los datos sean consistentes y válidos antes de almacenarlos en la base de datos.
* Al agregar validaciones en diferentes capas del sistema, se puede garantizar que los datos ingresados sean precisos y coherentes en todo momento, lo que ayuda a mejorar la calidad de los datos y la eficiencia del sistema. Además, también se pueden proporcionar mensajes claros y precisos sobre los errores de entrada de datos, lo que mejora la experiencia del usuario.

Es mejor agregar las validaciones en la capa de dominio porque esta capa representa la lógica de negocio y es la encargada de asegurar que las entidades cumplan con las reglas y restricciones que se han definido para el sistema. Al incluir las validaciones en la capa de dominio, se garantiza que las reglas de negocio sean consistentes en todas las capas del sistema y se evita la repetición de código en múltiples capas.

Además, la capa de dominio suele ser la capa más estable y menos propensa a cambios en comparación con las capas de presentación o de acceso a datos. Al colocar las validaciones en la capa de dominio, se reduce la posibilidad de tener que hacer cambios significativos en otras capas del sistema en caso de que se cambien o agreguen nuevas reglas de negocio.

Es decir, al agregar las validaciones en la capa de dominio se obtiene un código más limpio, mantenible y coherente en todas las capas del sistema, lo que a su vez se traduce en un software más robusto y confiable.

## TÉRMINOS

Antes de realizar la practica presente es necesario conocer algunos métodos que son necesarios para entender los códigos y algunas definiciones que se utilizaran durante la práctica.

Tabla 1. Términos

|  |  |
| --- | --- |
| TextBox | Un TextBox es un control de usuario en la interfaz gráfica de usuario (GUI) que permite a los usuarios ingresar texto en una aplicación. Es una de las características más comunes en la mayoría de las aplicaciones de software y se utiliza para recibir entrada de texto de los usuarios |
| Label | Un label es una etiqueta o marca descriptiva que se asigna a un elemento o grupo de elementos para identificarlos y organizarlos. En programación, los labels pueden ser utilizados en diversos contextos, como en la identificación de formularios, botones, textos, entre otros elementos gráficos en una interfaz de usuario. |
| CSS |  |
| .NET | Es una plataforma de desarrollo de software de Microsoft que permite desarrollar aplicaciones para una variedad de dispositivos y sistemas operativos. |
| .NET Core | Es una implementación de .NET que se centra en la portabilidad y el rendimiento. Se utiliza para crear aplicaciones que se ejecutan en múltiples plataformas, incluyendo Windows, macOS y Linux. |
| .NET Framework | Es la implementación original de .NET que se utiliza principalmente para aplicaciones Windows y se ejecutan en el sistema operativo Windows. |
| .NET Standard | Es un estándar que define una API común que se puede utilizar en todas las implementaciones de .NET. Se utiliza para asegurar la compatibilidad entre diferentes implementaciones de .NET. |
| API | API es el acrónimo de "Application Programming Interface" y se refiere a un conjunto de reglas y convenciones que se utilizan para comunicar una aplicación con otras partes o componentes. |
| CLI | CLI es el acrónimo de "Common Language Infrastructure" y se refiere a la infraestructura común que se utiliza para ejecutar aplicaciones .NET. |
| CRUD | Acrónimo de Create, Read, Update, Delete, que hace referencia a las operaciones básicas de la persistencia de datos. |
| Capa de Presentación | Capa que se encarga de presentar la información al usuario final y de recibir sus interacciones. |
| Capa de Dominio | Capa que se encarga de representar el modelo de dominio de la aplicación y de aplicar la lógica de negocio. |
| Capa de Acceso a Datos | Capa que se encarga de interactuar con la base de datos y realizar operaciones de persistencia. |
| Patrones de Diseño | Soluciones probadas y comprobadas a problemas comunes de diseño de software. |
| MVC | Modelo-Vista-Controlador, un patrón de diseño arquitectónico que separa la aplicación en tres componentes principales. |
| Seguridad | Conjunto de medidas y procedimientos destinados a proteger la información y los sistemas informáticos de posibles amenazas y ataques. |
| Autenticación | Proceso de identificación de un usuario mediante el ingreso de credenciales (usuario y contraseña, por ejemplo). |
| Autorización | Proceso mediante el cual se otorgan o deniegan permisos a un usuario o grupo de usuarios para acceder a ciertas partes de la aplicación. |
| Encriptación | Proceso mediante el cual se transforma la información de manera que sólo pueda ser leída por aquellos que tengan la clave para decodificarla. |
| Validación de Datos | Proceso mediante el cual se comprueba si los datos ingresados cumplen con ciertos criterios, como el tipo de dato, la longitud o el formato. |
| Herencia | Mecanismo de la POO mediante el cual una clase puede heredar los atributos y métodos de otra. |

# 4. DESARROLLO

## 4.1 CREACIÓN DEL PROYECTO Y GENERACIÓN DE SU ESTRUCTURA.

Como se indicó al inicio de este trabajo en el presente trabajo se utilizará una arquitectura en capas para desarrollar un CRUD, por lo cual será necesario definir adecuadamente la estructura del proyecto y los archivos del mismo antes de realizar cualquier codificación.

Lo primero es crear una solución en blanco para ir agregando poco a poco las distintas capas del proyecto, para ello abra el IDE de Visual Studio, seleccione crear nuevo proyecto, y en el tipo elija solución en blanco.

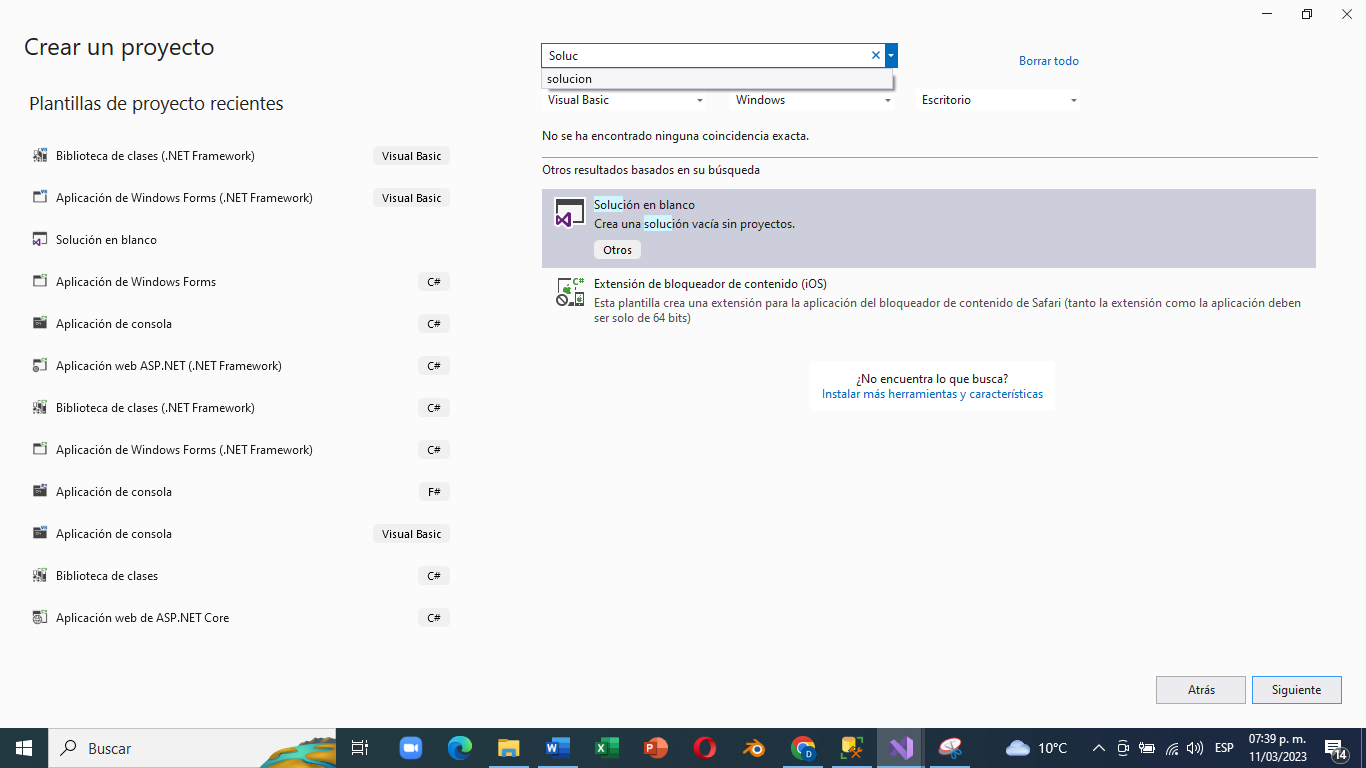


Figura 6. Selección de tipo de aplicación.

Coloque el nombre “LayeredAppi y seleccione la ubicación del proyecto en donde se desee guardar.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Figura 7. Selección de nombre y ubicación.

Una vez creada la solución en blanco será necesario crear 3 proyectos dentro de la solución en blanco para que cada uno de ellos sea una capa de la aplicación entonces deberá crear una aplicación de Windows Forms de tipo NET FRAMEWORK llamada presentación, mientras que será necesario crear 2 biblioteca de clases también de NET FRAMEWORK con los nombres dominio y acceso a datos.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Figura 8. Lista de capas.

Una vez creada las capas será necesario crear una serie de carpetas para poder alamcenar los distintos archivos de cada capa según su funcionalidad de tal manera se tendra lo siguiente.

* AccesoDatos: 3 carpetas con los nombres de Contratos, Entidades y Repositorios respectivamente.
* Dominio: 3 carpetas con los nombres de Modelos, Objetos de valores y Servicios respectivamente.
* Presentación: 3 carpetas con los nombres de Formularios, Modelos de vista y Soportes respectivamente.

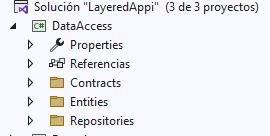


Figura 9. Carpetas AccesoDatos.

Texto

Descripción generada automáticamente

Figura 10. Carpetas Dominio.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Figura 11. Carpetas Presentación.

Por ultimo será necesario establecer las relaciones entre las distintas capas, como se indico en la teoría una capa solo puede comunicarse con la de abajo, por lo cual la capa de presentación solo podrá comunicarse con la dominio y la de dominio con acceso a datos, para ello de clic derecho sobre una de las capas y seleccione referencias, busque en proyectos y seleccione el proyecto con el cual tiene comunicación la capa.

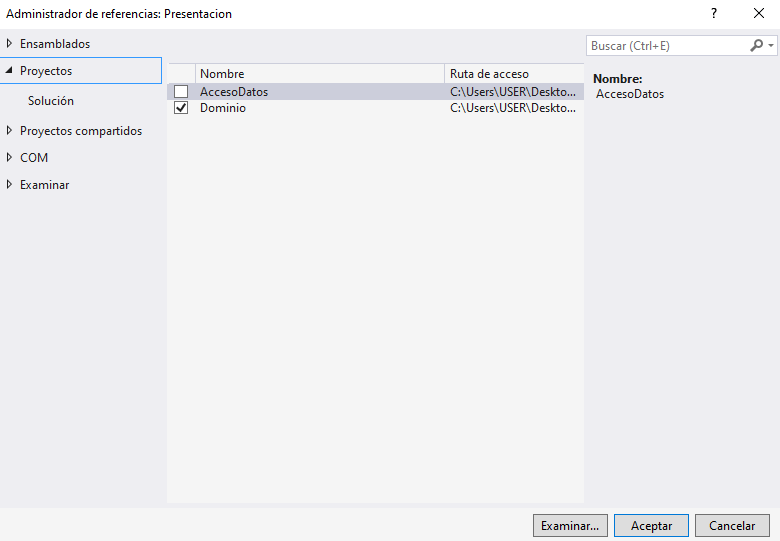


Figura 12. Formas de agregar referencias.

## 4.1 CAPA DE ACCESO A DATOS.

En la capa de acceso a datos se tiene todo lo relacionado con la conexión a la base de datos, los métodos SQL a ser ejecutados y que recibirán los datos de la base de datos la configuración de las cadenas de conexión y las interfaces de comunicación con la capa de Dominio, para lograr esto será necesario tener una serie de clase de C# en las distintas carpetas de esta capa, por lo cual deberá crear las siguientes clases en las distintas carpetas de la aplicación.

* Carpeta contratos: las clases IEmployeeRepository y IGenericRepository.
* Carpeta Entidades: la clase Employee.
* Carpeta Repositorios: las clases EmployeeRepository, MasterRepository y Repository.

Para crear una clase de clic derecho sobre la carpeta y seleccione nueva clase, incluye los nombres indicados y proceda a dar clic en crear.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Figura 13. Creación de clase.

Al final tendrá los siguientes archivos como en la siguiente imagen.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Figura 14. Lista de archivos por carpeta.

Tabla Código IEmployeeRepository

|  |
| --- |
| using DataAccess.Entities;  using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace DataAccess.Contracts  {  public interface IEmployeeRepository: IGenericRepository<Employee>  {  //otros metodos  //IEnumerable<Employee> GetBySalary();  }  } |

Tabla Código IGenericRepository

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace DataAccess.Contracts  {  public interface IGenericRepository<Entity> where Entity : class  {  int Add(Entity entity);  int Adit(Entity entity);  int Remove(int idPk);  IEnumerable<Entity> GetAll();  }  } |

Tabla Código Vista Employee

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace DataAccess.Entities  {  public class Employee  {  public int idPk { get; set; }  public string idNumber { get; set; }  public string name { get; set; }  public string mail { get; set; }  public DateTime birthday { get; set; }    }  } |

Tabla Código EmployeeRepository

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using DataAccess.Contracts;  using DataAccess.Entities;  using System.Data;  using System.Data.SqlClient;  namespace DataAccess.Repositories  {  public class EmployeeRepository : MasterRepository, IEmployeeRepository  {  //Campos  private string selectAll;  private string insert;  private string update;  private string delete;  //Propiedades  //::  //Constructores  public EmployeeRepository() {  selectAll = "select \*from Employee";  insert = "insert into Employee values (@idNumber,@name,@mail,@birthday)";  update = "update Employee set IdNumber=@idNumber,Name=@name,Mail=@mail,Birthday=@birthday where idPk=@idPK";  delete = "delete from Employee where idPk=@idPK";  }  //Metodos, Comportamientos  public int Add(Employee entity)  {  parameters = new List<SqlParameter>();  parameters.Add(new SqlParameter("@idNumber", entity.idNumber));  parameters.Add(new SqlParameter("@name", entity.name));  parameters.Add(new SqlParameter("@mail", entity.mail));  parameters.Add(new SqlParameter("@birthday", entity.birthday));  return ExecuteNonQuery(insert);  }  public int Adit(Employee entity)  {  parameters = new List<SqlParameter>();  parameters.Add(new SqlParameter("@idPK", entity.idPk));  parameters.Add(new SqlParameter("@idNumber", entity.idNumber));  parameters.Add(new SqlParameter("@name", entity.name));  parameters.Add(new SqlParameter("@mail", entity.mail));  parameters.Add(new SqlParameter("@birthday", entity.birthday));  return ExecuteNonQuery(update);  }  public IEnumerable<Employee> GetAll()  {  var tableResult = ExecuteReader(selectAll);  var listEmployees = new List<Employee>();  foreach(DataRow item in tableResult.Rows)  {  listEmployees.Add(new Employee  {  idPk = Convert.ToInt32(item[0]),  idNumber = item[1].ToString(),  name = item[2].ToString(),  mail = item[3].ToString(),  birthday = Convert.ToDateTime(item[4]),  });  }  return listEmployees;  }  public int Remove(int idPk)  {  parameters = new List<SqlParameter>();  parameters.Add(new SqlParameter("@idPK", idPk));  return ExecuteNonQuery(delete);  }  }  } |

Tabla Código MasterRepository

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Data.SqlClient;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using System.Data;  namespace DataAccess.Repositories  {  public abstract class MasterRepository:Repository  {  protected List<SqlParameter> parameters;  protected int ExecuteNonQuery(string transactSql)  {  using (var connection = GetConnection())  {  connection.Open();  using (var command = new SqlCommand())  {  command.Connection= connection;  command.CommandText= transactSql;  command.CommandType= CommandType.Text;  foreach(SqlParameter item in parameters)  {  command.Parameters.Add(item);  }  int result=command.ExecuteNonQuery();  parameters.Clear();  return result;  }  }  }  protected DataTable ExecuteReader(string transactSql)  {  using (var connection = GetConnection())  {  connection.Open();  using (var command = new SqlCommand())  {  command.Connection = connection;  command.CommandText = transactSql;  command.CommandType = CommandType.Text;  SqlDataReader reader = command.ExecuteReader();  using(var table = new DataTable()) {  table.Load(reader);  reader.Dispose();  return table;  }  }  }  }  }  } |

Tabla Código Repository

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using System.Configuration;  using System.Data;  using System.Data.SqlClient;  namespace DataAccess.Repositories  {  public abstract class Repository  {  private readonly string connectionString;  public Repository ()  {  connectionString = ConfigurationManager.ConnectionStrings["connMyCompany"].ToString();  }  protected SqlConnection GetConnection()  {  return new SqlConnection(connectionString);  }  }  } |

## 4.2 CAPA DE DOMINIO.

La capa de Dominio tiene la función esencial de representar el conocimiento del negocio y la lógica del dominio de la aplicación. Por tanto, es común que en esta capa se definan los modelos o entidades que representan los objetos de negocio, así como las reglas de negocio que rigen su comportamiento.

En la capa de dominio, además de las entidades de negocio y las reglas de negocio, se pueden definir también los repositorios o interfaces para acceder a los datos almacenados en la base de datos, los servicios de dominio que encapsulan la lógica de negocio, y los eventos de dominio que permiten la comunicación entre los diferentes componentes de la aplicación.

A continuación, se presenta la lista de clases creadas en cada carpeta.

* Carpeta Modelos: la clase EmployeeModel.
* Carpeta ObjetosDeValores: la clase EntityState.
* Carpeta Repositorios: las clases EmployeeRepository, MasterRepository y Repository.

Para crear una clase de clic derecho sobre la carpeta y seleccione nueva clase, incluye los nombres indicados y proceda a dar clic en crear.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

*Figura 15. Creación de clase.*

Al final tendrá los siguientes archivos como en la siguiente imagen.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

*Figura 16. Lista de archivos por carpeta capa Dominio.*

*Tabla 8 Código EmployeeModel*

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using DataAccess.Contracts;  using DataAccess.Entities;  using DataAccess.Repositories;  using Domain.ValueObjects;  using System.ComponentModel.DataAnnotations;  namespace Domain.Models  {  public class EmployeeModel : IDisposable  {  private int idPk;  private string idNumber;  private string name;  private string mail;  private DateTime birthday;  private int age;  private IEmployeeRepository employeeRepository;  public EntityState State  {  private get;  set;  }  private List<EmployeeModel> listEmployees;  // PROPIEDADES/MODELOS DE VISTA/ VALIDAR DATOS  public int IdPk { get => idPk; set => idPk = value; }  [Required(ErrorMessage = "El campo numero de identificación es requerida")]  [RegularExpression("([0-9]+)", ErrorMessage = "El numero de identificacion solo debe ser numeros")]  [StringLength(maximumLength: 10, MinimumLength = 10, ErrorMessage = "El numero de identificacion debe ser de 10 digitos")]  public string IdNumber { get => idNumber; set => idNumber = value; }  [Required]  [RegularExpression("^[a-zA-Z ]+$", ErrorMessage = "El campo de nombre solo debe ser ingresado con letras")]  [StringLength(maximumLength: 100, MinimumLength = 3)]  public string Name { get => name; set => name = value; }  [Required]  [EmailAddress]  public string Mail { get => mail; set => mail = value; }  public DateTime Birthday { get => birthday; set => birthday = value; }  public int Age { get => age; private set => age = value; }  public EmployeeModel()  {  employeeRepository = new EmployeeRepository();  }  public string SaveChanges()  {  string message = null;  try  {  var employeeDataModel = new Employee();  employeeDataModel.idPk = idPk;  employeeDataModel.name = name;  employeeDataModel.mail = mail;  employeeDataModel.birthday = birthday;  switch (State)  {  case EntityState.Added:  employeeRepository.Add(employeeDataModel);  message = "Registrado correctamente";  break;  case EntityState.Modified:  employeeRepository.Adit(employeeDataModel);  message = "Editado correctamente";  break;  case EntityState.Delete:  employeeRepository.Remove(idPk);  message = "Eliminado correctamente";  break;  }  }  catch (Exception ex)  {  System.Data.SqlClient.SqlException sqlEx = ex as System.Data.SqlClient.SqlException;  if (sqlEx != null && sqlEx.Number == 2627)  message = "Duplicate record";  else  message = ex.ToString();  }  return message;  }  public List<EmployeeModel> GetAll()  {  var employeeDataModel = employeeRepository.GetAll();  listEmployees = new List<EmployeeModel>();  foreach (Employee item in employeeDataModel)  {  listEmployees.Add(new EmployeeModel  {  idPk = item.idPk,  idNumber = item.idNumber,  name = item.name,  mail = item.mail,  birthday = item.birthday,  age = CalculateAge(birthday)  });  }  return listEmployees;  }  public IEnumerable<EmployeeModel> FindById(string filter)  {  return listEmployees.FindAll(e => e.IdNumber.Contains(filter) || e.Name.Contains(filter));  }  private int CalculateAge(DateTime dateOfBirth)  {  DateTime currentDate = DateTime.Now;  int age = currentDate.Year - dateOfBirth.Year;  // Restar un año si el mes y el día de la fecha de nacimiento aún no han pasado este año  if (currentDate.Month < dateOfBirth.Month || (currentDate.Month == dateOfBirth.Month && currentDate.Day < dateOfBirth.Day))  {  age--;  }  return age;  }  public void Dispose()  {    }  }  } |

*Tabla 9 Código EntityState*

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace Domain.ValueObjects  {  public enum EntityState  {  Added,  Delete,  Modified  }  } |

**4.3 CAPA DE PRESENTACIÓN.**

Esta capa como se indicó en la teoría es la más cercana al usuario y se encarga de mostrar la información al usuario y de recibir sus acciones, es decir, la interacción entre el usuario y el sistema. En general, en esta capa se encuentra todo lo relacionado con la interfaz gráfica de usuario (GUI), como ventanas, formularios, botones, menús, etc.

En esta capa se suele tener código que se encarga de validar la entrada de datos del usuario (lo cual en este caso no funcionara de dicha manera puesto que en la aplicación estas validaciones fueron realizadas en la capa de Dominio) y de enviar esa información a la capa de aplicación para que realice las operaciones correspondientes. También puede incluirse en esta capa la lógica de presentación, que se encarga de mostrar los datos en la interfaz de usuario de manera adecuada y atractiva para el usuario.

A continuación, se presenta la lista de clases y archivos creados en cada carpeta.

* Carpeta Formularios: Formulario FormEmployee.
* Carpeta Soportes: la clase DataValidation.

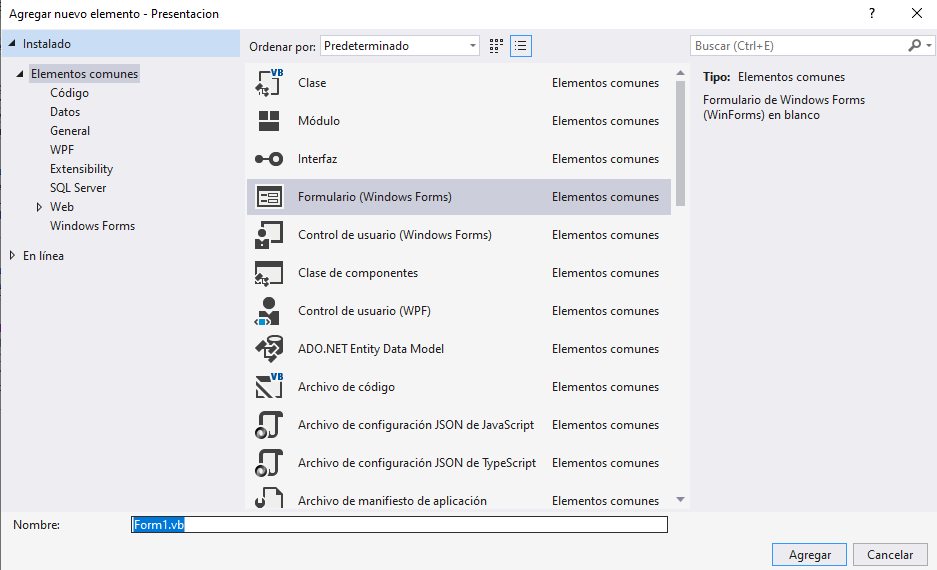
Para crear una clase de clic derecho sobre la carpeta y seleccione nueva clase, incluye los nombres indicados y proceda a dar clic en crear.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

*Figura 17. Creación de clase.*

Por otro lado para crear el formulario de Windows forms debe dar clic sobre la carpeta y seleccionar Windows forms, agregue el nombre y dar clic en crear.



*Figura 18. Creación de formulario*

Al final tendrá los siguientes archivos como en la siguiente imagen.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

*Figura 19. Lista de archivos por carpeta capa Dominio.*

Dentro del formulario creado deberá agregar un DataGriedVIew, un panel, dentro del panel 3 textbox y un date picker, además de 3 botones para crear, editar y quitar un registro y uno para salvar el registro, como se ve a continuación.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

*Figura 20. Interface de la vista.*

*Tabla 10 Código FormEmployee de eventos*

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.ComponentModel;  using System.Data;  using System.Drawing;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using System.Windows.Forms;  using Domain.Models;  using Domain.ValueObjects;  namespace Presentation.Forms  {  public partial class FormEmployee : Form  {  private EmployeeModel employee = new EmployeeModel();  public FormEmployee()  {  InitializeComponent();  panel1.Enabled= false;  }  private void FormEmployee\_Load(object sender, EventArgs e)  {  ListEmployees();  }  private void ListEmployees()  {  try  {  dataGridView1.DataSource = employee.GetAll();  }  catch(Exception ex) {  MessageBox.Show(ex.ToString());  }  }  private void textBox1\_TextChanged(object sender, EventArgs e)  {  dataGridView1.DataSource = employee.FindById(textBox1.Text);  }  private void button4\_Click(object sender, EventArgs e)  {  employee.IdNumber = txtIdNumber.Text;  employee.Mail= txtMail.Text;  employee.Name= txtName.Text;  employee.Birthday = txtBirthday.Value;  bool valid = new Helps.DataValidation(employee).Validate();  if (valid == true)  {  string result = employee.SaveChanges();  MessageBox.Show(result);  ListEmployees();  Restart();  }  }  private void Restart()  {  panel1.Enabled= false;  txtIdNumber.Clear();  txtMail.Clear();  txtName.Clear();  }  private void button1\_Click(object sender, EventArgs e)  {  panel1.Enabled = true;  employee.State = EntityState.Added;  }  private void button2\_Click(object sender, EventArgs e)  {  if (dataGridView1.SelectedRows.Count > 0)  {  panel1.Enabled = true;  employee.State= EntityState.Modified;  employee.IdPk = Convert.ToInt32(dataGridView1.CurrentRow.Cells[0].Value);  txtIdNumber.Text = dataGridView1.CurrentRow.Cells[1].Value.ToString();  txtName.Text = dataGridView1.CurrentRow.Cells[2].Value.ToString();  txtMail.Text = dataGridView1.CurrentRow.Cells[3].Value.ToString();  txtBirthday.Value = Convert.ToDateTime(dataGridView1.CurrentRow.Cells[4].Value.ToString());  }  else  MessageBox.Show("Selecciona una fila");  }  private void button3\_Click(object sender, EventArgs e)  {  if (dataGridView1.SelectedRows.Count > 0)  {    employee.State = EntityState.Delete;  employee.IdPk = Convert.ToInt32(dataGridView1.CurrentRow.Cells[0].Value);  string result = employee.SaveChanges();  MessageBox.Show(result);  ListEmployees();  }  else  MessageBox.Show("Selecciona una fila");  }  }  } |

*Tabla 11 Código DataValidation*

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using System.ComponentModel.DataAnnotations;  namespace Presentation.Helps  {  public class DataValidation  {  private ValidationContext context;  private List<ValidationResult> results;  private bool valid;  private string message;  public DataValidation(object instance)  {  context = new ValidationContext(instance);  results= new List<ValidationResult>();  valid = Validator.TryValidateObject(instance, context,results,true);  }  public bool Validate()  {  if(valid == false)  {  foreach(ValidationResult item in results)  {  message += item.ErrorMessage + "\n";  }  System.Windows.Forms.MessageBox.Show(message);  }  return valid;  }  }  } |

# 5. EJECUCIÓN DEL PROYECTO

Al ejecutar el proyecto obtendrá la siguiente vista.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Figura 21. Lista de registros

Al dar clic sobre el Quitar con un registro seleccionado este se eliminará.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Figura 22. Retiro de un registro.

Al dar clic sobre uno de los registros y elegir editar se podra ver los datos del registro.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Figura 23. Edición de registro.

Si se modifican los datos de este registro ellos serán guardados y se puede apreciar en la lista.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Figura 24. Edición de registro.

En caso de editar un registro o editarlo con datos inválidos se obtiene el siguiente mensaje.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Figura 25. Mensajes de validación.

# CONCLUSIONES

* La arquitectura en capas es una forma efectiva de organizar un proyecto de software, separando las diferentes responsabilidades en capas bien definidas y reduciendo la complejidad del código.
* Domain-Driven Design (DDD) es una metodología que nos ayuda a construir software de alta calidad al enfocarnos en el dominio del problema y crear un modelo de dominio rico y expresivo.
* Las validaciones son importantes en cualquier aplicación para garantizar la integridad y la consistencia de los datos. Es recomendable implementar las validaciones en la capa de dominio, ya que es donde se encuentra la lógica de negocio del sistema.
* En la capa de dominio se suele tener la lógica de negocio del sistema, mientras que en la capa de presentación se encuentra la interfaz de usuario y la lógica relacionada con la interacción del usuario.

# RECOMENDACIONES

* Es necesario asegurarse de comprender completamente los conceptos teóricos antes de comenzar con el desarrollo del proyecto. Esto te permitirá diseñar una arquitectura sólida y coherente, lo que resultará en un software robusto y escalable.
* Documentar adecuadamente todo el proceso de investigación y desarrollo, incluyendo las decisiones de diseño tomadas, los desafíos enfrentados y las soluciones implementadas. Esto te permitirá comprender mejor el proyecto y facilitará la tarea de futuros desarrolladores que trabajen en él.
* Tener cuidado al implementar las validaciones en el proyecto. Asegúrate de aplicarlas de manera consistente en todas las capas, preferiblemente en la capa de dominio, y de verificar que funcionen correctamente en diferentes escenarios.

# BIBLIOGRAFÍA

|  |  |
| --- | --- |
| [1] | «Arquitectura en Capas,» reactiveprogramming, [En línea]. Available: https://tinyurl.com/2nkcnw28. [Último acceso: 11 03 2023]. |
| [2] | «Arquitectura basada en capas.,» geeks, [En línea]. Available: https://tinyurl.com/ykz6vapf. [Último acceso: 11 03 2023]. |
| [3] | «Best Practice - An Introduction To Domain-Driven Design,» Microsoft, 22 01 2016. [En línea]. Available: https://tinyurl.com/2bcb6u8c. [Último acceso: 11 03 2023]. |
| [4] | «ARQUITECTURA DDD Y SUS CAPAS #MERGE,» rojiblancosite, [En línea]. Available: https://tinyurl.com/4yx2x832. [Último acceso: 11 03 2023]. |
| [5] | «Programación orientada a objetos,» IBM, 17 08 2021. [En línea]. Available: https://tinyurl.com/mc89rjpf. [Último acceso: 11 03 2023]. |
| [6] | «Validaciones,» ProcessPro, [En línea]. Available: https://tinyurl.com/2p9esvhc. [Último acceso: 11 03 2023]. |
| [7] | «Arquitectura en Capas – Análisis completo + Tradicional vs Modernas, DDD, DIP (Cap 5),» RJCode, [En línea]. Available: https://tinyurl.com/mrxbtbyf. [Último acceso: 11 03 2023]. |
| [9] | «Arquitectura en Capas Proyectos,» RJCode, [En línea]. Available: https://rjcodeadvance.com/category/capas/. [Último acceso: 11 03 2023]. |