Imagen que contiene interior, mamífero, gato, viendo

Descripción generada automáticamente

**Tema**

TALLER C7 CRUD CON ARQUITECTURA TRADICIONAL EN CAPAS, VISUAL BASIC Y SQL SERVER

**Tutor**

Ing. Eduardo Mauricio Campaña Ortega

MIS.MDU.CCNA.CCIA.

PhD. (c) Ingeniería de Software

PhD. (c) Seguridad Información

**Fecha**

12/03/2023

**TALLER C7 CRUD CON ARQUITECTURA TRADICIONAL EN CAPAS, VISUAL BASIC Y SQL SERVER**

[INTRODUCCIÓN 5](#_Toc129465200)

[OBJETIVOS 5](#_Toc129465201)

[OBJETIVO GENERAL 5](#_Toc129465202)

[OBJETIVOS ESPECIFICOS 5](#_Toc129465203)

[MARCO TEÓRICO 6](#_Toc129465204)

[VISUAL BASIC 6](#_Toc129465205)

[SQL SERVER 6](#_Toc129465206)

[ARQUITECTURA EN CAPAS 7](#_Toc129465207)

[HISTORIA Y EVOLUCIÓN DE ARQUITECTURA EN CAPAS 7](#_Toc129465208)

[DEFINICIÓN DE ARQUITECTURA EN CAPAS 7](#_Toc129465209)

[PRINCIPIOS DE LA ARQUITECTURA EN CAPAS 8](#_Toc129465210)

[MODELOS DE ARQUITECTURA EN CAPAS 9](#_Toc129465211)

[DISEÑO DE CADA CAPA 11](#_Toc129465212)

[INTERFAZ ENTRE CAPAS 25](#_Toc129465213)

[IMPLEMENTACIÓN DE LA ARQUITECTURA EN CAPAS 26](#_Toc129465214)

[SEGURIDAD EN UNA ARQUITECTURA EN CAPAS 26](#_Toc129465215)

[PATRONES DE DISEÑO RELACIONADOS 27](#_Toc129465216)

[VENTAJAS Y DESVENTAJAS 28](#_Toc129465217)

[EVOLUCIONES 29](#_Toc129465218)

[MODELO ANÉMICO 33](#_Toc129465219)

[PARTE PRÁCTICA 34](#_Toc129465220)

[DESARROLLO DEL PROYECTO 34](#_Toc129465221)

[EJECUCIÓN 49](#_Toc129465222)

[CONCLUSIONES 54](#_Toc129465223)

[RECOMENDACIONES 54](#_Toc129465224)

[BIBLIOGRAFÍA 55](#_Toc129465225)

**INDICE DE IMÁGENES**

[Figura 1. Estructura básica arquitectura en capas 8](#_Toc129461609)

[Figura 2. Arquitectura en capas tradicional 11](#_Toc129461610)

[Figura 3. Arquitectura en capas flexible + DIP 30](#_Toc129461611)

[Figura 4. Arquitectura hexagonal 31](#_Toc129461612)

[Figura 5. Arquitectura Onion 32](#_Toc129461613)

[Figura 6. Arquitectura Clean 33](#_Toc129461614)

[Figura 7. Script para SQL Server 34](#_Toc129461615)

[Figura 8. Inserción de Registros en SQL Server Management Studio 34](#_Toc129461616)

[Figura 9. Selección de la Plantilla para el Proyecto 35](#_Toc129461617)

[Figura 10. Nombre y Ubicación del Proyecto 35](#_Toc129461618)

[Figura 11. Creación de un archivo de vista Windows Forms 36](#_Toc129461619)

[Figura 12. Archivo de Vista Aplicación de Windows Forms 36](#_Toc129461620)

[Figura 13. Definir el nombre para la Aplicación de Windows Forms 37](#_Toc129461621)

[Figura 14. Agregar tipo de proyecto Biblioteca de Clases 37](#_Toc129461622)

[Figura 15. Nombre Domain para la Biblioteca de clases 38](#_Toc129461623)

[Figura 16. Biblioteca de Clases para la Solución 38](#_Toc129461624)

[Figura 17. Nombre DataAccess para la Biblioteca de Clases 39](#_Toc129461625)

[Figura 18. Estructura inicial para el proyecto 39](#_Toc129461626)

[Figura 19. Referencias entre Capas 40](#_Toc129461627)

[Figura 20. Selección de las capas a referenciar 40](#_Toc129461628)

[Figura 21. Referencia de la capa DataAccess 41](#_Toc129461629)

[Figura 22. Carpetas para la estructura de cada capa 41](#_Toc129461630)

[Figura 23. Codificación de la clase Employee 42](#_Toc129461631)

[Figura 24. Interfaz en la carpeta Contracts en DataAccess 42](#_Toc129461632)

[Figura 25. Métodos de la interfaz IGenericRepository 43](#_Toc129461633)

[Figura 26. Codificación de la interfaz IEmployeeRepository 43](#_Toc129461634)

[Figura 27. Conexión a la Base de Datos 43](#_Toc129461635)

[Figura 28. Selección del origen de datos 44](#_Toc129461636)

[Figura 29. Propiedades de conexión a la Base de Datos 44](#_Toc129461637)

[Figura 30. Archivo de configuración App.config 45](#_Toc129461638)

[Figura 31. Referencia a System.Configuration 45](#_Toc129461639)

[Figura 32. Clase Repository en la capa DataAccess 45](#_Toc129461640)

[Figura 33. Clase MasterRepository en la capa DataAccess 46](#_Toc129461641)

[Figura 34. Clase EmployeeRepository en la capa DataAccess 47](#_Toc129461642)

[Figura 35. Enum EntityState en la capa Domain 47](#_Toc129461643)

[Figura 36. Referencia a System.ComponentModel.DataAnnotations 48](#_Toc129461644)

[Figura 37. Formulario en la capa de Presentación 48](#_Toc129461645)

[Figura 38. Ventana con los datos de los empleados 49](#_Toc129461646)

[Figura 39. Datos filtrados en la tabla de FormEmployee 49](#_Toc129461647)

[Figura 40. Validación de las entradas en el formulario 50](#_Toc129461648)

[Figura 41. Registro de un nuevo empleado 50](#_Toc129461649)

[Figura 42. Vista del nuevo registro en la tabla de empleados 51](#_Toc129461650)

[Figura 43. Editar el registro de un empleado 51](#_Toc129461651)

[Figura 44. Editar los datos del empleado seleccionado 52](#_Toc129461652)

[Figura 45. Registro de datos actualizado en la tabla 52](#_Toc129461653)

[Figura 46. Eliminar un registro de empleado 53](#_Toc129461654)

[Figura 47. Notificación de registro eliminado 53](#_Toc129461655)

**ÍNDICE DE TABLAS**

[Tabla 1. Patrones aplicados a la capa de presentación 15](#_Toc129461656)

[Tabla 2. Patrones aplicados a la capa de datos 20](#_Toc129461657)

[Tabla 3. Patrones aplicados a la capa de negocio 24](#_Toc129461658)

[Tabla 4. Ventajas y desventajas de la arquitectura por capas 29](#_Toc129461659)

# INTRODUCCIÓN

Hoy en día, los patrones de arquitectura son ampliamente utilizados en la industria del software y son una parte importante de la educación y la formación en ciencias de la computación y en ingeniería de software.

Los patrones de arquitectura son soluciones probadas y comprobadas para problemas comunes de diseño de software y arquitectura de sistemas. Estos patrones son una forma efectiva de comunicar soluciones de diseño entre arquitectos de software y desarrolladores, ya que permiten que los equipos de desarrollo construyan sistemas con estructuras confiables y escalables.

Los patrones de arquitectura se centran en la organización de los componentes de un sistema, en lugar de en el detalle de su implementación. Por lo tanto, pueden aplicarse a sistemas de cualquier lenguaje o plataforma [1].

Uno de los patrones de software más comunes es el patrón en capas o "Layered Pattern". Este patrón divide el software en capas separadas, donde cada capa es responsable de un conjunto específico de tareas. La capa superior se comunica con la capa inferior a través de una interfaz bien definida, lo que significa que cada capa no necesita saber nada sobre las capas superiores o inferiores.

# OBJETIVOS

## OBJETIVO GENERAL

Investigar sobre los patrones de arquitectura y el patrón MVC en la ingeniera de software para resolver problemas en el desarrollo de sistemas y aplicaciones, para esto realizar una práctica con el lenguaje de programación Visual Basic.

## OBJETIVOS ESPECIFICOS

* Investigar sobre los patrones en la Ingeniería y desarrollo de Software, los tipos y categorías.
* Realizar una práctica para la búsqueda y filtrado de datos en el que se use la arquitectura tradicional en capas.
* Documentar todo el proceso de la parte práctica para usarlo como manual y guía en futuros proyectos.

# MARCO TEÓRICO

## VISUAL BASIC

Visual Basic es un lenguaje de programación de alto nivel que se utiliza principalmente para el desarrollo de aplicaciones de software para Windows. Fue desarrollado originalmente por Microsoft en la década de 1990 y ha sido una herramienta de programación popular desde entonces. Es un lenguaje de programación orientado a objetos, lo que significa que utiliza conceptos como clases y objetos para modelar el comportamiento de las aplicaciones.

Visual Basic se basa en el modelo de programación de eventos, lo que significa que los programas se ejecutan en respuesta a eventos como clics de botón o entradas de teclado. Los programadores de Visual Basic utilizan una interfaz gráfica de usuario (GUI) para diseñar las interfaces de usuario de sus aplicaciones, lo que facilita la creación de aplicaciones fáciles de usar y atractivas visualmente.

Visual Basic también incluye una amplia variedad de bibliotecas y herramientas para facilitar el desarrollo de aplicaciones. Por ejemplo, la biblioteca de controles de Windows proporciona una amplia variedad de componentes gráficos que se pueden utilizar en las interfaces de usuario de las aplicaciones. También hay herramientas para la depuración de programas y para el análisis de rendimiento, lo que facilita la identificación y corrección de errores en el código.

## SQL SERVER

SQL Server es un sistema de gestión de bases de datos relacionales desarrollado por Microsoft. Aquí hay algunos aspectos clave sobre SQL Server:

* Estructura: SQL Server utiliza un modelo de datos relacional, lo que significa que los datos se organizan en tablas, cada una de las cuales tiene una clave principal única. Las relaciones entre las tablas se definen mediante claves foráneas.
* Lenguaje: SQL (Structured Query Language) es el lenguaje utilizado para interactuar con SQL Server. SQL se utiliza para crear, modificar y consultar bases de datos, así como para definir las relaciones entre las tablas.
* Características: SQL Server ofrece una amplia gama de características, incluyendo transacciones ACID, replicación de datos, copias de seguridad y recuperación, y herramientas de gestión y monitorización. SQL Server también incluye un motor de almacenamiento de objetos grandes (LOB), que permite el almacenamiento y recuperación de datos binarios grandes, como imágenes o archivos de audio.
* Ediciones: SQL Server viene en varias ediciones, incluyendo la edición gratuita Express, la edición estándar y la edición empresarial. Cada edición tiene diferentes características y limitaciones de capacidad.
* Integración: SQL Server se integra con otras herramientas de Microsoft, como Visual Studio y Azure, lo que permite la creación y gestión de aplicaciones empresariales basadas en la nube. También se puede utilizar en conjunto con otras tecnologías de Microsoft, como SharePoint y Dynamics.
* Seguridad: SQL Server ofrece una amplia gama de características de seguridad, incluyendo autenticación de usuarios, control de acceso basado en roles y encriptación de datos. También tiene una característica de auditoría que permite el seguimiento de las actividades de los usuarios.

En general, SQL Server es un sistema de gestión de bases de datos fiable y escalable que se utiliza en una amplia variedad de aplicaciones empresariales. Si estás interesado en aprender sobre la gestión de bases de datos relacionales y cómo utilizar SQL para interactuar con ellas, SQL Server es una excelente opción para comenzar.

## ARQUITECTURA EN CAPAS

### HISTORIA Y EVOLUCIÓN DE ARQUITECTURA EN CAPAS

La arquitectura de capas es un enfoque de diseño de software que se utiliza para crear sistemas complejos y estructurados. Esta arquitectura se basa en la idea de que un sistema puede descomponerse en capas, donde cada capa es responsable de un conjunto específico de tareas y se comunica con las capas adyacentes a través de interfaces bien definidas.

La arquitectura de capas ha evolucionado con el tiempo y ha sido aplicada en diferentes contextos. Una de las primeras aplicaciones de esta arquitectura se dio en el diseño de sistemas operativos en los años 60 y 70. En estos sistemas, la arquitectura de capas se utilizaba para separar el kernel del sistema operativo de las aplicaciones de usuario y de los controladores de dispositivos. De esta forma, se lograba una mayor estabilidad y seguridad del sistema, ya que los fallos en las aplicaciones de usuario o los controladores de dispositivos no afectaban núcleo del sistema operativo [2].

Posteriormente, la arquitectura de capas se utilizó en el diseño de sistemas de bases de datos. En este caso, la capa más baja de la arquitectura correspondía al motor de almacenamiento de la base de datos, mientras que las capas superiores se encargaban de la gestión de consultas y la presentación de resultados.

Con el desarrollo de los sistemas distribuidos, la arquitectura de capas ha sido aplicada en el diseño de sistemas cliente-servidor y en la web. En estos casos, la capa de presentación corresponde al cliente, mientras que la capa de aplicación y la capa de datos se encuentran en el servidor. Esta arquitectura ha permitido la separación de las tareas de presentación y lógica de negocio, lo que facilita la evolución y el mantenimiento del sistema.

### DEFINICIÓN DE ARQUITECTURA EN CAPAS

La arquitectura en capas es un modelo de diseño de software en el que las diferentes funcionalidades de una aplicación se organizan en capas separadas y se comunican entre sí a través de interfaces definidas. Cada capa se centra en una función específica y puede ser desarrollada, probada y mantenido de manera independiente [3].

La arquitectura en capas generalmente se divide en tres capas principales:

* Capa de presentación: esta capa es responsable de la interfaz de usuario y la interacción con el usuario final. Aquí se encuentran las páginas web, aplicaciones móviles o cualquier otra interfaz de usuario que interactúa con la aplicación.
* Capa de lógica de negocio: esta capa contiene la lógica de negocio de la aplicación. Aquí se encuentra el código que se encarga de realizar las operaciones y cálculos necesarios para que la aplicación funcione.
* Capa de acceso a datos: esta capa se encarga de acceder a los datos necesarios para que la aplicación funcione. Aquí se encuentran los servidores de bases de datos y otros sistemas de almacenamiento de datos.

Cada capa se comunica con la capa inmediatamente superior y la capa inmediatamente inferior mediante interfaces definidas. Esto significa que la capa de presentación solo puede acceder a la capa de lógica de negocio a través de las interfaces definidas, y la capa de lógica de negocio solo puede acceder a la capa de acceso a datos a través de las interfaces definidas.

Este modelo de arquitectura permite una mayor modularidad y flexibilidad en el diseño de aplicaciones, lo que facilita la creación de aplicaciones escalables y fáciles de mantener. Además, la separación de responsabilidades entre capas permite una mayor capacidad de prueba y depuración de la aplicación en su conjunto.

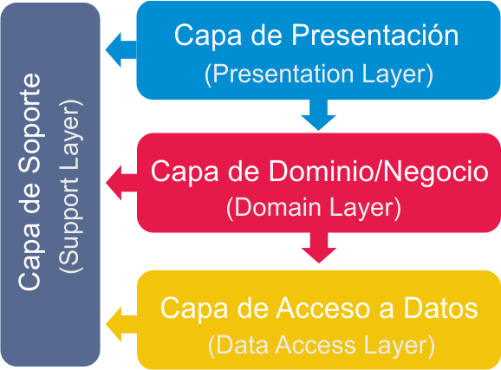


Figura 1. Estructura básica arquitectura en capas

### PRINCIPIOS DE LA ARQUITECTURA EN CAPAS

Los principios fundamentales que rigen la arquitectura en capas de software son:

* Separación de responsabilidades: cada capa tiene una responsabilidad específica y no se entromete en las funciones de otras capas. Esto permite una mayor modularidad y reutilización de código.
* Independencia de tecnología: cada capa debe poder utilizar diferentes tecnologías sin afectar el resto del sistema. Esto permite una mayor flexibilidad en la elección de tecnologías y una mayor adaptabilidad a cambios tecnológicos.
* Comunicación por interfaces: la comunicación entre capas se realiza a través de interfaces bien definidas, lo que permite una mayor interoperabilidad y facilita la integración de nuevas capas.
* Jerarquía de dependencias: cada capa depende únicamente de la capa inmediatamente inferior, lo que reduce la complejidad y facilita el mantenimiento del sistema.
* Centralización del control: la lógica de control se encuentra en las capas superiores del sistema, lo que permite una mayor coherencia y consistencia en el comportamiento del sistema.

En el diseño de sistemas en capas, estos principios se aplican mediante la definición de cada capa y sus funciones específicas, la definición de las interfaces entre capas y la definición de las dependencias entre capas. Es importante asegurarse de que cada capa tenga una responsabilidad claramente definida y que la comunicación entre capas se realice a través de interfaces bien definidas. Además, es importante asegurarse de que la jerarquía de dependencias se respete y que la lógica de control se encuentre en las capas superiores del sistema. Siguiendo estos principios, se puede crear un sistema en capas que sea modular, escalable y fácilmente mantenible [4].

### MODELOS DE ARQUITECTURA EN CAPAS

Existen varios modelos para representar la arquitectura en capas de software, entre los que destacan los siguientes:

* Modelo en capas: este modelo representa la arquitectura en capas como una serie de capas apiladas verticalmente, donde cada capa tiene una responsabilidad específica y se comunica con la capa inmediatamente superior e inferior a través de interfaces bien definidas.
* Modelo en tuberías (pipes and filters): este modelo representa la arquitectura en capas como una serie de filtros que procesan datos que fluyen a través de una tubería. Cada filtro tiene una responsabilidad específica y se comunica con los filtros adyacentes a través de interfaces bien definidas.
* Modelo en niveles (tiered architecture): este modelo representa la arquitectura en capas como una serie de niveles horizontales, donde cada nivel tiene una responsabilidad específica y se comunica con los niveles adyacentes a través de interfaces bien definidas.
* Modelo cliente-servidor: este modelo representa la arquitectura en capas como una separación clara entre el cliente y el servidor, donde el cliente realiza solicitudes al servidor y el servidor procesa las solicitudes y envía respuestas al cliente.

Para documentar y comunicar la arquitectura en capas, se pueden utilizar diferentes técnicas y herramientas, entre las que destacan las siguientes:

* Diagramas de capas: se utilizan para representar la arquitectura en capas como un conjunto de capas apiladas verticalmente, donde cada capa tiene una responsabilidad específica y se comunica con las capas adyacentes a través de interfaces bien definidas.
* Diagramas de tuberías: se utilizan para representar la arquitectura en capas como una serie de filtros que procesan datos que fluyen a través de una tubería. Cada filtro tiene una responsabilidad específica y se comunica con los filtros adyacentes a través de interfaces bien definidas.
* Diagramas de niveles: se utilizan para representar la arquitectura en capas como una serie de niveles horizontales, donde cada nivel tiene una responsabilidad específica y se comunica con los niveles adyacentes a través de interfaces bien definidas.
* Diagramas de cliente-servidor: se utilizan para representar la arquitectura en capas como una separación clara entre el cliente y el servidor, donde el cliente realiza solicitudes al servidor y el servidor procesa las solicitudes y envía respuestas al cliente.

Además de estos diagramas, se pueden utilizar documentos de arquitectura, especificaciones de interfaces, modelos de datos y otros artefactos para documentar y comunicar la arquitectura en capas de software. Es importante utilizar herramientas y técnicas que permitan una comunicación clara y efectiva de la arquitectura a todas las partes interesadas en el proyecto [5].

#### ESTRATIFICACIÓN ESTRICTA – CAPAS TRADICIONAL

La estratificación estricta o capas tradicional es una de las formas más simples de arquitectura de capas. En este modelo, se divide el sistema en tres capas principales: la capa de presentación, la capa de aplicación y la capa de datos.

La capa de presentación es la capa más externa y se encarga de la interacción con el usuario final. Esta capa se enfoca en la presentación de la información y en la interacción con el usuario. En esta capa, se definen elementos como la interfaz de usuario, la lógica de navegación y la presentación de datos [5].

La capa de aplicación es la capa intermedia, que se encarga de la lógica de negocio. En esta capa se procesan las solicitudes del usuario y se realizan las operaciones necesarias para satisfacerlas. Aquí se definen elementos como la lógica de aplicación, el acceso a servicios externos y la coordinación de las operaciones.

La capa de datos es la capa más interna, que se encarga del almacenamiento y acceso a los datos. En esta capa se definen elementos como la base de datos, el modelo de datos y el acceso a los datos.

Este modelo tiene una ventaja clara en la claridad de responsabilidades de cada capa, lo que facilita el mantenimiento y evolución de un sistema. Sin embargo, también tiene una limitación en la flexibilidad para adaptarse a nuevos requerimientos o cambios en el negocio, ya que puede requerir modificaciones en todas las capas.

Es importante tener en cuenta que este modelo es solo una de las muchas posibles formas de arquitectura de capas y que se pueden definir diferentes variantes y subcapas dentro de cada capa. La elección del modelo adecuado dependerá del contexto y los requerimientos del sistema en cuestión.

Diagrama, Texto

Descripción generada automáticamente

Figura 2. Arquitectura en capas tradicional

### DISEÑO DE CADA CAPA

El diseño de cada capa en una arquitectura en capas de software depende de la funcionalidad específica que se desea implementar. Sin embargo, en términos generales, se pueden identificar algunas funciones y responsabilidades comunes para cada capa. A continuación, se describen las principales capas y sus funciones y responsabilidades:

#### CAPA DE PRESENTACIÓN

La capa de presentación es una de las capas de la arquitectura en capas de software. Esta capa es la que se encarga de interactuar con los usuarios del sistema y de proporcionar una interfaz gráfica o de línea de comandos para acceder a las funcionalidades del sistema.

La capa de presentación también es responsable de validar y verificar los datos de entrada proporcionados por el usuario, y de enviarlos a la capa de negocio o de servicio para su procesamiento. Una vez que la capa de negocio o de servicio procesa los datos, la capa de presentación se encarga de mostrar los resultados al usuario [7].

Existen varios tipos de capas de presentación, incluyendo la capa de presentación web, la capa de presentación de escritorio y la capa de presentación móvil. La elección de la capa de presentación dependerá del tipo de sistema que se esté desarrollando y del público objetivo.

Algunas de las tecnologías y herramientas comunes utilizadas en la capa de presentación incluyen HTML, CSS, JavaScript, frameworks como React, Angular o Vue.js para aplicaciones web, y librerías como Swing o JavaFX para aplicaciones de escritorio. Para aplicaciones móviles, se utilizan herramientas como Android Studio para Android y Xcode para iOS.

La capa de presentación es importante porque es la interfaz que los usuarios ven y utilizan para interactuar con el sistema. Por lo tanto, es esencial que la capa de presentación sea fácil de usar, accesible y estéticamente agradable para los usuarios. También es importante que se implementen medidas de seguridad para proteger la información confidencial del usuario.

En esta capa puedes aplicar patrones (Opcional) como:

|  |  |
| --- | --- |
| PATRONES | DESCRIPCIÓN |
| Modelo vista controlador (MVC) | El patrón Modelo-Vista-Controlador (MVC) es un patrón de diseño de software que se utiliza para desarrollar aplicaciones de software escalables y fáciles de mantener. Se divide en tres componentes principales:   * Modelo: El modelo representa los datos y las reglas de negocio de la aplicación. Se encarga de gestionar los datos, validarlos y realizar las operaciones necesarias sobre ellos. * Vista: La vista es la interfaz de usuario que muestra los datos al usuario y le permite interactuar con ellos. Se encarga de recibir las acciones del usuario y enviarlas al controlador. * Controlador: El controlador se encarga de manejar las interacciones entre la vista y el modelo. Recibe las acciones del usuario de la vista, procesa la lógica de la aplicación en el modelo y actualiza la vista con los cambios. |
| Modelo jerárquico vista controlador (HMVC – PAC) | El Modelo Jerárquico Vista Controlador (HMVC) es una extensión del patrón Modelo Vista Controlador (MVC) que se utiliza para desarrollar aplicaciones web más grandes y complejas. El HMVC se basa en la idea de que los componentes de una aplicación web pueden organizarse en una jerarquía de módulos MVC anidados.  El HMVC se compone de los siguientes componentes:   * Controlador Principal: Este controlador es responsable de manejar las solicitudes de los usuarios y coordinar la comunicación entre los módulos. * Módulos: Los módulos son subaplicaciones MVC que se encargan de manejar una porción específica de la aplicación. Cada módulo tiene su propio controlador, modelo y vista. * Controladores de Módulo: Cada módulo tiene su propio controlador que se encarga de manejar las solicitudes específicas del módulo. * Modelos de Módulo: Cada módulo tiene su propio modelo que se encarga de manejar la lógica de negocio específica del módulo. * Vistas de Módulo: Cada módulo tiene su propia vista que se encarga de mostrar los datos específicos del módulo al usuario. |
| Modelo vista presentador (MVP) | El patrón Modelo Vista Presentador (MVP) es un patrón de diseño de software que se utiliza para desarrollar aplicaciones que presentan datos a los usuarios y les permiten interactuar con ellos. El MVP es una evolución del patrón Modelo Vista Controlador (MVC), pero se centra más en la separación de responsabilidades entre la presentación y la lógica de negocio.  El MVP se compone de los siguientes componentes:   * Modelo: El modelo representa los datos y la lógica de negocio de la aplicación. Se encarga de gestionar los datos, validarlos y realizar las operaciones necesarias sobre ellos. * Vista: La vista es la interfaz de usuario que muestra los datos al usuario y le permite interactuar con ellos. La vista es pasiva y no realiza ninguna lógica de negocio por sí misma. * Presentador: El presentador se encarga de manejar las interacciones entre la vista y el modelo. Recibe las acciones del usuario de la vista, procesa la lógica de la aplicación en el modelo y actualiza la vista con los cambios. |
| Modelo vista – vista modelo (MVVM) | El patrón Modelo Vista Vista-Modelo (MVVM) es un patrón de diseño de software que se utiliza en el desarrollo de aplicaciones de interfaz de usuario. Este patrón se basa en la separación de responsabilidades entre la lógica de presentación, la lógica de negocio y los datos de la aplicación.  El MVVM se compone de los siguientes componentes:   * Vista: La vista es la interfaz de usuario de la aplicación. Es responsable de mostrar los datos y permitir la interacción del usuario. En el caso del MVVM, la vista se compone de elementos de la interfaz de usuario y enlaza a la lógica de presentación. * Modelo de Vista: El modelo de vista es el intermediario entre la vista y el modelo. El modelo de vista es responsable de realizar la lógica de presentación y preparar los datos para que sean presentados por la vista. El modelo de vista se actualiza automáticamente cuando los datos del modelo cambian, y se encarga de actualizar la vista en consecuencia. * Modelo: El modelo es el núcleo de la lógica de negocio de la aplicación. Es responsable de manejar los datos y realizar las operaciones de negocio necesarias. El modelo se actualiza automáticamente cuando los datos cambian. |
| Modelo vista presentador vista modelo | El Modelo Vista Presentador Vista Modelo (MVVM) es un patrón de diseño de software que se utiliza en el desarrollo de aplicaciones de interfaz de usuario. Es una variación del patrón Modelo Vista Presentador (MVP) que utiliza una arquitectura más moderna y orientada a objetos.  El MVVM se compone de los siguientes componentes:   * Vista: La vista es la interfaz de usuario de la aplicación. Es responsable de mostrar los datos y permitir la interacción del usuario. * Presentador: El presentador se encarga de manejar las interacciones entre la vista y el modelo. Recibe las acciones del usuario de la vista, procesa la lógica de la aplicación en el modelo y actualiza la vista con los cambios. * Modelo: El modelo representa los datos y la lógica de negocio de la aplicación. Se encarga de gestionar los datos, validarlos y realizar las operaciones necesarias sobre ellos. * Vista Modelo: El vista modelo es el intermediario entre la vista y el modelo. El vista modelo se encarga de preparar los datos para que sean presentados por la vista y actualizar el modelo cuando los datos cambian. |
| Page controller | Page Controller es un patrón de diseño de software que se utiliza en el desarrollo de aplicaciones web. En este patrón, una sola clase, conocida como el controlador de página, maneja todas las solicitudes de una página web. La idea es que el controlador de página actúe como un intermediario entre el cliente y los componentes de la aplicación, y que sea responsable de coordinar y gestionar las solicitudes de la página web.  El controlador de página se encarga de manejar todas las solicitudes HTTP que se realizan a la página web. Cuando se recibe una solicitud, el controlador de página determina la acción que debe realizarse y llama al componente correspondiente para procesar la solicitud. Por ejemplo, si un usuario envía un formulario, el controlador de página recibirá los datos del formulario y llamará al componente correspondiente para procesarlos.  El controlador de página también puede manejar la creación y destrucción de los componentes de la aplicación. Esto significa que puede crear un componente cuando es necesario y destruirlo cuando ya no se necesita. De esta manera, el controlador de página puede manejar eficientemente los recursos de la aplicación y reducir la carga en el servidor.  El patrón de diseño Page Controller se utiliza comúnmente en aplicaciones web que utilizan el modelo de arquitectura basado en servlets y JSP (JavaServer Pages). Este modelo separa la lógica de presentación de la lógica de negocio, lo que facilita el mantenimiento y la actualización de la aplicación. Además, el uso de un controlador de página puede facilitar la implementación de la seguridad en la aplicación, ya que el controlador de página puede ser responsable de validar las solicitudes y asegurarse de que solo los usuarios autorizados puedan acceder a ciertas partes de la aplicación. |
| Front controller | Front Controller es un patrón de diseño de software que se utiliza en el desarrollo de aplicaciones web para centralizar la gestión de las solicitudes del cliente. En este patrón, un controlador central, conocido como el Front Controller, maneja todas las solicitudes de la aplicación web y decide qué acción tomar en función de la solicitud recibida.  El Front Controller actúa como una puerta de entrada a la aplicación web, recibiendo todas las solicitudes del cliente y procesándolas antes de enviarlas al controlador adecuado. El Front Controller puede realizar tareas como autenticación, autorización, validación y enrutamiento de solicitudes antes de enviarlas al controlador correspondiente. También puede manejar errores y enviar respuestas al cliente.  El Front Controller se encarga de manejar todo el flujo de la aplicación, lo que significa que tiene acceso a todos los componentes y recursos de la aplicación. Puede coordinar la ejecución de múltiples controladores y asegurarse de que se ejecuten en el orden correcto. Además, puede manejar la respuesta del cliente y enviarla de vuelta al navegador del usuario.  El patrón de diseño Front Controller se utiliza comúnmente en aplicaciones web de gran escala que tienen múltiples controladores y una variedad de recursos y componentes. Al utilizar un Front Controller, se puede centralizar la gestión de solicitudes y simplificar el proceso de desarrollo y mantenimiento de la aplicación. También se puede mejorar la seguridad de la aplicación, ya que el Front Controller puede validar y autenticar las solicitudes antes de enviarlas a los controladores correspondientes. |
| Application controller | Application Controller es un patrón de diseño de software que se utiliza en el desarrollo de aplicaciones web para centralizar la gestión de las solicitudes del cliente y coordinar la ejecución de diferentes componentes y servicios de la aplicación. En este patrón, un controlador central, conocido como el Application Controller, maneja todas las solicitudes de la aplicación web y decide qué acción tomar en función de la solicitud recibida.  El Application Controller es responsable de manejar todas las solicitudes de la aplicación y determinar qué acción tomar en función de la solicitud recibida. Puede realizar tareas como la autenticación y la autorización de usuarios, la validación de datos, el enrutamiento de solicitudes y la ejecución de servicios y componentes de la aplicación.  El Application Controller utiliza un conjunto de reglas para determinar qué acción tomar en función de la solicitud recibida. Estas reglas pueden ser configuradas en un archivo de configuración o en una base de datos y pueden ser actualizadas sin tener que cambiar el código fuente de la aplicación. Esto hace que el sistema sea más flexible y escalable.  El Application Controller también puede actuar como un intermediario entre el cliente y los servicios y componentes de la aplicación. Puede coordinar la ejecución de múltiples servicios y componentes y asegurarse de que se ejecuten en el orden correcto. Además, puede manejar la respuesta del cliente y enviarla de vuelta al navegador del usuario.  El patrón de diseño Application Controller se utiliza comúnmente en aplicaciones web de gran escala que tienen múltiples componentes y servicios y una variedad de solicitudes de usuario. Al utilizar un Application Controller, se puede centralizar la gestión de solicitudes y simplificar el proceso de desarrollo y mantenimiento de la aplicación. También se puede mejorar la seguridad de la aplicación, ya que el Application Controller puede validar y autenticar las solicitudes antes de enviarlas a los servicios y componentes correspondientes. |

Tabla 1. Patrones aplicados a la capa de presentación

#### CAPA DE APLICACIÓN

La capa de aplicación es una de las capas de la arquitectura en capas de software. Esta capa se encuentra en el medio de la arquitectura en capas y se encarga de coordinar y gestionar las funcionalidades del sistema.

La capa de aplicación es la encargada de procesar la lógica de negocio y de comunicarse con otras capas, como la capa de presentación y la capa de acceso a datos. Además, la capa de aplicación es responsable de gestionar los servicios de la aplicación y de proporcionar una interfaz para su consumo por parte de los clientes [6].

En general, la capa de aplicación se compone de varios componentes o módulos, cada uno de los cuales se encarga de una tarea específica en la aplicación. Algunas de las funcionalidades que pueden incluirse en la capa de aplicación son la gestión de usuarios, la validación de datos, la gestión de transacciones, la gestión de errores y el control de acceso.

Algunas de las tecnologías y herramientas comunes utilizadas en la capa de aplicación incluyen lenguajes de programación como Java, C#, Python o Ruby, y frameworks como Spring, .NET, Flask o Ruby on Rails. Además, la capa de aplicación puede utilizar herramientas de integración, como ESBs (Enterprise Service Bus) o plataformas de orquestación, para comunicarse con otras aplicaciones.

La capa de aplicación es importante porque es la encargada de gestionar y coordinar las funcionalidades del sistema, así como de garantizar su coherencia y estabilidad. Además, la capa de aplicación puede ser la encargada de aplicar políticas de seguridad y de realizar la integración con otros sistemas o servicios.

#### CAPA DE DATOS

La capa de datos es una de las capas de la arquitectura en capas de software. Esta capa se encarga de gestionar los datos de la aplicación y proporcionar acceso a ellos a través de la capa de aplicación.

La capa de datos se divide en dos partes principales: la capa de acceso a datos y la capa de almacenamiento de datos. La capa de acceso a datos se encarga de proporcionar una interfaz para acceder y manipular los datos, mientras que la capa de almacenamiento de datos se encarga de gestionar la persistencia de los datos en el sistema de almacenamiento.

La capa de acceso a datos puede estar compuesta por diferentes componentes o módulos, como los controladores JDBC (Java Database Connectivity), los ORM (Object-Relational Mapping) o los servicios web de datos. Estos componentes pueden proporcionar diferentes niveles de abstracción para el acceso a los datos, lo que permite a la capa de aplicación interactuar con la capa de datos de una forma más sencilla y eficiente.

La capa de almacenamiento de datos puede incluir diferentes tecnologías de bases de datos, como MySQL, PostgreSQL, MongoDB o Cassandra. Además, también puede incluir tecnologías de almacenamiento en la nube, como Amazon S3 o Google Cloud Storage [7].

Es importante mencionar que la capa de datos es una capa crítica para la integridad y seguridad de la información del sistema. Por ello, se deben tener en cuenta las mejores prácticas de diseño y gestión de bases de datos, como la normalización de datos, la implementación de claves primarias y foráneas, el uso de transacciones y el control de acceso a los datos.

Esta capa también es conocida como capa de persistencia o capa de integración, en ella se utilizar patrones (Opcional) como:

|  |  |
| --- | --- |
| PATRONES | DESCRIPCIÓN |
| Enlace a datos de tabla | El patrón Enlace a datos de tabla (Table Data Gateway) es un patrón de diseño de software que se utiliza para encapsular el acceso a una tabla de base de datos en una clase separada. Esta clase, conocida como Gateway, proporciona una interfaz para realizar operaciones de CRUD (crear, leer, actualizar y eliminar) en la tabla de base de datos.  El patrón Enlace a datos de tabla se utiliza para separar la lógica de acceso a datos de la lógica de negocio de una aplicación. Al encapsular el acceso a la tabla de base de datos en una clase separada, se pueden realizar cambios en la estructura de la tabla de base de datos sin afectar a la lógica de negocio de la aplicación.  El Gateway proporciona una interfaz para realizar operaciones CRUD en la tabla de base de datos. Esta interfaz puede incluir métodos para recuperar registros individuales, recuperar todos los registros de la tabla, agregar nuevos registros, actualizar registros existentes y eliminar registros existentes. El Gateway también puede proporcionar métodos para realizar operaciones de filtrado y ordenación en los datos de la tabla.  El patrón Enlace a datos de tabla se utiliza comúnmente en aplicaciones web y de bases de datos, donde se necesita una forma eficiente de acceder y manipular los datos de la tabla. Al utilizar este patrón, se puede separar la lógica de acceso a datos de la lógica de negocio de la aplicación y mejorar la escalabilidad y mantenibilidad de la aplicación. |
| Registro activo | El patrón Registro Activo (Active Record) es un patrón de diseño de software que se utiliza para representar un objeto de base de datos como un objeto en un lenguaje de programación orientado a objetos. En este patrón, cada registro en una tabla de base de datos se representa como un objeto y el objeto es responsable de gestionar su propio ciclo de vida, incluyendo su creación, recuperación, actualización y eliminación.  En el patrón Registro Activo, cada objeto es responsable de su propia persistencia en la base de datos. El objeto contiene la lógica para crear, leer, actualizar y eliminar registros en la tabla de base de datos correspondiente. Además, el objeto también puede contener la lógica de validación de datos y la lógica de negocio.  El patrón Registro Activo se utiliza comúnmente en aplicaciones web y de bases de datos, donde se necesita una forma eficiente de acceder y manipular los datos de la tabla. Al utilizar este patrón, se pueden reducir la cantidad de código repetitivo necesario para interactuar con la base de datos y mejorar la mantenibilidad y escalabilidad de la aplicación. |
| Mapeado de datos | El patrón Mapeado de Datos (Data Mapping) es un patrón de diseño de software que se utiliza para mapear los datos de una fuente de datos a una estructura de objetos en un lenguaje de programación orientado a objetos. Este patrón se utiliza comúnmente en aplicaciones que necesitan interactuar con bases de datos o servicios web.  El patrón Mapeado de Datos se utiliza para abstraer la lógica de acceso a datos de la lógica de negocio de la aplicación. La idea es que cada objeto de la aplicación se corresponda con un registro o entidad en la base de datos, y que estos objetos puedan ser creados, leídos, actualizados y eliminados utilizando métodos de una clase de mapeo de datos.  La clase de mapeo de datos es responsable de realizar las operaciones necesarias para interactuar con la fuente de datos. Esto puede incluir la creación de conexiones a la base de datos, la realización de consultas y la transformación de los resultados de las consultas en objetos de la aplicación. La clase de mapeo de datos también puede incluir lógica de validación de datos y lógica de negocio para garantizar la integridad y coherencia de los datos.  El patrón Mapeado de Datos se utiliza comúnmente en aplicaciones web y de bases de datos, donde se necesita una forma eficiente de acceder y manipular los datos de la tabla. Al utilizar este patrón, se pueden separar la lógica de acceso a datos de la lógica de negocio de la aplicación y mejorar la mantenibilidad y escalabilidad de la aplicación.  Una de las principales ventajas del patrón Mapeado de Datos es que puede simplificar la tarea de interactuar con una fuente de datos, como una base de datos o un servicio web. Sin embargo, una limitación del patrón es que puede resultar complicado de mantener para aplicaciones con modelos de datos complejos o que necesiten interactuar con múltiples fuentes de datos. En estos casos, se pueden utilizar otros patrones, como el patrón Modelo-Vista-Controlador (MVC) o el patrón Enlace a Datos de Tabla, para gestionar la lógica de acceso a datos de la aplicación. |
| Objeto de consulta | El patrón Objeto de Consulta (Query Object) es un patrón de diseño de software que se utiliza para encapsular una consulta a una fuente de datos, como una base de datos o un servicio web, en un objeto de consulta reutilizable y fácilmente configurable. Este patrón se utiliza comúnmente en aplicaciones web y de bases de datos, donde se necesita una forma eficiente de consultar datos de una fuente de datos.  En el patrón Objeto de Consulta, se crea un objeto de consulta que encapsula una consulta a la fuente de datos. El objeto de consulta contiene toda la información necesaria para realizar la consulta, como los criterios de búsqueda, los campos a recuperar y los parámetros de ordenación y paginación. Además, el objeto de consulta puede contener lógica para validar los criterios de búsqueda y procesar los resultados de la consulta.  El patrón Objeto de Consulta permite encapsular la lógica de consulta en un objeto reutilizable y fácilmente configurable. Esto significa que se pueden crear objetos de consulta personalizados para cada necesidad de la aplicación sin tener que escribir código repetitivo. Además, al utilizar objetos de consulta, se pueden mantener separadas la lógica de consulta y la lógica de negocio de la aplicación, lo que puede mejorar la mantenibilidad y escalabilidad de la aplicación. |
| Objeto de acceso a datos | El patrón Objeto de Acceso a Datos (Data Access Object, DAO) es un patrón de diseño de software que se utiliza para abstraer la lógica de acceso a datos de la lógica de negocio de la aplicación. Este patrón se utiliza comúnmente en aplicaciones que necesitan interactuar con bases de datos o servicios web.  En el patrón Objeto de Acceso a Datos, se crea una clase DAO que encapsula la lógica de acceso a datos. La clase DAO proporciona métodos para crear, leer, actualizar y eliminar (CRUD) datos en la fuente de datos. Además, la clase DAO puede proporcionar métodos para realizar consultas personalizadas y realizar operaciones de transacción.  La idea detrás del patrón DAO es que cada entidad o tabla de la base de datos tenga su propia clase DAO. Esto significa que cada entidad de la aplicación se corresponda con una entidad o tabla en la base de datos, y que estas entidades puedan ser creadas, leídas, actualizadas y eliminadas utilizando métodos de su correspondiente clase DAO.  El patrón DAO permite separar la lógica de acceso a datos de la lógica de negocio de la aplicación. Esto significa que la lógica de negocio de la aplicación no necesita preocuparse por cómo se accede a los datos. Además, al utilizar el patrón DAO, se puede mejorar la mantenibilidad y escalabilidad de la aplicación, ya que la lógica de acceso a datos está centralizada en una clase DAO.  Una de las principales ventajas del patrón DAO es que puede simplificar la tarea de interactuar con una fuente de datos, como una base de datos o un servicio web. Sin embargo, una limitación del patrón es que puede resultar complicado de mantener para aplicaciones con modelos de datos complejos o que necesiten interactuar con múltiples fuentes de datos. En estos casos, se pueden utilizar otros patrones, como el patrón Mapeado de Datos o el patrón Modelo-Vista-Controlador (MVC), para gestionar la lógica de acceso a datos de la aplicación. |
| Repositorio | El patrón Repositorio (Repository) es un patrón de diseño de software que se utiliza para abstraer la lógica de acceso a datos de la lógica de negocio de la aplicación. Este patrón se utiliza comúnmente en aplicaciones que necesitan interactuar con bases de datos o servicios web.  En el patrón Repositorio, se crea una clase Repositorio que encapsula la lógica de acceso a datos. La clase Repositorio proporciona métodos para crear, leer, actualizar y eliminar (CRUD) datos en la fuente de datos. Además, la clase Repositorio puede proporcionar métodos para realizar consultas personalizadas y realizar operaciones de transacción.  La idea detrás del patrón Repositorio es que cada entidad de la aplicación tenga su propio repositorio. Esto significa que cada entidad de la aplicación se corresponda con una entidad o tabla en la base de datos, y que estas entidades puedan ser creadas, leídas, actualizadas y eliminadas utilizando métodos de su correspondiente repositorio.  El patrón Repositorio permite separar la lógica de acceso a datos de la lógica de negocio de la aplicación. Esto significa que la lógica de negocio de la aplicación no necesita preocuparse por cómo se accede a los datos. Además, al utilizar el patrón Repositorio, se puede mejorar la mantenibilidad y escalabilidad de la aplicación, ya que la lógica de acceso a datos está centralizada en una clase Repositorio. |

Tabla 2. Patrones aplicados a la capa de datos

#### CAPA DE NEGOCIO O DOMINIO

La capa de negocio o dominio es una de las capas de la arquitectura en capas de software. Esta capa se encarga de implementar la lógica de negocio de la aplicación y es responsable de la manipulación y gestión de los datos de la aplicación.

La capa de negocio es el núcleo de la aplicación y se encarga de procesar los datos recibidos de la capa de presentación o de la capa de servicios y proporcionar una respuesta adecuada a la capa que los solicitó. Además, la capa de negocio también es responsable de mantener la consistencia y la integridad de los datos de la aplicación y de aplicar reglas de negocio específicas.

La capa de negocio se divide en diferentes módulos o componentes que se encargan de implementar la lógica de negocio específica de cada proceso o función de la aplicación. Estos componentes pueden incluir controladores de flujo, componentes de validación, componentes de procesamiento y componentes de almacenamiento.

Es importante que la capa de negocio sea independiente de la capa de presentación y de la capa de datos, lo que significa que no debe estar acoplada directamente con ellas. Esto se logra a través del uso de interfaces y protocolos estandarizados para la comunicación entre las capas [8].

Además, la capa de negocio debe ser flexible y adaptable a los cambios en la lógica de negocio y los requisitos de la aplicación. Para lograr esto, es común utilizar patrones de diseño como el patrón de diseño de fábrica y el patrón de diseño de inyección de dependencias.

En esta capa se puede usar patrones (Opcional) como:

|  |  |
| --- | --- |
| PATRONES | DESCRIPCIÓN |
| Entidad de dominio, modelo de dominio u objeto de negocio | El patrón Entidad de Dominio, Modelo de Dominio u Objeto de Negocio es un patrón de diseño de software que se utiliza para representar los conceptos y objetos del dominio de una aplicación. Este patrón se utiliza comúnmente en aplicaciones que tienen una lógica de negocio compleja.  El patrón Entidad de Dominio se basa en la idea de que una aplicación puede ser vista como una serie de objetos que representan conceptos del mundo real. Estos objetos se llaman Entidades de Dominio o Modelos de Dominio, y tienen una identidad única y un conjunto de propiedades y métodos que los definen. Las Entidades de Dominio representan la lógica de negocio de la aplicación y encapsulan la lógica y las reglas que rigen el comportamiento del sistema.  Las Entidades de Dominio suelen ser objetos persistentes que se almacenan en una base de datos o en otro tipo de almacenamiento de datos. En la mayoría de los casos, las Entidades de Dominio se corresponden con las tablas de una base de datos relacional y se utilizan para representar la información almacenada en esas tablas.  El patrón Entidad de Dominio permite separar la lógica de negocio de la lógica de acceso a datos y proporciona una manera de representar la información y la lógica de la aplicación de manera clara y concisa. Además, el patrón Entidad de Dominio puede mejorar la escalabilidad y mantenibilidad de la aplicación al proporcionar una abstracción de alto nivel de la lógica de negocio. |
| Objeto de valor | El patrón Objeto de Valor es un patrón de diseño de software que se utiliza para representar valores inmutables que se utilizan en una aplicación. Los objetos de valor se utilizan para encapsular valores simples y complejos y para proporcionar una manera de comparar y manipular esos valores de manera eficiente.  Un Objeto de Valor es un objeto que representa un valor en sí mismo, en lugar de una entidad o modelo de dominio en la aplicación. Los objetos de valor pueden ser objetos simples, como una fecha o un número de teléfono, o pueden ser objetos más complejos, como una dirección o un conjunto de coordenadas.  Una de las principales ventajas del patrón Objeto de Valor es que puede mejorar la legibilidad y la claridad del código al proporcionar una manera clara y concisa de representar valores complejos. Además, los objetos de valor son inmutables, lo que significa que una vez que se crean, no se pueden modificar. Esto hace que los objetos de valor sean más seguros y menos propensos a errores que los objetos mutables. |
| Ruta agregada | El "patrón ruta agregada" o "Aggregate Route Pattern" en inglés, se refiere a una funcionalidad en los sistemas de telefonía y comunicaciones que permite a los usuarios llamar a múltiples destinos utilizando un solo patrón de marcación o número de teléfono.  En otras palabras, la ruta agregada permite agrupar varios números de teléfono en un solo patrón de marcación, lo que facilita la administración y configuración de las llamadas entrantes y salientes. Por ejemplo, en lugar de tener que crear rutas de marcación separadas para cada número de teléfono de una empresa, se puede crear un solo patrón de marcación que abarque todos los números necesarios.  Esta funcionalidad es especialmente útil en empresas u organizaciones que tienen múltiples líneas telefónicas o números de extensión, ya que permite consolidar la administración de llamadas y reducir el tiempo y esfuerzo necesario para configurar las rutas de marcación. |
| Script de transacción | El término "patrón script de transacción" o "Transaction Script Pattern" en inglés, se refiere a un patrón de diseño utilizado en el desarrollo de software para manejar transacciones de base de datos en aplicaciones web o de negocio.  Este patrón implica encapsular la lógica de negocio y las operaciones de base de datos de una transacción en un único script o archivo de código, que puede ser ejecutado cuando se solicita una operación en particular. El script de transacción es responsable de realizar todas las acciones necesarias para completar la transacción, incluyendo la validación de entradas, la ejecución de operaciones de base de datos y la generación de respuestas para el usuario.  El patrón script de transacción es comúnmente utilizado en aplicaciones web que utilizan patrones de arquitectura como el Modelo-Vista-Controlador (MVC), donde el script de transacción se utiliza como controlador para manejar las solicitudes de los usuarios y coordinar la interacción entre el modelo de datos y la vista de usuario. |
| Tabla modular | El "patrón tabla modular" o "Modular Table Pattern" en inglés, es un patrón de diseño utilizado en el desarrollo de software para representar y manejar datos en una estructura modular y escalable. Este patrón se utiliza comúnmente en bases de datos relacionales y se basa en la idea de dividir una tabla grande en módulos o secciones más pequeñas y manejables.  Cada módulo de la tabla modular es responsable de manejar un conjunto específico de datos, lo que facilita la agregación y el análisis de datos en diferentes partes de la aplicación. Además, cada módulo puede ser diseñado para cumplir con requisitos específicos de rendimiento, seguridad y escalabilidad.  Este patrón también puede ayudar a mejorar la flexibilidad y la modularidad del código de la aplicación, ya que los módulos de la tabla modular pueden ser agregados, eliminados o modificados sin afectar al resto de la aplicación. |
| Objeto de transferencia de datos | El "patrón objeto de transferencia de datos" o "Data Transfer Object Pattern" en inglés, es un patrón de diseño utilizado en el desarrollo de software para transferir datos entre diferentes componentes o capas de una aplicación. Este patrón se utiliza para reducir el número de llamadas a la base de datos y mejorar el rendimiento de la aplicación al minimizar el tráfico de red.  El objeto de transferencia de datos (DTO, por sus siglas en inglés) es una clase que contiene datos que serán transferidos entre diferentes componentes de la aplicación. Los DTOs se utilizan para encapsular la información y los datos que deben ser transferidos entre diferentes capas de la aplicación, como la capa de presentación, la capa de servicios y la capa de acceso a datos.  El patrón DTO ayuda a simplificar el diseño de la aplicación al proporcionar un objeto que contiene solo los datos necesarios para realizar una operación específica. Esto ayuda a mejorar el rendimiento de la aplicación al reducir la cantidad de datos transferidos y el tiempo necesario para realizar una operación.  Además, el uso del patrón DTO también puede mejorar la seguridad de la aplicación al limitar la exposición de datos sensibles a través de la red. Por ejemplo, si un DTO solo contiene información necesaria para mostrar una lista de usuarios, no se transferirá información confidencial como contraseñas u otra información personal. |
| Delegado de negocios | El "patrón delegado de negocios" o "Business Delegate Pattern" en inglés, es un patrón de diseño utilizado en el desarrollo de software para separar la lógica de negocio de la capa de presentación o de la interfaz de usuario de una aplicación.  Este patrón implica la creación de una clase delegada que actúa como intermediario entre la capa de presentación y la capa de servicios o de negocio. El delegado de negocios es responsable de coordinar las solicitudes del cliente con los servicios de negocio, y oculta la complejidad de la implementación de los servicios detrás de una interfaz más simple y fácil de usar.  El patrón delegado de negocios también puede mejorar la flexibilidad y la modularidad de la aplicación al permitir que los servicios de negocio sean reemplazados o modificados sin afectar a la capa de presentación. Esto puede ser útil en situaciones en las que se requiere una actualización o modificación de los servicios de negocio sin afectar a la experiencia del usuario o a la funcionalidad de la aplicación.  Además, el uso del patrón delegado de negocios también puede mejorar la seguridad de la aplicación al proporcionar una capa de abstracción entre la capa de presentación y los servicios de negocio. Esto puede limitar la exposición de la lógica de negocio y los detalles de la implementación a través de la red. |
| Unidad de trabajo | El "patrón unidad de trabajo" o "Unit of Work Pattern" en inglés, es un patrón de diseño utilizado en el desarrollo de software para administrar transacciones y operaciones relacionales en una aplicación.  Este patrón implica la creación de una clase llamada "unidad de trabajo" que se encarga de manejar todas las operaciones relacionales que se realizan en una transacción. La unidad de trabajo mantiene un registro de todas las entidades que se han modificado durante la transacción y se asegura de que los cambios sean guardados en la base de datos de manera coherente y atómica.  El patrón unidad de trabajo también puede ayudar a mejorar el rendimiento de la aplicación al reducir el número de llamadas a la base de datos. En lugar de realizar una operación en la base de datos cada vez que se modifica una entidad, la unidad de trabajo mantiene un registro de todas las operaciones que se realizarán y las ejecuta en una sola transacción.  Además, el uso del patrón unidad de trabajo también puede mejorar la modularidad y la flexibilidad del código de la aplicación al proporcionar una capa de abstracción entre la lógica de negocio y la capa de acceso a datos. Esto permite que los desarrolladores realicen cambios en la lógica de negocio o en la capa de acceso a datos sin afectar al resto de la aplicación. |

Tabla 3. Patrones aplicados a la capa de negocio

#### CAPA COMÚN DE SOPORTE O CORTE TRANSVERSAL

La capa común de soporte, también conocida como capa transversal o capa compartida, es una capa en la arquitectura en capas de software que proporciona servicios comunes a todas las demás capas. Esta capa puede incluir componentes y servicios que se utilizan en toda la aplicación, como, por ejemplo, la autenticación, la autorización, la gestión de errores y el registro de eventos [9].

La capa común de soporte se encarga de proporcionar una funcionalidad centralizada que puede ser utilizada por todas las capas de la aplicación. Esta capa se utiliza para reducir la duplicación de código y para garantizar la coherencia y la consistencia en toda la aplicación.

Algunos de los componentes que pueden estar presentes en la capa común de soporte son:

* Servicios de autenticación y autorización: para garantizar que los usuarios estén autorizados a realizar acciones en la aplicación y para proteger los datos de la aplicación.
* Servicios de registro y seguimiento: para mantener un registro de eventos y errores en la aplicación y para permitir la supervisión y el análisis.
* Servicios de comunicación: para establecer la comunicación entre los diferentes componentes y servicios de la aplicación.
* Servicios de gestión de errores: para capturar y gestionar los errores que puedan surgir durante la ejecución de la aplicación.

La capa común de soporte se utiliza para mejorar la modularidad y la mantenibilidad de la aplicación, al separar la funcionalidad común de la aplicación en una capa separada. Esto permite que los cambios en la funcionalidad común se realicen en un solo lugar, en lugar de tener que realizar cambios en todas las capas de la aplicación.

Es importante tener en cuenta que el diseño de cada capa debe ser coherente y cohesivo, y que las interfaces entre las capas deben ser claras y bien definidas. Cada capa debe tener una responsabilidad clara y separada de las demás capas, y los componentes dentro de cada capa deben estar diseñados para cumplir con esa responsabilidad. Además, es importante que cada capa sea independiente y que pueda ser modificada sin afectar a las demás capas, lo que se conoce como principio de separación de preocupaciones.

### INTERFAZ ENTRE CAPAS

La interfaz entre cada capa en una arquitectura en capas de software es crucial para garantizar una comunicación efectiva entre ellas. Una interfaz bien diseñada permite que cada capa se comunique con las demás de manera eficiente y sin problemas [10].

A continuación, se describen algunas prácticas recomendadas para establecer una interfaz efectiva entre cada capa:

* Definir claramente los protocolos de comunicación: es importante definir claramente los protocolos de comunicación que se utilizarán entre cada capa. Esto incluye los formatos de los datos, los métodos de acceso y las reglas de validación.
* Utilizar estándares de la industria: es recomendable utilizar estándares de la industria para la comunicación entre capas. Esto ayuda a garantizar la interoperabilidad y la compatibilidad entre diferentes sistemas.
* Establecer una arquitectura de servicios: una arquitectura de servicios proporciona una forma estándar de exponer los servicios de una capa a las demás capas. Esto puede hacerse mediante servicios web, servicios REST o cualquier otra tecnología de servicios.
* Diseñar interfaces simples: las interfaces entre capas deben ser simples y fáciles de entender. Esto ayuda a garantizar una implementación coherente y una comunicación efectiva.
* Establecer contratos de servicio: los contratos de servicio establecen las expectativas y las responsabilidades de cada capa. Estos contratos deben ser acordados por todas las capas y adheridos durante todo el ciclo de vida del sistema.
* Probar y validar las interfaces: es importante probar y validar las interfaces entre capas para garantizar que se estén comunicando efectivamente. Esto puede hacerse mediante pruebas unitarias, pruebas de integración y pruebas de sistema.

Al establecer una interfaz efectiva entre cada capa, se garantiza una comunicación fluida y coherente entre ellas, lo que es esencial para una arquitectura en capas de software eficaz.

### IMPLEMENTACIÓN DE LA ARQUITECTURA EN CAPAS

La implementación de la arquitectura en capas de software puede variar según el lenguaje y la tecnología utilizada. Sin embargo, existen algunas prácticas recomendadas que se aplican a cualquier implementación de la arquitectura en capas.

En general, se recomienda utilizar una combinación de lenguajes y tecnologías que sean compatibles con la arquitectura en capas. Por ejemplo, los servicios web y los servicios REST son tecnologías comunes utilizadas en arquitecturas en capas [11].

A continuación, se describen algunas herramientas y frameworks disponibles para facilitar la implementación de la arquitectura en capas en diferentes lenguajes y tecnologías:

* Java: Para la implementación en Java, se pueden utilizar frameworks como Spring, Hibernate y Struts. Estos frameworks proporcionan soporte para la creación de servicios web, servicios REST y capas de persistencia de datos.
* .NET: Para la implementación en .NET, se puede utilizar el framework ASP.NET. Este framework proporciona soporte para la creación de servicios web, servicios REST y capas de persistencia de datos.
* Python: Para la implementación en Python, se puede utilizar el framework Django. Este framework proporciona soporte para la creación de servicios web y capas de persistencia de datos.
* Ruby: Para la implementación en Ruby, se puede utilizar el framework Ruby on Rails. Este framework proporciona soporte para la creación de servicios web y capas de persistencia de datos.
* PHP: Para la implementación en PHP, se puede utilizar el framework Laravel. Este framework proporciona soporte para la creación de servicios web y capas de persistencia de datos.

Además de estos frameworks, existen otras herramientas y tecnologías que pueden facilitar la implementación de la arquitectura en capas, como los contenedores de aplicaciones (por ejemplo, Docker) y los servicios en la nube (por ejemplo, AWS y Azure).

En resumen, la implementación de la arquitectura en capas de software puede variar según el lenguaje y la tecnología utilizada, pero existen herramientas y frameworks disponibles para facilitar la implementación. Lo importante es seguir las prácticas recomendadas para la implementación de una arquitectura en capas efectiva.

### SEGURIDAD EN UNA ARQUITECTURA EN CAPAS

La seguridad es un aspecto crítico en cualquier sistema de software y en una arquitectura en capas, cada capa debe ser protegida adecuadamente para evitar posibles vulnerabilidades. A continuación, se describen algunas mejores prácticas para asegurar la seguridad en una arquitectura en capas de software [9]:

* Autenticación y autorización: Cada capa debe contar con un mecanismo de autenticación y autorización para asegurar que solo los usuarios autorizados puedan acceder a la capa correspondiente.
* Encriptación: Es importante que los datos que se transmiten entre las diferentes capas estén encriptados para evitar que sean interceptados por terceros.
* Validación de entrada: Cada capa debe validar cuidadosamente los datos de entrada para evitar ataques de inyección de código malicioso. Esto es especialmente importante en la capa de presentación y la capa de lógica de negocio.
* Seguridad en la capa de presentación: Es importante asegurarse de que la capa de presentación no permita la ejecución de código malicioso. Para ello, se deben limitar las opciones del usuario y deshabilitar la ejecución de scripts maliciosos.
* Seguridad en la capa de lógica de negocio: La capa de lógica de negocio debe ser protegida para evitar que se realicen operaciones no autorizadas o se acceda a datos confidenciales. Es importante asegurarse de que esta capa tenga un acceso seguro a la capa de persistencia de datos.
* Seguridad en la capa de persistencia de datos: La capa de persistencia de datos debe ser protegida para evitar la exposición de datos confidenciales y el acceso no autorizado. Se deben implementar mecanismos de seguridad adecuados para proteger la base de datos, como la encriptación y el control de acceso.
* Monitoreo y registro de actividad: Es importante llevar un registro de la actividad del sistema y monitorear regularmente las diferentes capas para detectar posibles vulnerabilidades y tomar medidas preventivas.

En resumen, la seguridad es un aspecto crítico en una arquitectura en capas de software y cada capa debe ser protegida adecuadamente para evitar posibles vulnerabilidades. Es importante seguir las mejores prácticas de seguridad, implementar mecanismos de seguridad adecuados y monitorear regularmente el sistema para detectar posibles amenazas.

### PATRONES DE DISEÑO RELACIONADOS

Existen varios patrones de diseño que se utilizan comúnmente en conjunción con la arquitectura en capas de software para mejorar la calidad y la mantenibilidad del sistema. A continuación, se describen algunos de los patrones de diseño más comunes [10]:

* Patrón de Inyección de Dependencias (Dependency Injection): Este patrón se utiliza para facilitar la inyección de dependencias entre las diferentes capas de la arquitectura en capas. La idea es que cada capa no tenga una dependencia directa de la capa inferior, sino que reciba las dependencias necesarias a través de un mecanismo de inyección de dependencias.
* Patrón de Fábrica (Factory): Este patrón se utiliza para crear objetos de diferentes tipos de forma dinámica y flexible. En una arquitectura en capas, se puede utilizar el patrón de fábrica para crear objetos de la capa de lógica de negocio de forma dinámica, sin tener que conocer la implementación concreta de dicha capa.
* Patrón de Decorador (Decorator): Este patrón se utiliza para agregar funcionalidad adicional a un objeto existente sin modificar su estructura. En una arquitectura en capas, se puede utilizar el patrón de decorador para agregar funcionalidad a una capa específica sin modificar su estructura.
* Patrón de Observador (Observer): Este patrón se utiliza para definir una relación de dependencia entre objetos para que cuando un objeto cambie su estado, todos los objetos que dependen de él sean notificados y actualizados automáticamente. En una arquitectura en capas, se puede utilizar el patrón de observador para notificar a la capa superior cuando la capa inferior cambia su estado.
* Patrón de Repositorio (Repository): Este patrón se utiliza para abstraer la capa de persistencia de datos para facilitar la separación de responsabilidades. En una arquitectura en capas, se puede utilizar el patrón de repositorio para encapsular la lógica de acceso a la base de datos y ofrecer una interfaz más sencilla a la capa superior.

La utilización de patrones de diseño en conjunción con la arquitectura en capas de software puede mejorar la calidad y la mantenibilidad del sistema. Estos patrones de diseño permiten una mayor flexibilidad, escalabilidad y facilidad de mantenimiento, además de facilitar la separación de responsabilidades y la modularidad del sistema.

### VENTAJAS Y DESVENTAJAS

La arquitectura en capas es un enfoque comúnmente utilizado en el diseño de sistemas de software. A continuación, se describen algunas de las ventajas y desventajas de este enfoque:

|  |  |
| --- | --- |
| VENTAJAS | DESVENTAJAS |
| Modularidad:  La arquitectura en capas permite separar el sistema en módulos más pequeños y manejables, lo que facilita la comprensión y la mantenibilidad del código. | **Dificultad para probar:**  La arquitectura por capas puede dificultar la realización de pruebas completas en el sistema debido a la complejidad y la cantidad de componentes interdependientes. |
| Escalabilidad:  Al separar el sistema en capas, se puede escalar cada capa de forma independiente según las necesidades del sistema. | **Complejidad:**  La arquitectura en capas puede introducir una complejidad adicional en el diseño del sistema, lo que puede dificultar la comprensión del código. |
| Flexibilidad:  La arquitectura en capas permite cambiar la implementación de una capa sin afectar las otras capas, lo que permite una mayor flexibilidad y adaptabilidad del sistema. | **Overhead de comunicación:**  La comunicación entre capas puede ser costosa en términos de rendimiento, lo que puede limitar la escalabilidad del sistema. |
| Separación de responsabilidades:  La arquitectura en capas permite separar las responsabilidades de cada capa, lo que facilita la comprensión y la organización del código. | **Rigidez:**  La arquitectura en capas puede ser demasiado rígida en algunos casos, lo que puede limitar la flexibilidad y adaptabilidad del sistema. |

Tabla 4. Ventajas y desventajas de la arquitectura por capas

Es apropiado utilizar una arquitectura en capas cuando se busca una separación clara de responsabilidades y una mayor modularidad del sistema. Este enfoque es especialmente adecuado para sistemas grandes y complejos en los que se necesitan diferentes niveles de abstracción y escalabilidad. Sin embargo, en sistemas pequeños y sencillos, puede resultar innecesario y excesivamente complejo utilizar una arquitectura en capas.

En general, la elección de la arquitectura adecuada depende de varios factores, como la complejidad del sistema, los requisitos de rendimiento y escalabilidad, y las habilidades y preferencias del equipo de desarrollo. En algunos casos, puede ser apropiado utilizar otros enfoques de arquitectura, como la arquitectura basada en microservicios o la arquitectura orientada a eventos, que pueden ofrecer una mayor flexibilidad y adaptabilidad en determinados contextos.

### EVOLUCIONES

#### ARQUITECTURA EN CAPAS + DIP

La arquitectura en capas y el principio de inversión de dependencias (DIP) son dos conceptos importantes en el desarrollo de software. La arquitectura en capas se enfoca en dividir un sistema en capas lógicas, mientras que el principio de DIP se enfoca en establecer una separación entre los componentes del sistema y sus dependencias [10].

La arquitectura en capas puede ser compatible con el principio de DIP, ya que permite establecer una separación clara entre las diferentes capas del sistema. Por ejemplo, en una arquitectura en capas típica, la capa de presentación depende de la capa de aplicación y la capa de aplicación depende de la capa de datos. Sin embargo, al aplicar el principio de DIP, se busca que la capa de presentación no dependa directamente de la capa de aplicación, sino de una interfaz o abstracción que representa a la capa de aplicación.

La aplicación del principio de DIP en una arquitectura en capas puede ofrecer varios beneficios, como una mayor modularidad y facilidad de mantenimiento. Al separar los componentes del sistema de sus dependencias, se pueden realizar cambios y actualizaciones sin afectar al resto del sistema. Además, al utilizar abstracciones en lugar de implementaciones concretas, se pueden intercambiar componentes fácilmente sin modificar el resto del sistema.

Sin embargo, también es importante tener en cuenta que la aplicación del principio de DIP en una arquitectura en capas puede aumentar la complejidad del diseño y la cantidad de código necesario para implementar el sistema. Además, puede ser necesario definir una estructura de interfaces y abstracciones adecuada para que el sistema sea eficiente y fácil de mantener.

Diagrama

Descripción generada automáticamente

Figura 3. Arquitectura en capas flexible + DIP

#### ARQUITECTURA HEXAGONAL

La arquitectura hexagonal, también conocida como arquitectura de puertos y adaptadores, es una arquitectura de software que busca separar el núcleo de la aplicación de su entorno externo y de sus adaptaciones a diferentes tecnologías. La idea principal es separar la lógica de negocio y las reglas de negocio del código específico de la tecnología y las implementaciones de entrada y salida.

En la arquitectura hexagonal, el núcleo de la aplicación se encuentra en el centro y se comunica con el exterior a través de puertos. Un puerto es una interfaz que define cómo el núcleo de la aplicación se comunica con el exterior. Por ejemplo, un puerto podría definir cómo se realiza la lectura y escritura de datos en una base de datos o cómo se recibe y envía información por una interfaz de usuario [8].

En la arquitectura hexagonal, los adaptadores se encargan de conectar el núcleo de la aplicación con el exterior. Los adaptadores pueden ser de diferentes tipos, como adaptadores de base de datos, adaptadores de servicios web o adaptadores de interfaces de usuario. Los adaptadores implementan los puertos definidos por la aplicación y se encargan de transformar los datos en un formato que pueda ser utilizado por el núcleo de la aplicación.

La arquitectura hexagonal ofrece varios beneficios, como una mayor modularidad, facilidad de mantenimiento y una mayor independencia de la tecnología. Al separar la lógica de negocio del código específico de la tecnología y las implementaciones de entrada y salida, se pueden realizar cambios y actualizaciones en el sistema sin afectar al resto del sistema. Además, esta arquitectura permite una fácil adaptación a diferentes tecnologías y entornos.

Sin embargo, también es importante tener en cuenta que la aplicación de la arquitectura hexagonal puede aumentar la complejidad del diseño y la cantidad de código necesario. Además, puede ser necesario definir una estructura de puertos y adaptadores adecuada para que el sistema sea eficiente y fácil de mantener.

Diagrama

Descripción generada automáticamente

Figura 4. Arquitectura hexagonal

#### ARQUITECTURA ONION

La arquitectura Onion (también conocida como arquitectura de cebolla) es una arquitectura de software que se basa en la idea de que una aplicación se divide en capas, pero en lugar de tener capas fijas y rígidas como en otras arquitecturas, estas capas están organizadas en torno al núcleo de la aplicación y se construyen de forma progresiva, en capas concéntricas que rodean el núcleo.

El núcleo de la aplicación contiene la lógica de negocio y es independiente de cualquier tecnología externa. Las capas exteriores de la aplicación proporcionan servicios que interactúan con el exterior y que dependen de la tecnología específica utilizada en la aplicación [11].

En la arquitectura Onion, cada capa se comunica con la capa anterior a través de interfaces bien definidas. Cada capa es responsable de proporcionar un conjunto de servicios a la capa siguiente y consume los servicios de la capa anterior. Las capas superiores dependen de las capas inferiores, pero las capas inferiores no saben nada acerca de las capas superiores.

La arquitectura Onion se utiliza a menudo para aplicaciones web, donde el núcleo de la aplicación puede ser una API REST y las capas exteriores pueden ser servicios de la nube, servicios de autenticación, servicios de almacenamiento y servicios de interfaz de usuario. La idea es que cada capa proporcione un conjunto de servicios que puedan ser reutilizados en toda la aplicación.

La arquitectura Onion ofrece varios beneficios, como una mayor modularidad y facilidad de mantenimiento. Al separar la lógica de negocio de las capas que dependen de la tecnología, se pueden realizar cambios y actualizaciones en el sistema sin afectar al resto del sistema. Además, esta arquitectura permite una fácil adaptación a diferentes tecnologías y entornos.

Sin embargo, también es importante tener en cuenta que la aplicación de la arquitectura Onion puede aumentar la complejidad del diseño y la cantidad de código necesario. Además, puede ser necesario definir una estructura de capas adecuada para que el sistema sea eficiente y fácil de mantener.

Gráfico, Gráfico de proyección solar

Descripción generada automáticamente

Figura 5. Arquitectura Onion

#### ARQUITECTURA CLEAN

La arquitectura Clean (también conocida como arquitectura Hexagonal o Puertos y Adaptadores) es una arquitectura de software que se centra en separar las diferentes capas de una aplicación y hacer que cada capa sea independiente de las demás [10].

La arquitectura Clean se basa en la idea de que una aplicación se divide en tres capas principales:

* Capa de presentación: Esta capa es responsable de la interfaz de usuario de la aplicación. En esta capa se encuentran los componentes que interactúan con el usuario, como las pantallas, los formularios y los botones.
* Capa de dominio: Esta capa contiene la lógica de negocio de la aplicación. Aquí se definen las entidades y los casos de uso que representan la lógica central de la aplicación. Esta capa es independiente de la tecnología utilizada para implementar la aplicación.
* Capa de infraestructura: Esta capa es responsable de la integración de la aplicación con el entorno externo, como bases de datos, servicios web, sistemas de correo electrónico, etc. En esta capa se encuentran los adaptadores que permiten a la aplicación comunicarse con el exterior.

La arquitectura Clean se basa en el principio de que las capas de una aplicación deben ser independientes unas de otras. Por lo tanto, cada capa debe tener sus propias interfaces y no debe depender directamente de las capas adyacentes. En lugar de ello, las capas se comunican a través de interfaces, que permiten una fácil sustitución y modificación de las capas.

Además, la arquitectura Clean enfatiza la importancia de las pruebas automatizadas y la separación de las pruebas de unidad y las pruebas de integración. Las pruebas de unidad se centran en probar cada componente de forma aislada, mientras que las pruebas de integración se centran en comprobar el correcto funcionamiento de las interfaces entre las capas.

Diagrama

Descripción generada automáticamente

Figura 6. Arquitectura Clean

### MODELO ANÉMICO

El modelo anémico (o modelo anémico de dominio) es un patrón de diseño de software que se refiere a una aplicación que tiene una arquitectura basada en objetos, pero que carece de una lógica de negocio adecuada. En este modelo, los objetos sólo contienen datos y no tienen la capacidad de procesarlos.

El modelo anémico se caracteriza por tener una gran cantidad de objetos que representan entidades del mundo real, pero que no tienen métodos para realizar operaciones complejas en los datos que representan. En lugar de eso, la lógica de negocio se encuentra en capas superiores de la aplicación, como en los controladores o en los servicios.

El resultado de esto es que el modelo anémico puede ser menos eficiente y más difícil de mantener a largo plazo. Los desarrolladores pueden tener que crear muchos objetos y capas adicionales para manejar la lógica de negocio, lo que aumenta la complejidad de la aplicación. El modelo anémico también puede llevar a problemas de acoplamiento y cohesión en la aplicación, ya que los objetos pueden ser demasiado dependientes de otros objetos o no tener suficiente cohesión en su propia responsabilidad. Esto puede dificultar la comprensión y la modificación de la aplicación en el futuro.

En general, el modelo anémico se considera un antipatrón de diseño, ya que puede llevar a problemas de mantenimiento y escalabilidad en la aplicación. En su lugar, se recomienda un enfoque más integral para el diseño de aplicaciones, que incluya la lógica de negocio y los objetos de datos en un modelo de dominio cohesivo y bien definido.

# PARTE PRÁCTICA

## DESARROLLO DEL PROYECTO

Script para la creación de la base de datos e inserción de registros en SQL Server.

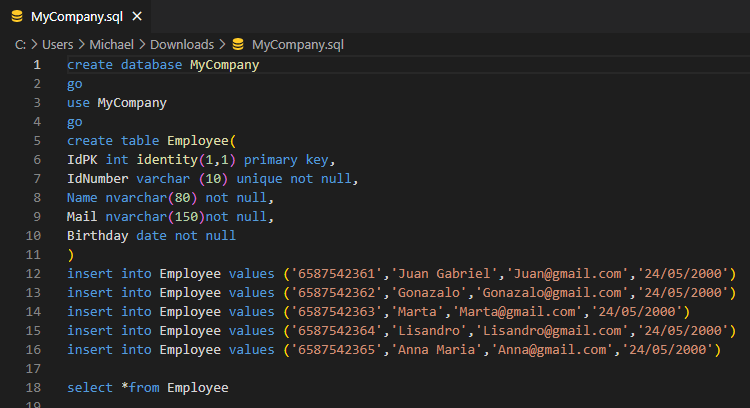


Figura 7. Script para SQL Server

Creación e inserción de registros en la herramienta de gestión de Base de Datos SQL Server Management Studio.

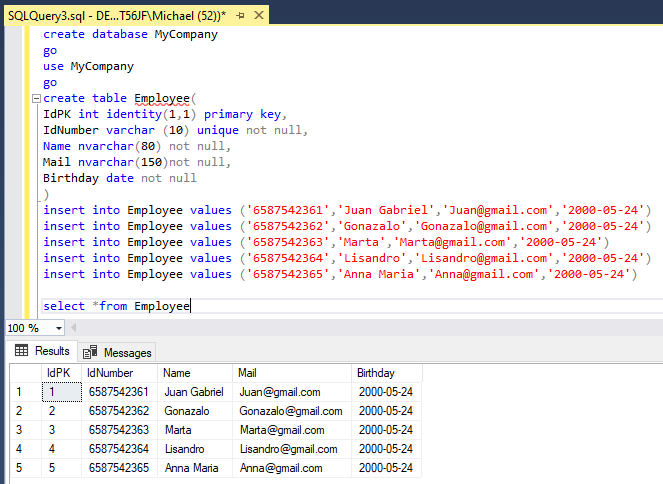


Figura 8. Inserción de Registros en SQL Server Management Studio

A continuación, crear un nuevo proyecto en Visual Studio seleccionando la platilla Solución en Blanco.

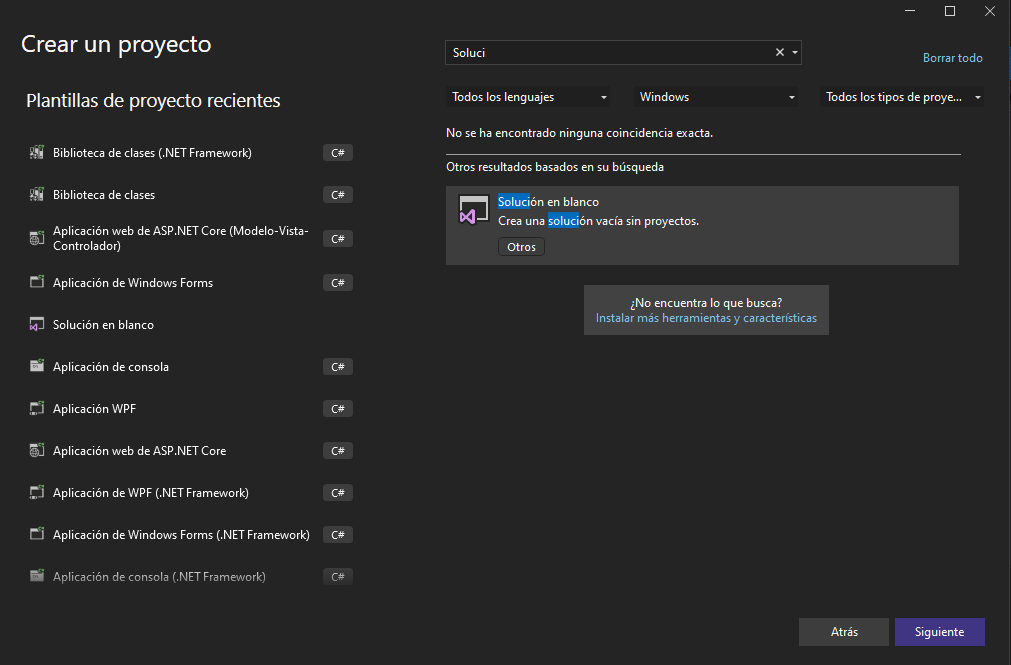


Figura 9. Selección de la Plantilla para el Proyecto

Asignar un nombre y ubicación para el proyecto.

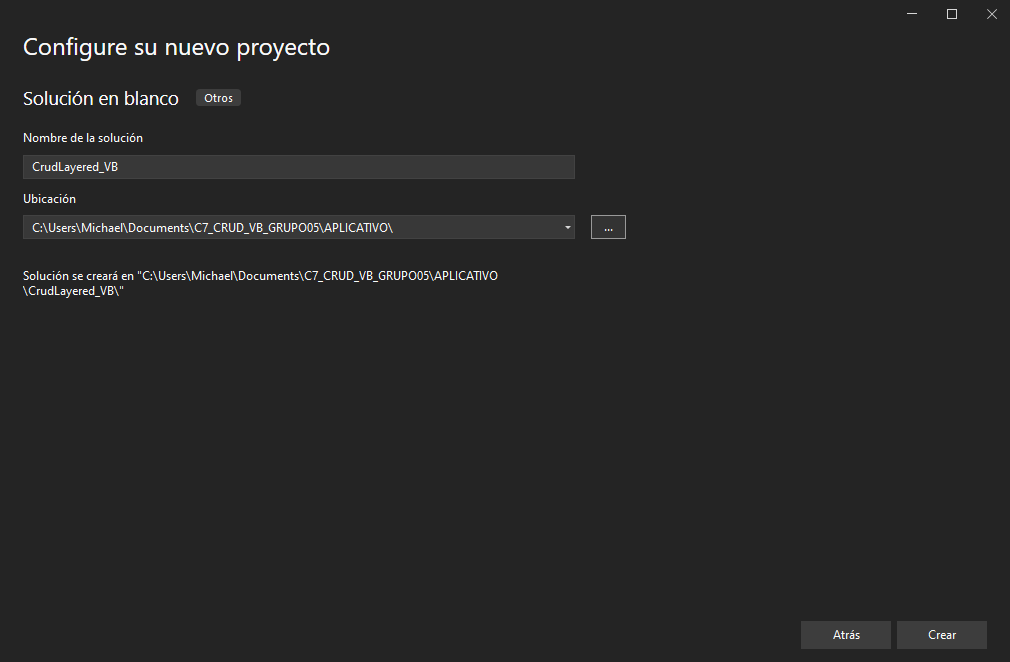


Figura 10. Nombre y Ubicación del Proyecto

Agregar un nuevo proyecto del tipo Windows Form a la solución.

Captura de pantalla de computadora

Descripción generada automáticamente

Figura 11. Creación de un archivo de vista Windows Forms

En la siguiente ventana, seleccionar el tipo de archivo Aplicación de Windows Forms.

Texto

Descripción generada automáticamente

Figura 12. Archivo de Vista Aplicación de Windows Forms

Asignar un nombre a la Aplicación de Windows Forms.

Texto

Descripción generada automáticamente

Figura 13. Definir el nombre para la Aplicación de Windows Forms

Agregar un nuevo proyecto del tipo Biblioteca de Clases a la solución.

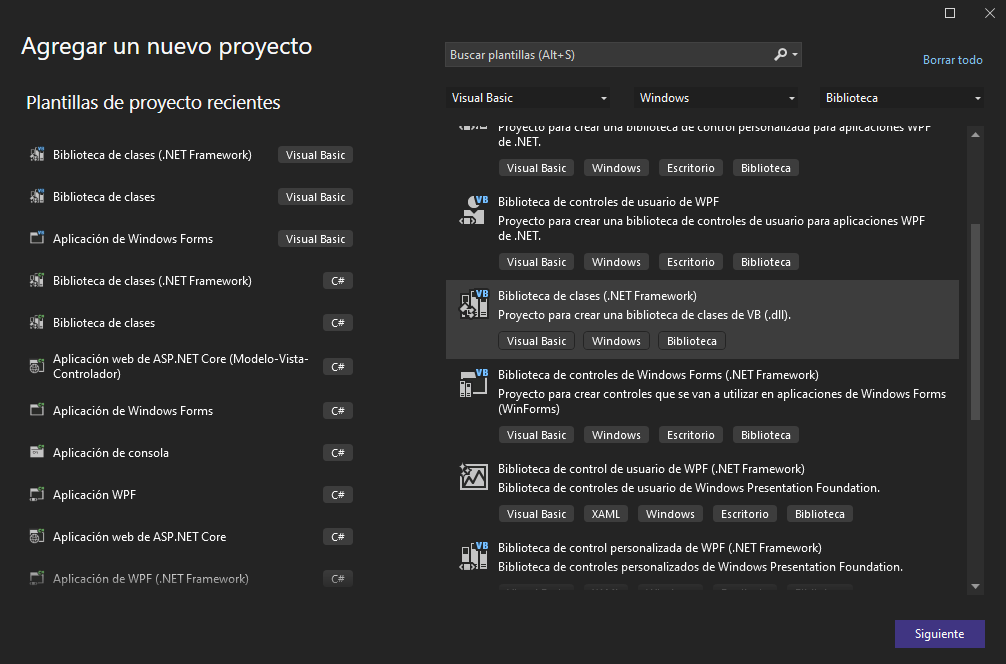


Figura 14. Agregar tipo de proyecto Biblioteca de Clases

Asignar el nombre para la Biblioteca de clases.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Figura 15. Nombre Domain para la Biblioteca de clases

Agregar otro proyecto nuevo de tipo Biblioteca de clases a la solución.

Texto

Descripción generada automáticamente

Figura 16. Biblioteca de Clases para la Solución

Se asigna un nombre para la Biblioteca de clases.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Figura 17. Nombre DataAccess para la Biblioteca de Clases

A continuación, se muestra la estructura inicial del proyecto.

Captura de pantalla de computadora

Descripción generada automáticamente

Figura 18. Estructura inicial para el proyecto

Una vez definida la estructura para el proyecto, se agrega las referencias entre las capas.

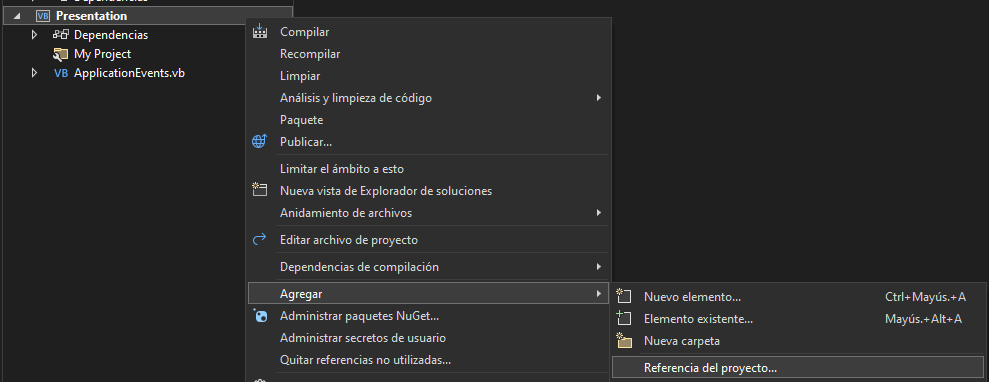


Figura 19. Referencias entre Capas

Seleccionar las capas que se referenciaran para la solución y clic en Aceptar.

Captura de pantalla de un celular

Descripción generada automáticamente

Figura 20. Selección de las capas a referenciar

Se repite el mismo proceso para la capa de DataAccess y clic en Aceptar.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Figura 21. Referencia de la capa DataAccess

A continuación, se reestructuramos el proyecto con las subcarpetas para cada capa.

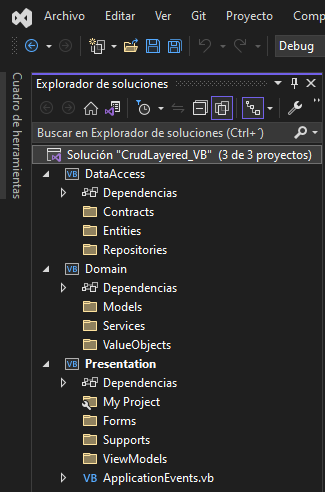


Figura 22. Carpetas para la estructura de cada capa

Para la capa DataAccess se define una clase Employee con la siguiente codificación:

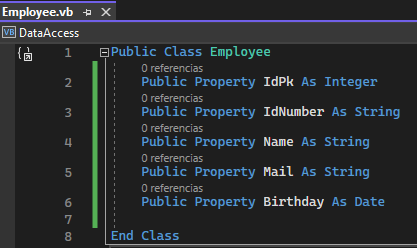


Figura 23. Codificación de la clase Employee

En la carpeta Contracts de la capa DataAccess se crea una interface.

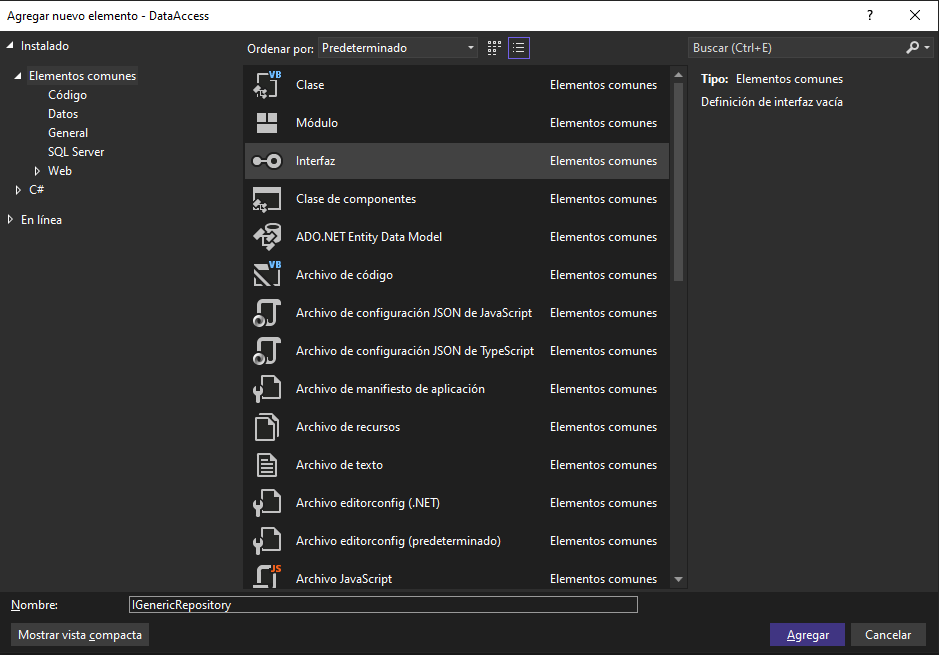


Figura 24. Interfaz en la carpeta Contracts en DataAccess

En la interfaz creada se defninen los siguientes metodos:

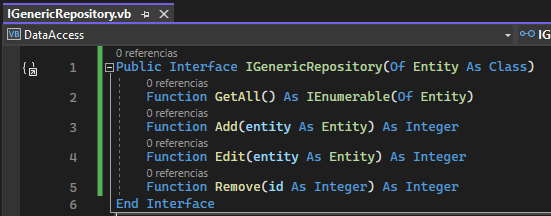


Figura 25. Métodos de la interfaz IGenericRepository

En la misma carpeta Contracts se crea otra interfaz llamada IEmployeeRepository con la siguiente codificación:

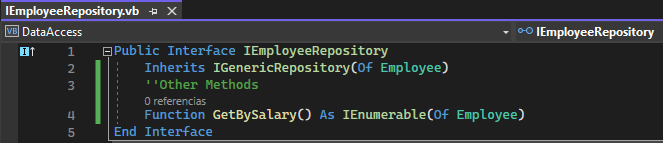


Figura 26. Codificación de la interfaz IEmployeeRepository

A continuación, se configura la conexión a la base de datos, para ello se tiene que configurar la cadena de conexión.

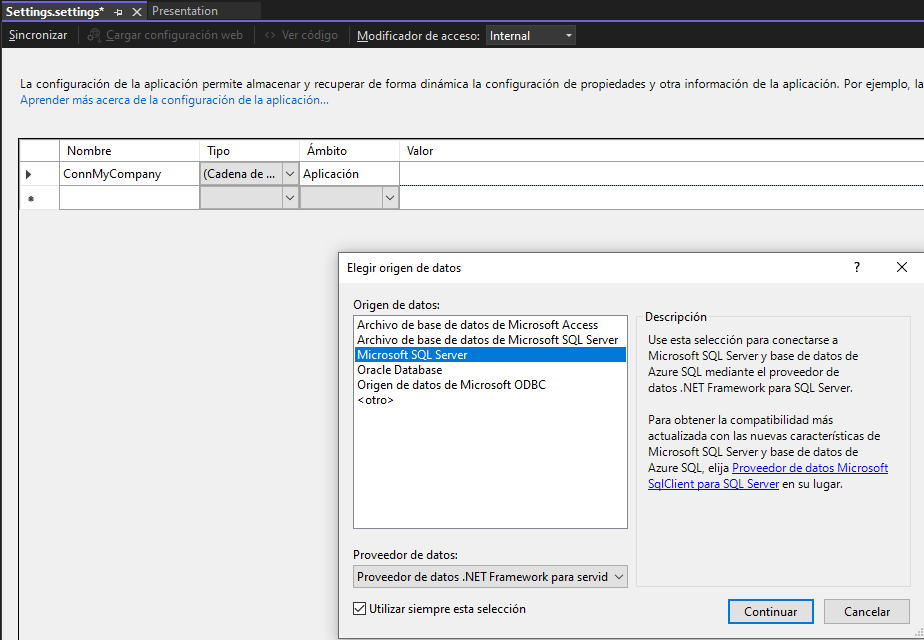


Figura 27. Conexión a la Base de Datos

Se elige el origen de los datos, en este caso Microsoft SQL Server.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Figura 28. Selección del origen de datos

Una vez seleccionado el origen de los datos, configurar las propiedades de conexión.

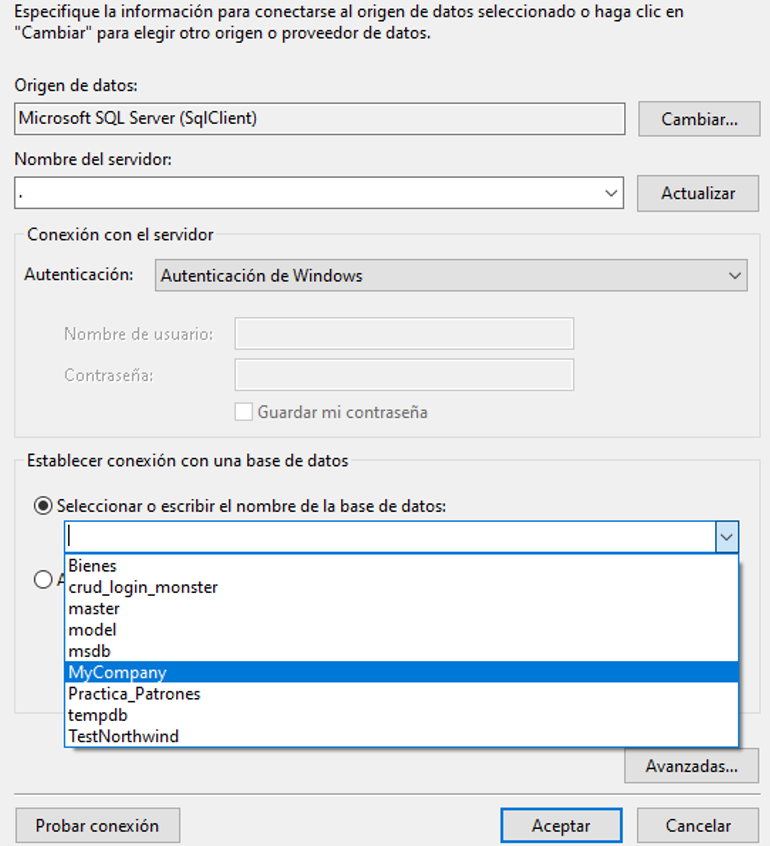


Figura 29. Propiedades de conexión a la Base de Datos

En el archivo App.config se define la configuración como se muestra a continuación:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Figura 30. Archivo de configuración App.config

A continuación, se agrega una referencia a System.Configuration.

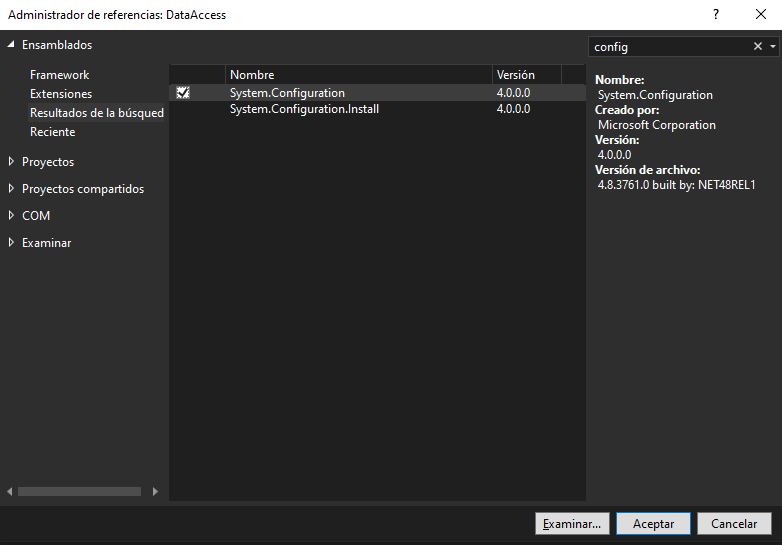


Figura 31. Referencia a System.Configuration

En la capa DataAccess se crea una nueva clase Repository con la siguiente codificación:

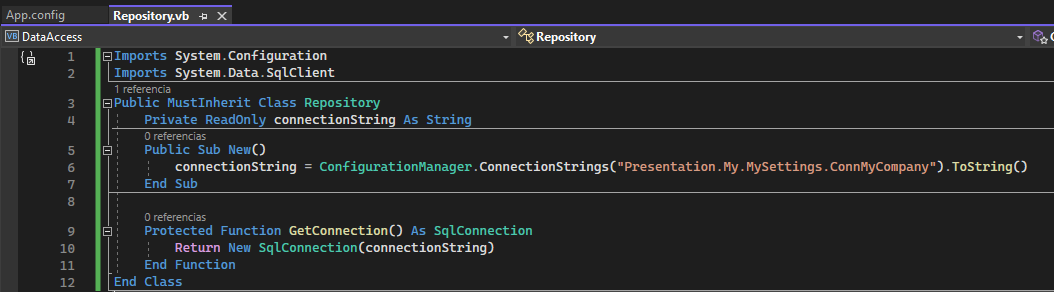


Figura 32. Clase Repository en la capa DataAccess

En la capa DataAccess se crea otra clase MasterRepository con la la siguiente codificación:

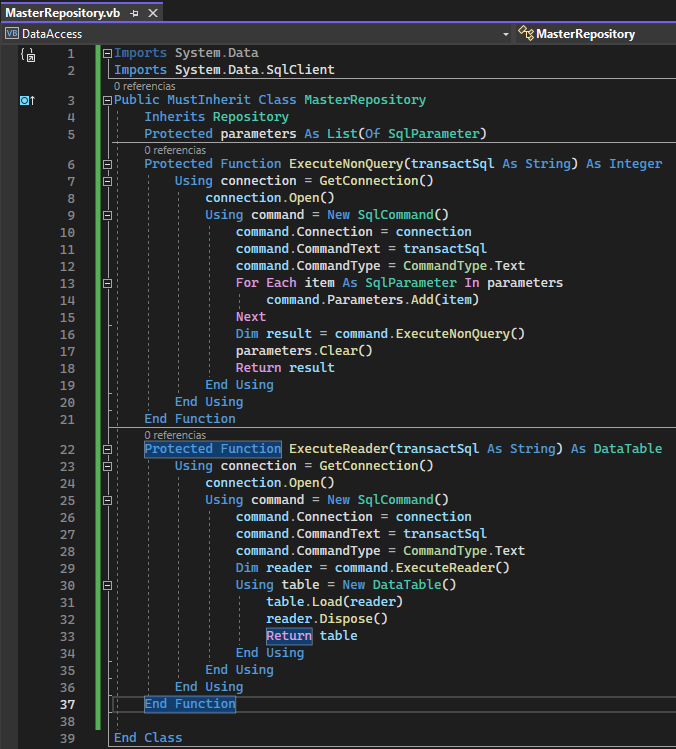


Figura 33. Clase MasterRepository en la capa DataAccess

Se define otra clase en la capa DataAccess con el nombre EmployeeRepository con la siguiente codificación:

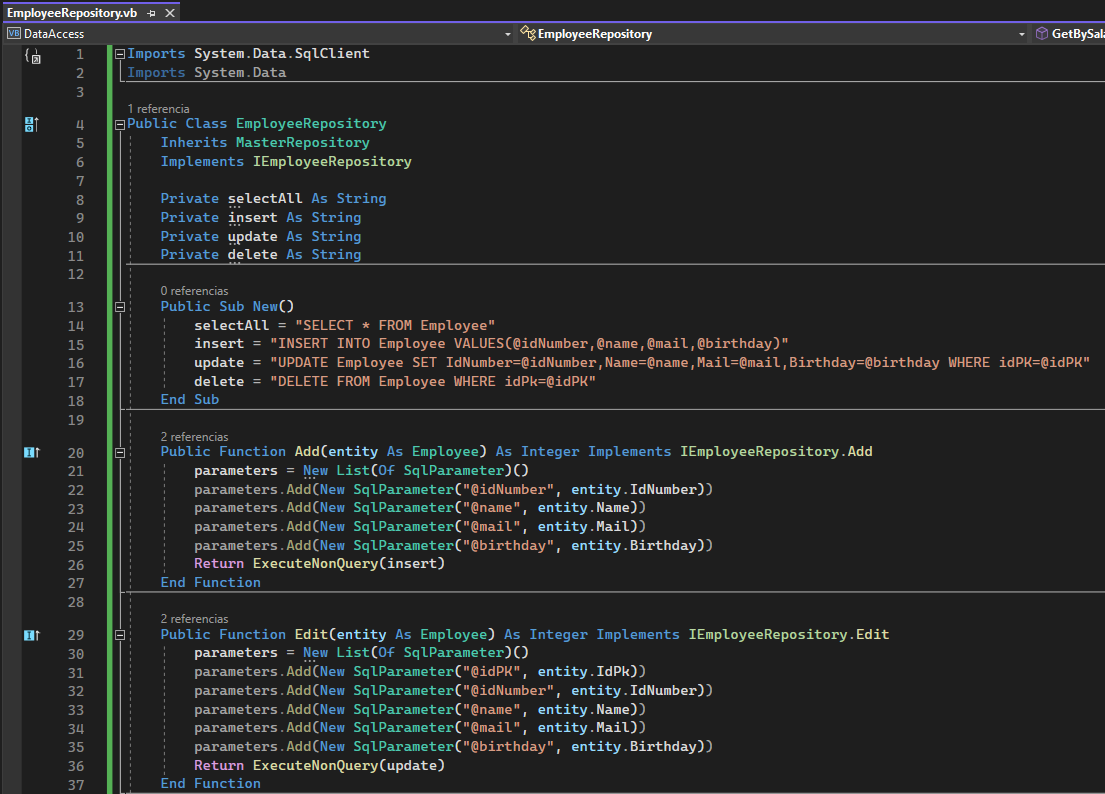


Figura 34. Clase EmployeeRepository en la capa DataAccess

Se define un tipo de dato Enum con el nombre EntityState en la capa Domain.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Figura 35. Enum EntityState en la capa Domain

A continuación, se agrega una nueva referencia a System.ComponentModel.DataAnnotations.

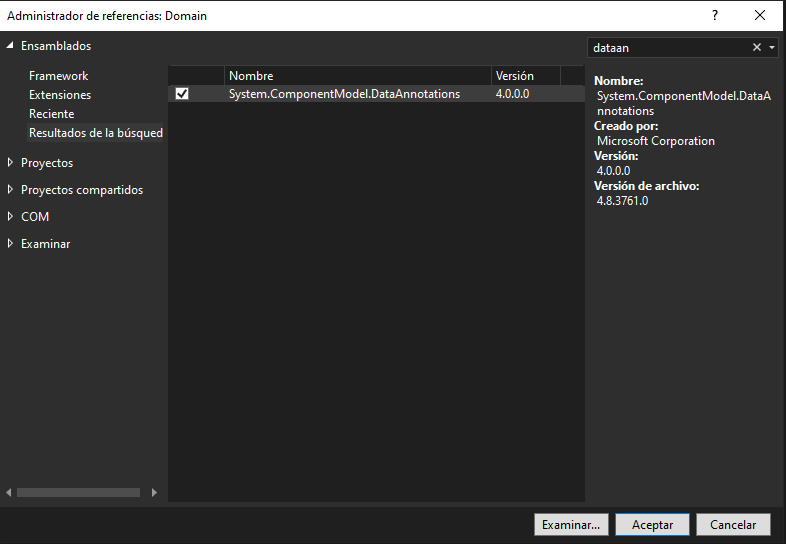


Figura 36. Referencia a System.ComponentModel.DataAnnotations

Una vez configurada la referencia, se crea la vista agregando un nuevo elemento en la capa de presentación Formulario (Windows Forms).

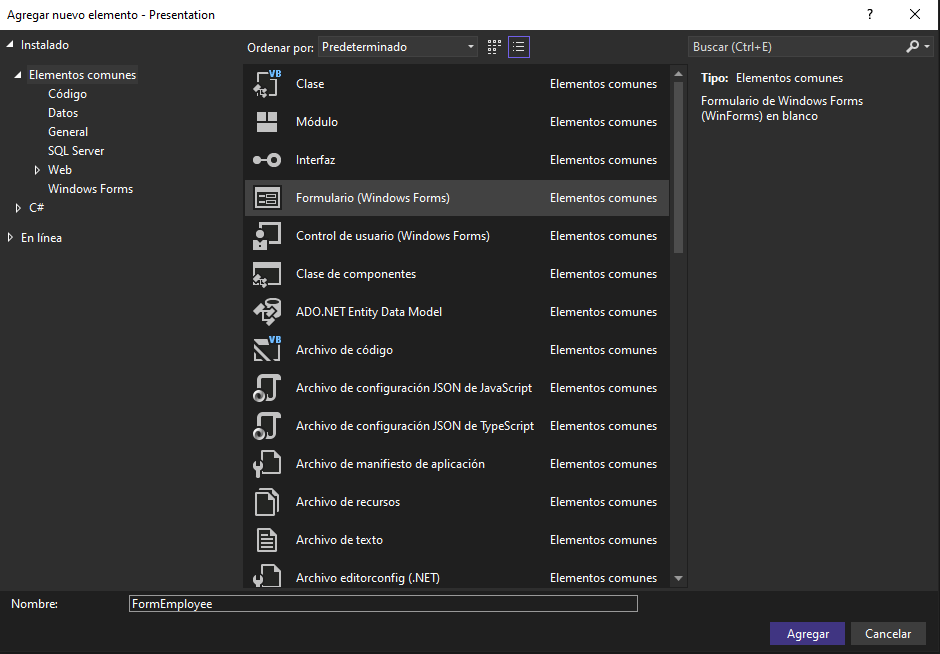


Figura 37. Formulario en la capa de Presentación

## EJECUCIÓN

Al ejecutar el proyecto, se verifica la visualización de los datos en la tabla de la vista FormEmployee.

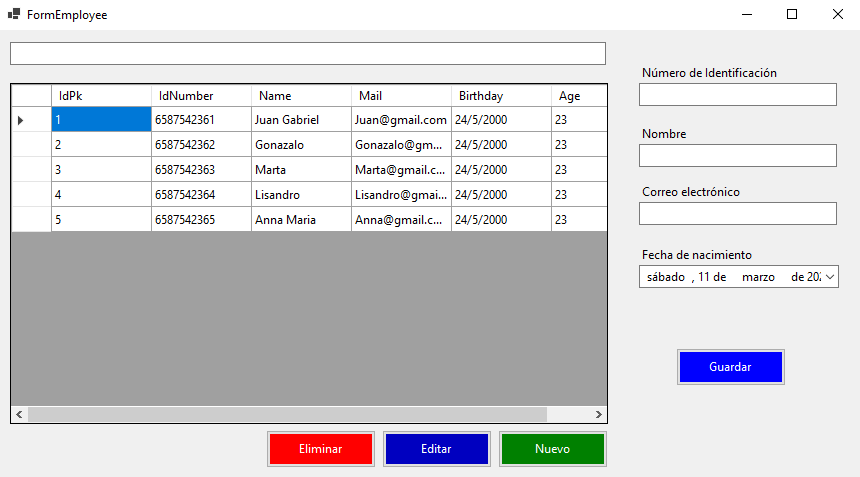


Figura 38. Ventana con los datos de los empleados

En la entrada de texto, se ingresa el valor por el que se van a filtrar los datos que se muestran en la tabla.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Figura 39. Datos filtrados en la tabla de FormEmployee

A continuación, se verifica la validación de los datos ingresados en el formulario.

Interfaz de usuario gráfica, Aplicación, Tabla

Descripción generada automáticamente

Figura 40. Validación de las entradas en el formulario

Una vez validados las entradas de texto en el formulario, se registra un nuevo empleado como se muestra en la figura:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Figura 41. Registro de un nuevo empleado

Una vez guardado el nuevo empleado, este se muestra en la tabla de empleados.

Interfaz de usuario gráfica

Descripción generada automáticamente

Figura 42. Vista del nuevo registro en la tabla de empleados

Para editar un registro de empleado se selecciona la fila en la tabla.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Figura 43. Editar el registro de un empleado

Una vez cambiado los datos del registro en el formulario, se da clic en Guardar.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Figura 44. Editar los datos del empleado seleccionado

Una vez editado el registro, este se muestra actualizado en la tabla de empleados.

Interfaz de usuario gráfica

Descripción generada automáticamente

Figura 45. Registro de datos actualizado en la tabla

A continuación, se selecciona un registro para eliminarlo.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Figura 46. Eliminar un registro de empleado

Una vea eliminado el registro, se muestra un mensaje de confirmación.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Figura 47. Notificación de registro eliminado

# CONCLUSIONES

* La arquitectura en capas puede presentar desafíos, como la necesidad de coordinar la comunicación entre las distintas capas y mantener la consistencia de los datos en toda la aplicación. En aplicaciones complejas, también puede resultar difícil definir las responsabilidades de cada capa y mantener la separación adecuada entre ellas.
* Aunque la arquitectura en capas es una herramienta poderosa en el diseño de software, su implementación exitosa requiere un enfoque cuidadoso y un buen conocimiento de las necesidades y objetivos específicos de la aplicación en cuestión.
* La arquitectura en capas es un enfoque popular en el diseño de software debido a su habilidad de separar las diferentes funciones de una aplicación en capas independientes y comunicarlas mediante interfaces definidas.
* Al dividir la aplicación en capas, cada capa puede ser desarrollada, probada y mantenida independientemente, lo que facilita la escalabilidad y modularidad del software. Además, esta arquitectura ofrece una mayor flexibilidad y capacidad de prueba, permitiendo a los equipos de desarrollo identificar y resolver problemas de manera más rápida y eficiente.

# RECOMENDACIONES

* La arquitectura en capas puede ser una buena opción para aplicaciones complejas, pero puede ser excesiva para aplicaciones más simples. Es importante evaluar la complejidad de la aplicación y determinar si la arquitectura en capas es la mejor opción para el diseño de la aplicación.
* Mantener la coherencia de los datos en toda la aplicación es crucial para evitar problemas de integridad de datos. Para ello, se deben implementar técnicas específicas, como la validación de datos en la capa de acceso a datos y la coordinación adecuada entre las diferentes capas.
* Como en cualquier arquitectura de software, es fundamental realizar pruebas exhaustivas para garantizar que la aplicación funcione correctamente y cumpla con los requisitos del usuario. Las pruebas deben incluir pruebas unitarias y pruebas de integración entre las diferentes capas de la aplicación.
* Para evitar problemas de dependencia y lograr mayor modularidad y flexibilidad, es fundamental definir con claridad las responsabilidades de cada capa y mantener su separación adecuada.
* Las interfaces entre las capas deben ser bien definidas y claras para evitar confusiones y problemas de comunicación. Asimismo, deben ser lo suficientemente flexibles para permitir cambios y actualizaciones sin afectar el funcionamiento de la aplicación.
* Si bien la arquitectura en capas puede mejorar la escalabilidad y la capacidad de prueba de la aplicación, también puede afectar su rendimiento. Por ello, es importante tener en cuenta tanto la escalabilidad como el rendimiento desde el principio y diseñar la arquitectura en consecuencia.

# **BIBLIOGRAFÍA**

1. Ramírez, J. (2023). Modelo de Arquitectura de Software. 46
2. Al-Fagih, Z., & Aldabbagh, A. (2018). A layered architecture model for smart city applications. IEEE Access, 6, 18646-18654.
3. Ayoubi, R., Ghosh, S., & Pathak, A. (2019). A layered architecture for the internet of things. In 2019 IEEE 5th World Forum on Internet of Things (WF-IoT) (pp. 395-400). IEEE.
4. Azim, T., & Kowalczyk, R. (2018). A layered architecture for context-aware computing systems. In Proceedings of the 2018 4th International Conference on Frontiers of Educational Technologies (ICFET 2018) (pp. 283-289). Atlantis Press.
5. Bari, M. F., Sharma, R. K., & Riaz, M. (2018). A review of layered architecture and protocol stacks for internet of things (IoT). Journal of Network and Computer Applications, 116, 39-55.
6. Gheorghe, L. A., & Gheorghe, C. M. (2020). A layered architecture for distributed systems. In Proceedings of the 2020 8th International Conference on Computers Communications and Control (ICCCC) (pp. 581-586). IEEE.
7. Kavi, K. M., & Bhatia, A. K. (2019). A layered architecture for cloud computing: A survey. Journal of Network and Computer Applications, 128, 37-55.
8. Khanna, S., Singh, G., & Singh, P. (2021). A layered architecture for edge computing: A review. Journal of Network and Computer Applications, 183, 103042.
9. Li, C., Guo, X., & Zhang, Y. (2019). A three-layered architecture for industrial internet of things. IEEE Access, 7, 133366-133375.
10. Nguyen, P. T., Nguyen, Q. T., & Nguyen, T. N. (2019). A four-layered architecture for blockchain-based applications. In 2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS) (pp. 628-633). IEEE.
11. Zhang, Q., Li, Y., & Liu, Z. (2019). A layered architecture for blockchain-based supply chain management systems. IEEE Access, 7, 114584-114595.