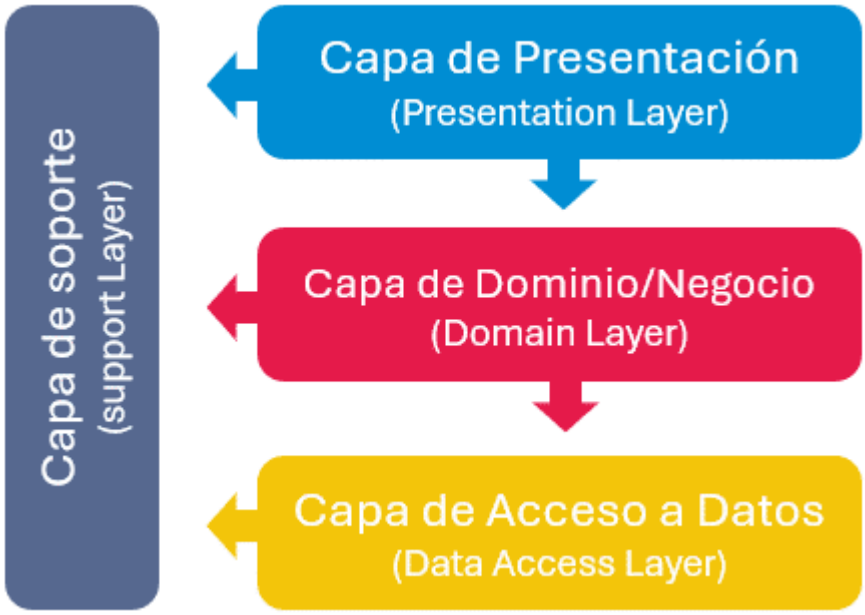
Patrones de diseño de arquitectura en capas

CÓRDOBA CREA FUTURO

1. Patrones de diseño de arquitectura en capas

Programar con patrones de arquitectura en capas mejora la organización, la mantenibilidad y la flexibilidad de las aplicaciones. Al separar el código en capas con responsabilidades específicas —como presentación, lógica de negocio y acceso a datos—, es posible realizar cambios en una capa sin afectar a las demás. Esto facilita las pruebas, las actualizaciones y la escalabilidad del sistema.

Figura 1. Patrón de diseño por capas



Fuente: elaboración propia

Para contextualizar los patrones de diseño con los temas tratados anteriormente, se presenta un resumen de los conceptos más importantes.

Introducción a la programación orientada a objetos

Principios SOLID —

acrónimo que agrupa cinco principios que favorecen el desarrollo eficiente, replicable, mantenible y escalable de software.

Beneficios de la modularidad y reutilización

al dividir un sistema en partes independientes (módulos), es posible reutilizar componentes en diferentes proyectos o secciones de una aplicación. Esto ahorra tiempo, mejora la consistencia y facilita la depuración y las actualizaciones.

Relación con patrones de diseño

los patrones de diseño son soluciones genéricas y reutilizables para problemas comunes en el desarrollo de software, aplicables en diversos contextos de programación. Su utilidad radica en que ofrecen un lenguaje común y plantillas para crear software más flexible, mantenible, escalable y de alta calidad. No se trata de código listo para usar, sino de pautas y descripciones de soluciones que los desarrolladores deben adaptar a sus proyectos.

Arquitectura en capas

La arquitectura en capas es un modelo de diseño de software que organiza una aplicación en capas horizontales, donde cada una cumple una responsabilidad específica y se comunica únicamente con la capa inmediatamente contigua. Este enfoque favorece la modularidad, la escalabilidad y el mantenimiento, ya que permite separar las

preocupaciones: los cambios en una capa no afectan directamente a las demás.

Un ejemplo habitual es la arquitectura de tres capas: presentación, lógica de negocio y acceso a datos. La capa de presentación interactúa con el usuario; la de lógica de negocio contiene las reglas del sistema; y la de acceso a datos gestiona la comunicación con la base de datos.

El diagrama general de arquitectura en capas representa gráficamente esta organización jerárquica y permite visualizar con claridad cómo se distribuyen las responsabilidades dentro del sistema.

Capa de presentación o IU

La capa de presentación, o interfaz de usuario, es la responsable de gestionar la interacción con el usuario. Incluye elementos como vistas renderizadas, archivos HTML de una página web, interfaces de línea de comandos o formularios de una aplicación de escritorio.

En esta capa, es posible aplicar diversos patrones de diseño que facilitan la organización del código y mejoran la separación de responsabilidades. Entre ellos, se encuentran los siguientes:

- Modelo-vista-controlador (MVC)
- Modelo jerárquico vista controlador (HMVC – PAC)
- Modelo vista presentador (MVP)
- Modelo vista – vista modelo (MVVM)
- Modelo vista presentador vista modelo
- *Page controller*
- *Front controller*

- *Application controller*

Capa de negocio o dominio

La capa de negocio, también conocida como capa de dominio, es el núcleo de la aplicación. Contiene la lógica principal, las reglas y las políticas que determinan el funcionamiento del sistema. Su propósito es encapsular los conceptos, entidades y comportamientos propios del negocio, sin depender de la interfaz de usuario ni de elementos técnicos como bases de datos o servicios externos.

Esta capa es opcional y se implementa solo cuando la complejidad de la lógica del sistema lo requiere. En ella, se pueden aplicar distintos patrones que ayudan a estructurar el código y representar mejor los procesos del negocio, tales como:

- objeto de valor;
- ruta agregada;
- script de transacción;
- tabla modular;

- objeto de transferencia de datos;
- *delegado de negocios;*
- *unidad de trabajo.*

Capa de acceso de datos o persistencia

La capa de acceso de datos, también llamada capa de persistencia o capa de integración, se encarga de almacenar y recuperar información desde un almacén de datos, como una base de datos, un servicio externo o un archivo plano.

En esta capa, se pueden aplicar diversos patrones que facilitan la interacción con las fuentes de datos y mejoran la organización del código:

- Enlace a datos de tabla
- Registro activo
- Mapeado de datos
- Objeto de consulta
- Objeto de acceso a datos

- *Repositorio*

¿Qué es la arquitectura en capas?

Como se mencionó, la arquitectura en capas es un patrón de diseño que divide una aplicación de programación orientada a objetos en distintos niveles, cada uno con una responsabilidad específica. Este enfoque organiza el código y facilita tanto su desarrollo como su mantenimiento.

Cada capa se comunica únicamente con la capa inmediatamente inferior, lo que favorece la modularidad, la separación de responsabilidades y la escalabilidad de la aplicación. Un ejemplo común en Java incluye capas de presentación, lógica de negocio y acceso a datos.

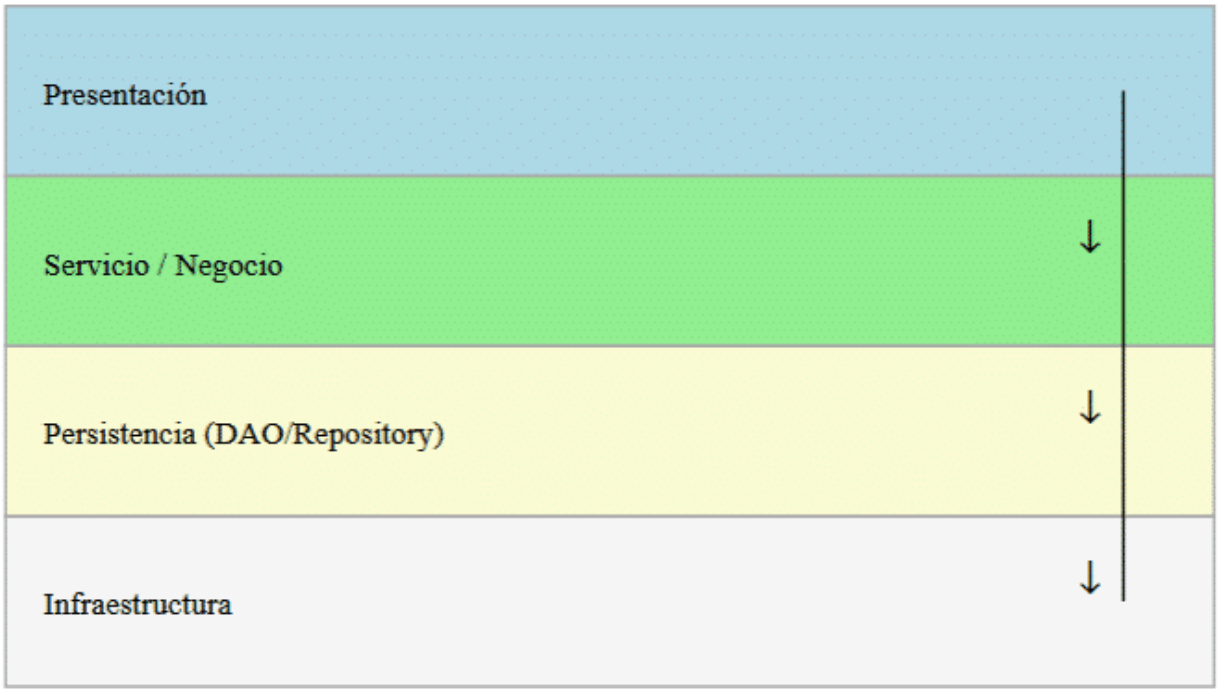
¿Cómo funciona?

La arquitectura en capas se basa en la separación de responsabilidades, donde cada capa se especializa en una tarea específica. Por ejemplo, una capa se encarga de la interfaz de usuario, otra gestiona la lógica de negocio y otra accede a la base de datos.

La comunicación entre capas es secuencial: una capa superior no puede interactuar directamente con una capa

inferior que no sea la siguiente en la jerarquía. Por ejemplo, la capa de presentación no accede directamente a la base de datos, sino que solicita la información a la capa de lógica de negocio, la cual, a su vez, se comunica con la capa de persistencia.

Figura 2. Modelo de separación de responsabilidades



Fuente: elaboración propia

Independencia de las capas

Las capas pueden desarrollarse, modificarse e incluso desplegarse de forma independiente, siempre que se respeten las interfaces de comunicación con las capas adyacentes. Esta característica permite una mayor flexibilidad en el desarrollo y mantenimiento del sistema (Oracle, s.f.).

Capas típicas en Java

En aplicaciones desarrolladas con Java, es común estructurar el sistema en capas bien definidas. A continuación, se describen las más utilizadas:

Capa de presentación —

Gestiona la interfaz de usuario y la interacción con el usuario. Suele implementarse con tecnologías como Swing, JavaFX o frameworks web como Spring MVC.

Capa de lógica de negocio —

contiene la lógica principal de la aplicación. Recibe las solicitudes desde la capa de presentación, las procesa y delega las operaciones a la capa inferior.

Capa de acceso a datos o persistencia

se encarga de la interacción con la fuente de datos, como una base de datos, para almacenar o recuperar información.

Capa de entidades

representa los objetos del dominio (por ejemplo, «usuario» o «producto»). Está compuesta por clases que contienen datos y se transfieren entre las distintas capas.

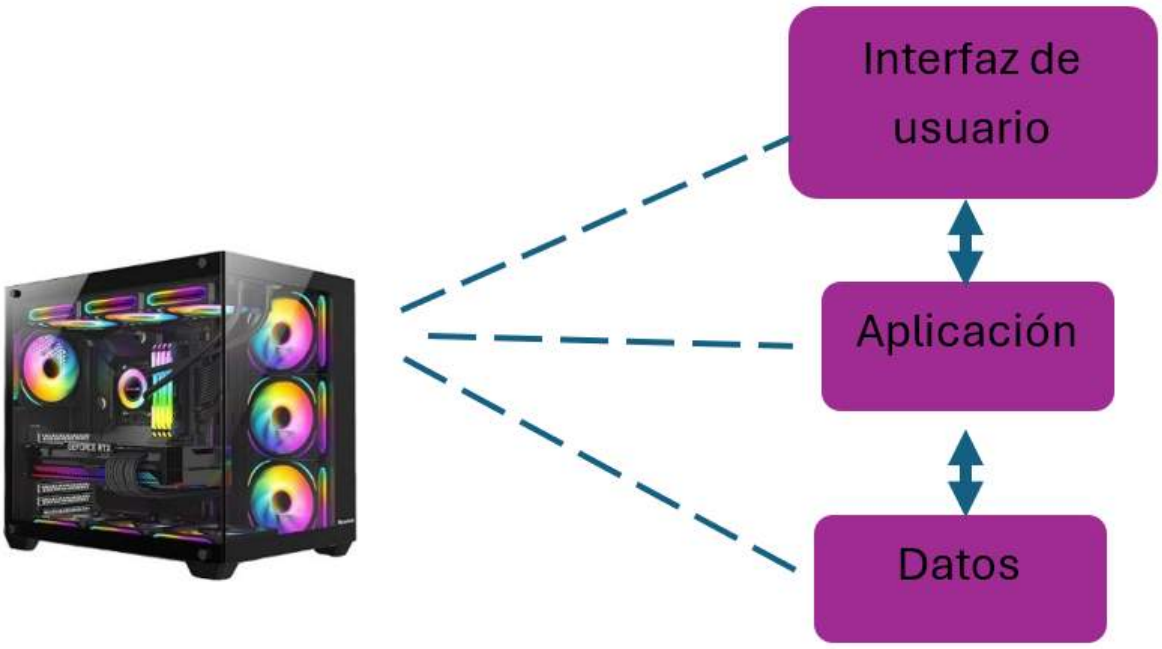
Patrón de diseño

El patrón de diseño por capas, también conocido como arquitectura de capas, es un enfoque arquitectónico ampliamente utilizado en el desarrollo de aplicaciones de software.

Como se mencionó anteriormente, su objetivo principal es dividir una aplicación en un conjunto de capas o niveles

lógicos, cada uno con responsabilidades específicas, tal como se puede apreciar en la figura 3

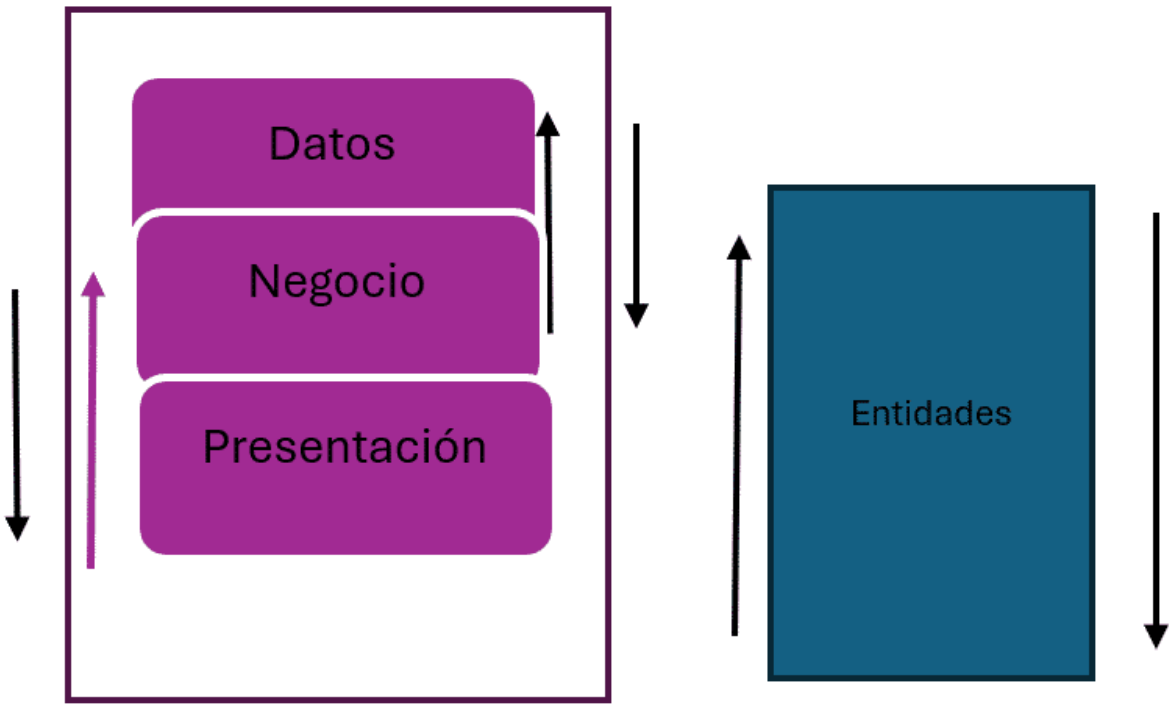
Figura 3. Patrón de diseño



Fuente: elaboración propia

Cada capa se comunica con las capas adyacentes a través de interfaces bien definidas.

Figura 4. Separación de funciones por capas



Fuente: elaboración propia

Como se ha visto, el patrón de diseño por capas promueve la modularidad, la mantenibilidad y la escalabilidad del software, además de facilitar la reutilización del código.

Capa de presentación

La capa de presentación (*presentation layer*) es la parte del sistema con la que interactúa el usuario. Contiene la interfaz de usuario, la lógica de presentación y la gestión de eventos. Puede tratarse de una interfaz gráfica (GUI) en una aplicación de escritorio o de una interfaz web en una aplicación web.

Una forma de entender esta capa es mediante una analogía: representa la planta baja de un edificio, la fachada. Es lo que los usuarios ven y con lo que interactúan. Aquí se gestionan la interfaz, la presentación de los datos y las acciones del usuario. Es como el salón donde se recibe a los invitados y se les muestra dónde está todo.

Capa de lógica de negocio

La capa de lógica de negocio (*business logic layer*) contiene la lógica principal de la aplicación. En ella se implementan las reglas y la funcionalidad específica del sistema. Es la

responsable de procesar datos, realizar cálculos y tomar decisiones basadas en la lógica del negocio.

Por ejemplo, si se necesita dar de alta a un usuario, en esta capa se encuentra la lógica que permite agregarlo, validarlo y preparar la información necesaria para que la capa de datos lo registre en la base de datos.

Siguiendo con la analogía anterior, esta capa sería como subir al primer piso del edificio: aquí se encuentran las reglas y los procesos que le dan sentido al sistema. Es como la cocina de una casa, donde se decide qué se va a preparar y cómo se va a servir.

Capa de acceso a datos

La capa de acceso a datos (*data access layer o data source*) se encarga de la interacción con el almacenamiento de información, que puede incluir bases de datos —como MySQL, SQL Server u Oracle—, archivos, servicios web u otras fuentes de datos. Esta capa realiza operaciones de lectura y escritura, y se encarga de mapear los datos entre la capa de lógica de negocio y la fuente de almacenamiento.

Si continuamos con la analogía, esta capa estaría en el sótano del edificio: es donde se almacena todo. Equivale a la

despensa o al almacén, donde se guarda la comida y las bebidas antes de ser preparadas y servidas en los niveles superiores.

Capa de entidades

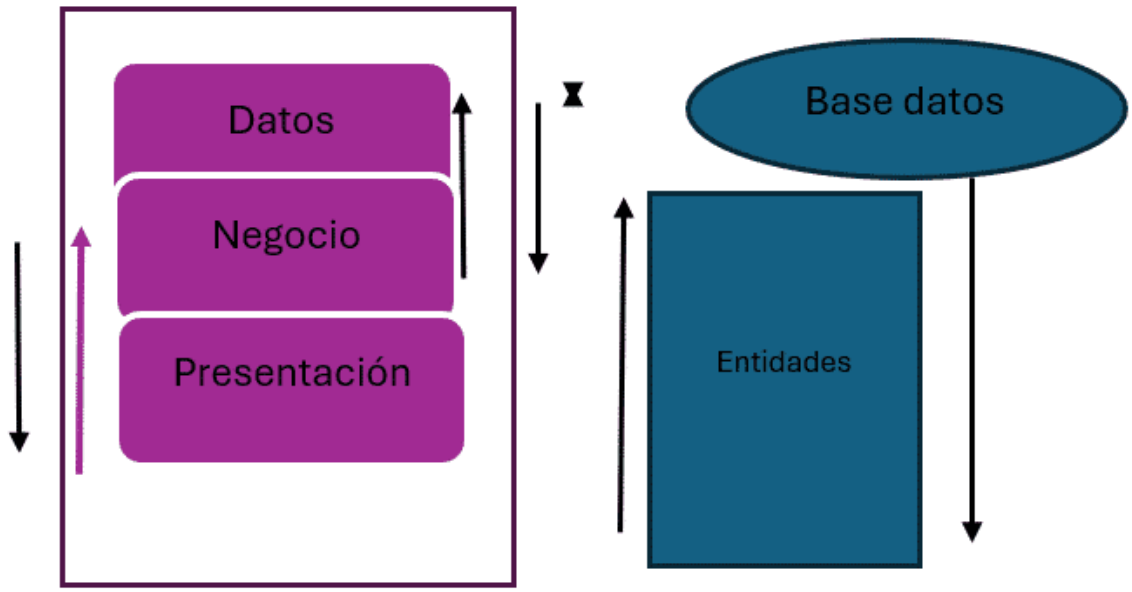
En algunas arquitecturas por capas, se incluye una capa de entidades que representa los objetos de datos utilizados en la aplicación. Estas entidades suelen reflejar la estructura de los datos almacenados y se emplean para transferir información entre la capa de acceso a datos y la capa de lógica de negocio.

Esta capa está estrechamente relacionada con las tablas de la base de datos, ya que las entidades funcionan como la representación de esos datos dentro del programa. Por ejemplo, si se define una clase «vehículo», podría tener atributos como color, cantidad de puertas o número de ruedas. Estos atributos forman parte de las entidades que viajan entre las distintas capas de la aplicación.

Para ejemplificar lo mencionado, en la capa de presentación (imagen 4) se tendrían dos instancias: una instancia de la entidad, que contiene todos los atributos definidos en el programa, y una instancia de la lógica de negocio, que recibe ese objeto para ejecutar las acciones correspondientes en un

CRUD —acrónimo de las cuatro operaciones básicas en una base de datos: crear, leer, actualizar y eliminar (*create, read, update, delete*)—. Por ejemplo, en el caso del alta de un usuario, la lógica de negocio procesaría el objeto recibido desde la interfaz y lo enviaría a la capa de acceso a datos para ser registrado en la base de datos.

Figura 5. Diagrama de interacción entre capas, entidades y base de datos



Fuente: elaboración propia

El uso de este patrón aporta varias ventajas al desarrollo de aplicaciones. Entre los principales beneficios, se destacan los

siguientes:

- **Separación de responsabilidades. Cada capa tiene una función definida, lo que facilita el mantenimiento y la comprensión del código.**
- **Reutilización:** las capas pueden reutilizarse en distintas aplicaciones o proyectos.
- **Escalabilidad:** permite escalar una aplicación de forma más sencilla, agregando o ajustando capas específicas.
- **Mantenibilidad:** es posible modificar o mejorar una capa sin afectar a las demás, siempre que se respete la separación de responsabilidades.

Las aplicaciones profesionales adoptan este patrón porque requieren un alto grado de modularidad y organización. La cantidad de capas a utilizar dependerá de las necesidades específicas del proyecto.

Arquitectura en capas

La arquitectura en capas consiste en dividir una aplicación en niveles, con el objetivo de que cada capa cumpla un rol claramente definido. Por ejemplo, se puede contar con una capa de presentación (UI), una capa de reglas de negocio (servicios) y una capa de acceso a datos (DAO). Sin embargo, este estilo arquitectónico no establece un número fijo de capas, sino que se basa en el principio de separación de responsabilidades (*separation of concerns, SoC*).

En la práctica, este enfoque suele implementarse en cuatro capas: presentación, negocio, persistencia y base de datos. No obstante, es común que las capas de negocio y persistencia se combinen en una sola, especialmente cuando la lógica de persistencia está incrustada dentro de la capa de negocio.

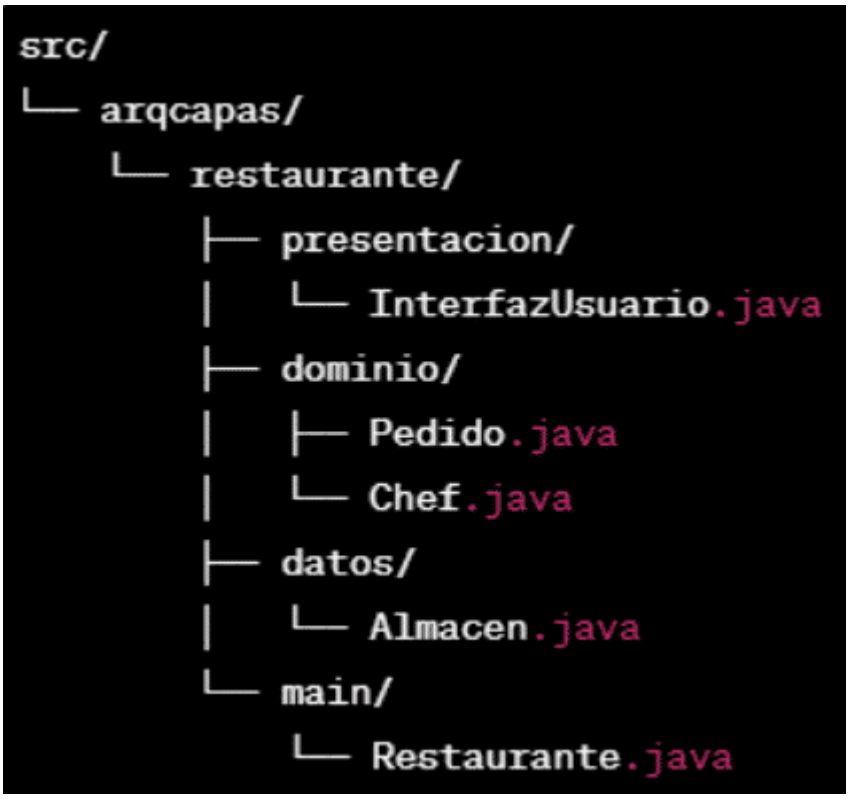
Ejemplo

Supongamos que se desea desarrollar un sistema para gestionar pedidos en un restaurante. El sistema debe permitir que los clientes realicen pedidos de comidas y que estos sean procesados por el personal de cocina.

Solución

Se propone una solución sencilla basada en una arquitectura de tres capas, organizada de la siguiente manera:

Figura 6. Carpetas del proyecto del ejemplo



Fuente: elaboración propia

presentación/ —

contiene las clases relacionadas con la interacción del usuario, como

la clase «InterfazUsuario».

dominio/ —

incluye las clases que representan la lógica de negocio, como «Pedido» y «Chef».

datos/ —

maneja el acceso y la manipulación de los datos, representado por la clase «Almacen».

main/ —

contiene la clase «Restaurante», que funciona como punto de entrada principal del programa. Aunque esta capa no forma parte de la arquitectura por capas tradicional, puede considerarse un componente encargado de iniciar y coordinar el sistema. Se trata de una clase de configuración, a veces denominada clase sucia, ya que conoce todos los paquetes.

En código Java, sería así:

Presentación

InterfazUsuario.java – Maneja la interacción con el usuario.

```
public class InterfazUsuario {

    private Chef chef;



    public InterfazUsuario(Chef chef) {

        this.chef = chef;

    }



    public void realizarPedido(String item) {

        Pedido pedido = new Pedido(item);

        System.out.println(pedido.describirPedido());

        String resultado = chef.prepararComida(pedido);

        System.out.println(resultado);
```

```
    }

}
```

Dominio

Pedido.java – Representa el pedido del cliente.

```
public class Pedido {

    private String item;



    public Pedido(String item) {

        this.item = item;

    }



    public String getItem() {

        return item;
```

```
    }


    public String describirPedido() {

        return "Pedido recibido para: " + item;

    }

}
```

Chef.java – Se encarga de la lógica de negocio, como preparar la comida.

```
public class Chef {

    private Almacen almacen;


    public Chef(Almacen almacen) {

        this.almacen = almacen;

    }
```

```
    public String prepararComida(Pedido pedido) {
                                        String ingredientes =
almacen.obtenerIngredientes(pedido.getItem());
        return "Comida preparada con " + ingredientes + " para
" + pedido.getItem();
    }
}
```

Datos

Almacen.java – Maneja el acceso a datos, por ejemplo, para obtener ingredientes

```
public class Almacen {
    public String obtenerIngredientes(String item) {
        // Simulación de la obtención de ingredientes
        return "tomate, queso, masa";
```

```
    }

}
```

Main

Restaurante.java – Clase principal que inicia y coordina el sistema.

```
public class Restaurante {

    public static void main(String[] args) {

        Almacen almacen = new Almacen();

        Chef chef = new Chef(almacen);

        InterfazUsuario interfaz = new InterfazUsuario(chef);

        interfaz.realizarPedido("pizza");

    }

}
```

Cada clase está asignada a una de las cuatro capas definidas: presentación, dominio, datos y main.

En resumen, la arquitectura de software en capas puede compararse con un edificio bien organizado o una fiesta bien planificada, donde cada componente tiene su lugar y propósito, y todos trabajan en conjunto para ofrecer una experiencia funcional y coherente.

Arquitectura separación de funciones en Java

En Java, la separación de funciones es fundamental para desarrollar aplicaciones modulares y mantenibles. Esto se logra mediante diversos mecanismos y patrones arquitectónicos que dividen el código en unidades lógicas con responsabilidades específicas.

Entre las técnicas más utilizadas se encuentran la arquitectura en capas, la separación de responsabilidades y patrones de diseño como MVC.

Arquitectura en capas

Este patrón organiza la aplicación en capas horizontales, cada una con una función definida y bien delimitada. Las capas más comunes son: presentación (encargada de la

interfaz de usuario), lógica de negocio (donde se procesan las reglas del sistema) y acceso a datos (que se comunica con las fuentes de información como bases de datos). Cada capa solo interactúa con las capas contiguas, lo que mantiene una separación clara de responsabilidades.

Este enfoque facilita el mantenimiento, la escalabilidad y las pruebas de la aplicación, ya que permite modificar o ampliar una capa sin afectar directamente a las demás.

Separación de responsabilidades (SoC)

La separación de responsabilidades es un principio fundamental en la arquitectura de software. Consiste en dividir el código en módulos o clases que se encargan de aspectos específicos de la aplicación. Esto permite, por ejemplo, que la lógica de la interfaz de usuario no se mezcle con la lógica de negocio, o que esta última se divida en componentes más pequeños y especializados.

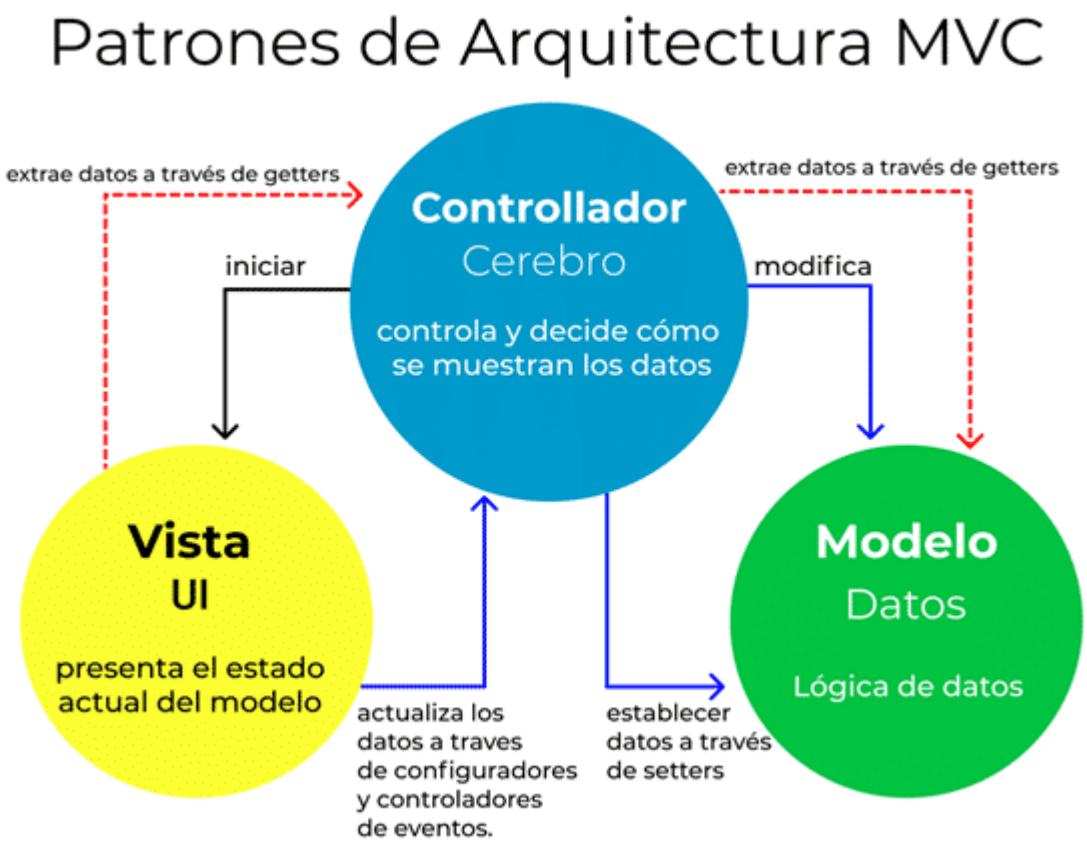
Una implementación concreta de este principio es la arquitectura modelo–vista–controlador (MVC), donde el modelo gestiona los datos y la lógica, la vista se encarga de mostrar la información al usuario, y el controlador coordina la interacción entre ambos. Este patrón ayuda a organizar el código de forma clara y facilita su mantenimiento.

Patrones de diseño

Java dispone de numerosos patrones de diseño que promueven la separación de funciones y la modularidad. Uno de los más conocidos es MVC, ideal para aplicaciones con interfaces de usuario complejas, ya que permite desarrollar y mantener de forma independiente la presentación, los datos y la lógica de interacción. Estos patrones no solo organizan el código, sino que también

ofrecen soluciones reutilizables a problemas comunes en el desarrollo de software.

Figura 7. Patrones de arquitectura MCV



Fuente: Hernández, s.f., https://short.do/6v8VW7

Existen otros patrones de diseño que también favorecen la modularidad y la separación de responsabilidades en Java. Algunos ejemplos son los siguientes:

1. **Factory:** permite crear objetos sin necesidad de especificar la clase concreta que se va a instanciar. Esto desacopla la creación de objetos de su uso, lo que facilita la extensión y el mantenimiento del código.

2. **Observer:** permite que múltiples objetos reciban notificaciones cuando un objeto observado cambia su estado. Este patrón facilita la gestión de dependencias, al separar el emisor de los receptores de eventos.

Microservicios

En el desarrollo de aplicaciones complejas, la arquitectura de microservicios propone dividir el sistema en servicios independientes, cada uno encargado de una funcionalidad específica. Estos servicios se comunican entre sí a través de API, y cada uno cuenta con su propia base de datos.

Esta arquitectura permite escalar y desplegar cada servicio de manera independiente, lo que aporta mayor flexibilidad y

agilidad al desarrollo. Sin embargo, también implica una mayor complejidad en la gestión de la comunicación, la seguridad y la consistencia entre servicios.

Ventajas de la separación de funciones

La separación de funciones en Java aporta numerosas ventajas al desarrollo de software. En primer lugar, mejora la **mantenibilidad**, ya que el código está dividido en módulos con responsabilidades claras, lo que facilita su comprensión y modificación. Además, permite **escalar partes específicas de la aplicación de forma independiente,** lo cual es especialmente útil en sistemas que crecen con el tiempo. La reutilización también se ve favorecida, ya que las funciones bien definidas pueden utilizarse en diferentes partes del sistema o incluso en otros proyectos.

Por otro lado, facilita la **realización de pruebas,** tanto a nivel de unidades individuales como en la integración entre componentes. También promueve la **independencia entre módulos**, reduciendo el impacto que puede tener un cambio en una parte del sistema sobre el resto.

En resumen, la separación de funciones en Java es un concepto clave para construir aplicaciones robustas, mantenibles y escalables. A través de técnicas como la arquitectura en capas, la separación de responsabilidades y el uso de patrones de diseño, se logra una organización del código que favorece tanto el desarrollo como la evolución de las aplicaciones.

MVC (modelo-vista-controlador)

Al comenzar a desarrollar aplicaciones en Java, es común que el código se vuelva difícil de mantener: lógica de negocio entremezclada con la interfaz de usuario, consultas a la base de datos insertadas en archivos HTML, y controladores que asumen más responsabilidades de las que deberían. En este contexto, los patrones arquitectónicos cumplen un rol fundamental para organizar y estructurar el desarrollo. Uno de los más populares y consolidados es MVC (modelo–vista–controlador).

Sin embargo, una aplicación bien estructurada va más allá de MVC. Dos patrones complementarios que suelen utilizarse junto a él son DAO (objeto de acceso a datos) y DTO (objeto de transferencia de datos). En conjunto, estos

conceptos permiten escribir código modular, mantenible y fácil de probar.

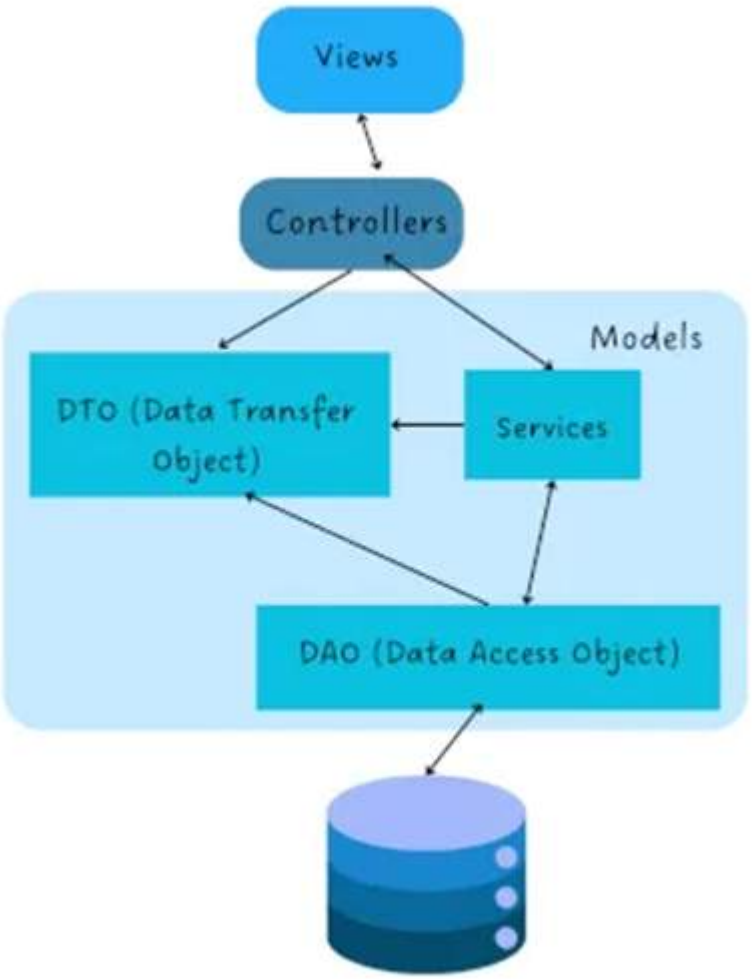
MVC divide la aplicación en tres componentes principales:

- **Modelo.** Gestiona los datos y la lógica de negocio.
- **Vista:** representa la interfaz de usuario con la que interactúan los usuarios.
- **Controlador:** actúa como intermediario, procesando las entradas del usuario y coordinando la comunicación entre el modelo y la vista.

Esta separación contribuye a mantener el código organizado y facilita la administración, las pruebas y el escalado de la aplicación.

La capa del modelo es el eje central de la estructura: representa los datos, aplica las reglas del negocio y gestiona la interacción con la base de datos, lo que la convierte en la columna vertebral de toda la lógica de la aplicación.

Figura 8. Modelo MCV



Fuente: Liyanage, 2025, https://short.do/UPE939

DAO (*data access object*)

La arquitectura en capas y el uso de componentes como controlador, servicio, DAO y DTO en Java permiten establecer una clara separación de responsabilidades. Esta organización

mejora el mantenimiento, la escalabilidad y las pruebas de la aplicación, ya que cada capa puede desarrollarse, modificarse y probarse de forma independiente, siempre que se respeten las interfaces y los contratos definidos entre ellas.

Como se observa en la figura 8, los componentes principales cumplen las siguientes funciones:

- Controlador. Gestiona la interacción con el usuario y la presentación de los resultados.
- Servicio: implementa la lógica de negocio y las reglas del dominio.
- DAO (Data Access Object): se encarga del acceso y la manipulación de los datos persistentes.
- DTO (Data Transfer Object): permite una transferencia eficiente y segura de datos entre capas.

A continuación, se describe el funcionamiento de cada uno de estos componentes.

Controlador

En una arquitectura por capas, el controlador cumple un rol clave en la interacción entre el usuario y la lógica interna de la aplicación. Su función principal es recibir las solicitudes del cliente, ya sea desde una interfaz de usuario o a través de una API REST. Una vez recibida la solicitud, el controlador coordina la comunicación entre las distintas capas del sistema para procesar la información y generar una respuesta adecuada.

Además, selecciona la vista o el recurso correspondiente que debe devolverse al cliente, asegurando así una experiencia de usuario coherente. En el contexto del patrón modelo–vista–controlador (MVC), el controlador actúa como intermediario entre el modelo, que gestiona los datos y la lógica de negocio, y la vista, que representa la interfaz del usuario.

En una aplicación Java basada en arquitectura en capas, cada componente tiene una responsabilidad bien definida. El controlador maneja las solicitudes del cliente y determina qué vista mostrar; el servicio contiene la lógica empresarial; el DAO se encarga del acceso y la manipulación de los datos; y los DTO permiten transferir datos de manera eficiente entre capas. Esta separación favorece la organización del

código, facilita el mantenimiento y permite escalar la aplicación con mayor facilidad.

Servicio

El servicio contiene la lógica de negocio de la aplicación. Se encarga de realizar operaciones que pueden requerir varios pasos o interacciones con otras capas. Recibe datos desde el controlador, los procesa, consulta al DAO cuando es necesario y devuelve los resultados al controlador.

DAO (*data access object*)

El DAO abstrae el acceso a los datos, ocultando los detalles de la persistencia, como el manejo de bases de datos o archivos. Proporciona una interfaz clara para acceder y manipular la información, de modo que ni el servicio ni el controlador necesitan conocer cómo se implementa ese acceso. Suele encargarse de operaciones básicas como crear, leer, actualizar y eliminar datos (CRUD) para una entidad específica.

DTO (*data transfer object*)

En español, «DTO» se refiere a objeto de transferencia de datos (Data Transfer Object). Un DTO es un objeto simple

que se utiliza para transferir datos entre diferentes capas de una aplicación, especialmente entre el cliente y el servidor o entre distintos servicios. Su función principal es transportar datos, sin incluir lógica de negocio, y se utiliza para reducir el número de llamadas y optimizar el rendimiento.

Aquí hay algunos puntos clave sobre los DTO:

- **Propósito.** Transferir datos entre diferentes partes de una aplicación, como entre el cliente y el servidor, o entre las distintas capas del sistema.
- **Características:** son objetos simples que no contienen lógica de negocio; su única función es transportar datos.
- **Beneficios**
  - **Reducir llamadas:** evitan múltiples llamadas a la base de datos o a servicios externos al agrupar los datos en un solo objeto.
  - **Optimizar rendimiento:** al reducir la cantidad de datos enviados, se mejora la eficiencia de la aplicación.

- **Independencia de la capa:** permiten desacoplar las diferentes capas de la aplicación, haciendo que los cambios en una capa no afecten a las demás.

- **Uso común:** se utilizan en aplicaciones web, servicios REST y sistemas distribuidos.

Ejemplo

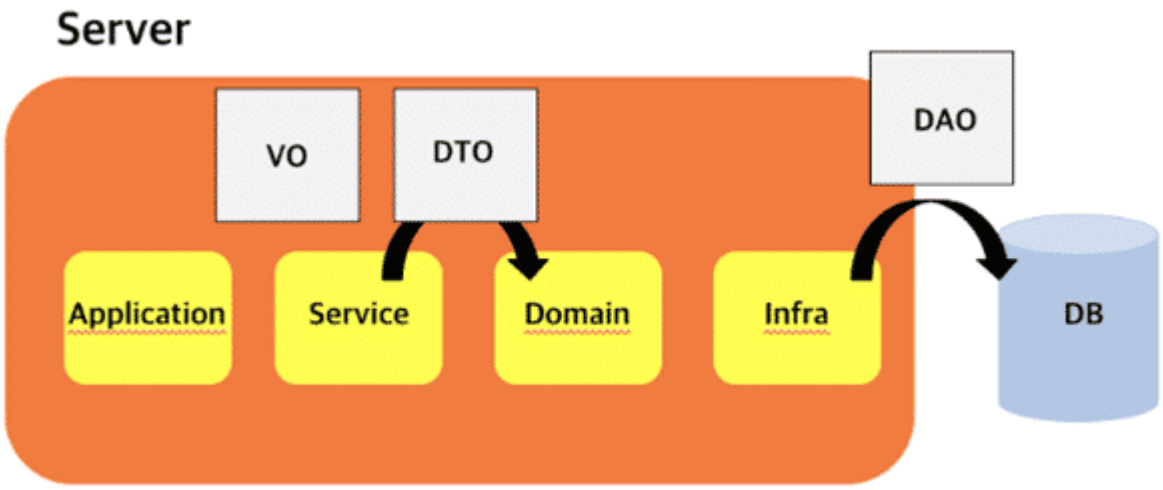
En una aplicación web, un DTO puede utilizarse para enviar la información de un usuario desde el servidor al cliente, incluyendo solo los datos necesarios para mostrar en la página, como el nombre y el correo electrónico, y excluyendo información sensible como contraseñas.

El DTO es un objeto plano (POJO, plain old java object) que se emplea para transportar datos entre capas. No contiene lógica de negocio, solo atributos que representan la información a transferir. Además, permite encapsular datos provenientes de diferentes fuentes y presentarlos de forma homogénea. Su uso ayuda a evitar el envío de datos innecesarios entre capas y facilita la gestión de la información.

Estructura de DAO

El data *access object* (DAO) se utiliza para crear una capa de persistencia que encapsula el acceso a la base de datos. De este modo, cuando la capa de lógica de negocio necesita interactuar con los datos, lo hace a través de la API que proporciona el DAO. Esta API generalmente incluye métodos CRUD (*create, read, update* y *delete*), que permiten realizar las operaciones básicas sobre las entidades del sistema.

Figura 9. Estructura de DAO



Fuente: Hermozo, 2024, https://short.do/ZIPV-z

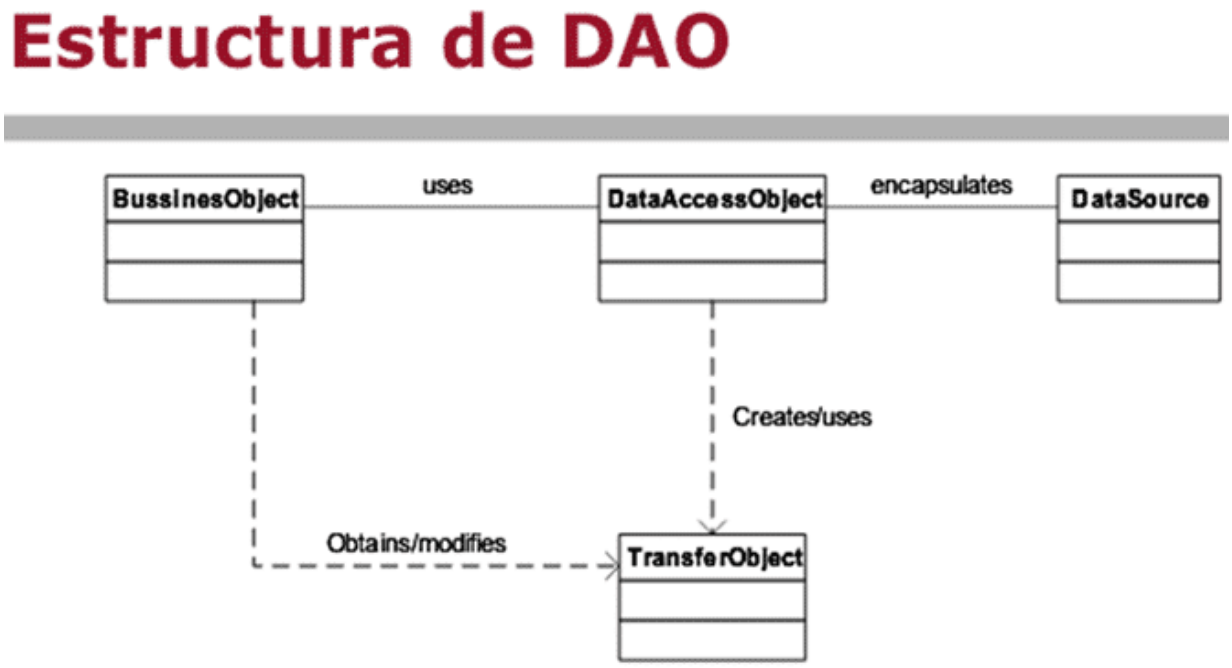
Entonces, por ejemplo, cuando la capa de lógica de negocio necesita guardar un dato en la base de datos, llama a un método create(). Lo que haga este método es responsabilidad del DAO y dependerá de cómo se

implemente. Puede que almacene los datos en una base de datos relacional, o que lo haga en archivos de texto. Lo importante es que la capa de lógica de negocio no necesita conocer esos detalles: solo sabe que el método *create*() guarda los datos, que el método *delete*() los elimina, que *update*() los actualiza, y así sucesivamente. El modo en que el DAO interactúa con la base de datos es completamente transparente para las demás capas.

Los DTO (*data transfer object*), también denominados VO (*value object*), son utilizados por el DAO para transportar datos entre la base de datos y la capa de lógica de negocio, y viceversa. Por ejemplo, cuando la lógica de negocio llama al método *create*(), el DAO inserta un nuevo dato. ¿Qué dato? El que la lógica de negocio le pasa como parámetro. ¿Y cómo se le pasa ese dato? A través de un DTO.

DAO oculta completamente los detalles del origen de datos de la aplicación hacia sus clientes. Como la interfaz que expone el DAO no cambia cuando la implementación del acceso a datos se modifica, este modelo permite adaptar el sistema a diferentes formas de almacenamiento sin afectar a los componentes de negocio. En esencia, DAO actúa como un adaptador entre el componente de negocio y la fuente de datos.

Figura 10. Relación entre DAO, capa de negocio y objetos de transferencia



Fuente: [imagen sin título sobre estructura de DAO], (s.f.), https://short.do/nGWlfH

- ***Business object:*** es el objeto que desea acceder a la fuente de datos para almacenar o consultar información.

- ***Data access object:*** abstrae al business object de los detalles del acceso a la fuente de datos.

- ***Data source:*** representa la implementación concreta de la fuente de datos.

- ***Transfer object:*** actúa como objeto intermedio entre el business object y el data access object.

En resumen, el patrón de diseño *data access object* (DAO) es una forma de reducir el acoplamiento entre la lógica de negocio y la lógica de persistencia. La lógica de negocio suele trabajar con objetos de dominio que deben almacenarse en una base de datos, un sistema de archivos u otro medio de persistencia. El patrón DAO permite encapsular el código necesario para realizar operaciones CRUD contra esa fuente de datos, manteniéndolo separado del resto de la aplicación. Esto significa que cualquier cambio en la lógica de persistencia no afecta a otras capas, lo que incrementa la estabilidad del sistema.

Además, este patrón resuelve un problema clásico en el desarrollo de aplicaciones, como las aplicaciones web, donde en una primera etapa se puede elegir una fuente de datos simple, como archivos de texto, y luego migrar a una base de datos como MySQL. Sin un DAO, ese cambio requeriría

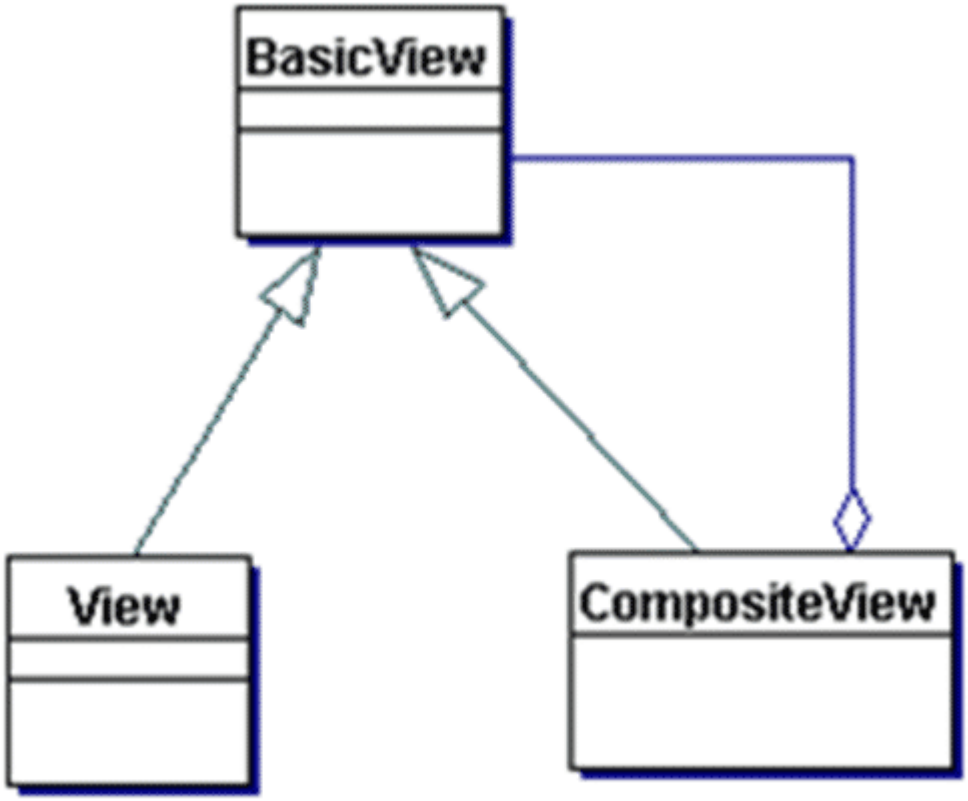
rehacer gran parte del código; con DAO, la modificación se limita al componente de acceso a datos.

Composite view (vista compuesta)

El patrón composite permite componer objetos dentro de estructuras de árbol para representar jerarquías parte–todo. Este enfoque posibilita que los clientes traten de forma uniforme tanto a objetos individuales como a composiciones de objetos.

En aplicaciones gráficas —como editores de dibujo o herramientas CASE— este patrón facilita que los usuarios construyan diagramas complejos a partir de componentes simples. Los elementos pueden agruparse para formar componentes más grandes, y estos, a su vez, combinarse para crear estructuras aún más complejas. Una implementación básica podría incluir clases para primitivas gráficas, como «Texto» o «Líneas», y otras clases contenedoras que agrupen estas primitivas para formar elementos compuestos.

Figura 11. Composite view (vista compuesta)



Fuente: [imagen sin título sobre composite view], (s.f.), https://short.do/KFRigO

Un problema habitual con este enfoque es que el código que utiliza estas clases debe tratar de forma distinta a los objetos primitivos y a los contenedores, aunque el usuario los manipule de manera uniforme. Esta necesidad de diferenciación incrementa la complejidad de la aplicación. El patrón composite resuelve este inconveniente mediante composición recursiva, permitiendo que los clientes no tengan que distinguir entre objetos individuales y composiciones.

Se recomienda usar el patrón composite cuando:

- se desea representar jerarquías de objetos parte–todo;
- se necesita que los clientes ignoren la diferencia entre objetos individuales y composiciones, y puedan tratar todos los elementos de forma uniforme dentro de la estructura.

Estructura

El patrón composite pertenece a la categoría de patrones con propósito estructural y alcance de objeto. Propone una forma de componer a sus participantes —en este caso, objetos— mediante relaciones que se determinan dinámicamente en tiempo de ejecución.

Este patrón define jerarquías de clases compuestas por objetos primitivos y objetos compuestos. A partir de los objetos primitivos se pueden construir, por composición, objetos más complejos, que a su vez pueden combinarse con otros en una estructura recursiva. Siempre que el código del cliente espere un objeto primitivo, podrá recibir también un objeto compuesto.

Una de las principales ventajas del patrón composite es que simplifica el desarrollo del cliente. Este puede tratar de forma uniforme tanto estructuras compuestas como objetos individuales, sin necesidad de saber si está interactuando con una hoja o con un componente compuesto. Esto evita la necesidad de programar funciones específicas para cada tipo de clase en el árbol, como estructuras con sentencias switch-case, reduciendo la complejidad del código.

Además, facilita la incorporación de nuevos tipos de componentes. Las últimas subclases de hoja o compuesto que se definan funcionarán automáticamente con las estructuras existentes y el código del cliente, sin necesidad de modificaciones.

Como desventaja, este enfoque puede hacer que el diseño sea demasiado general. Si se desea restringir qué tipos de componentes puede contener un compuesto, el sistema de tipos por sí solo no es suficiente para imponer esas limitaciones. En esos casos, será necesario aplicar validaciones adicionales en tiempo de ejecución.

Implementación:

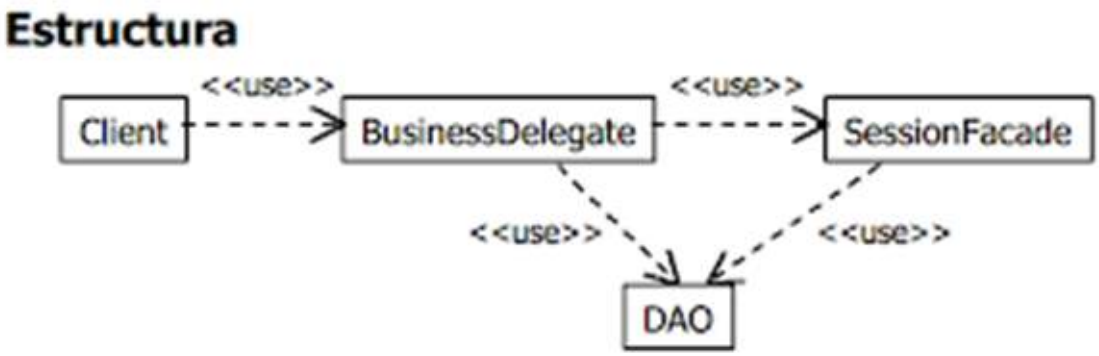
Una *composite view* puede implementarse siguiendo la estrategia JSP page view o bien la estrategia Servlet view, según lo propuesto por Alur.

El control de la vista puede realizarse de distintas maneras: mediante etiquetas estándar de JSP, como <jsp:include>; utilizando componentes JavaBeans; o mediante etiquetas personalizadas definidas en JSP.

Fast-lane reader (lector de carril rápido)

Este patrón propone un mecanismo para obtener grandes cantidades de información mediante un acceso directo a la base de datos.

Figura 12. Patrón *fast – lane reader*



Fuente: [imagen sin título sobre patrón fast – lane reader], (s.f.), https://short.do/t4_gW4

Implementación

El patrón *fast-lane reader* se aplica en casos de uso que implican búsquedas que devuelven una colección de objetos. Es especialmente útil cuando se realizan múltiples consultas y la eficiencia en el acceso a los datos es un factor relevante.

Además, existen otros patrones relevantes, como el *front controller* (controlador frontal), que centraliza la gestión de las solicitudes en aplicaciones web. También se pueden consultar diagramas de secuencia relacionados con este patrón para comprender mejor su funcionamiento.

Se sugiere investigar estos y otros patrones adicionales para ampliar el conocimiento sobre arquitecturas de software y su aplicación en distintos contextos.

DTO (*data transfer object*)

Un DTO es un objeto Java simple que utilizamos para transferir datos entre distintas partes de una aplicación, especialmente entre las capas de servicio y vista. Es un objeto ligero, sin lógica de negocio, que solo incluye campos, constructores, y los métodos *getters* y *setters*.

Podemos pensar en un DTO como una versión «desinfectada» de nuestros datos: expone únicamente la información necesaria para cumplir su función.

Figura 13. Ejemplo de implementación de un DTO (data transfer object) en Java

```java
public class UserDTO {
    private String username;
    private String email;

    public UserDTO(String username, String email) {
        this.username = username;
        this.email = email;
    }

    public String getUsername() { return username; }
    public void setUsername(String username) { this.username = username; }

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
}
```

DAO (*data access object*)

El DAO se encarga de gestionar todas las operaciones sobre la base de datos. Nos permite abstraer la lógica de interacción con la base de datos, de modo que el resto de la aplicación no tenga que preocuparse por las consultas SQL.

Esta clase se comunica directamente con la base de datos y devuelve DTO correctamente empaquetados, listos para ser utilizados por las capas superiores.

Figura 14. Ejemplo de implementación de un DAO (data access object) en Java

```java
public class UserDAO {
    private Connection connection;

    public UserDAO(Connection connection) {
        this.connection = connection;
    }

    public UserDTO getUserById(int id) throws SQLException {
        String query = "SELECT username, email FROM users WHERE id=?";
        PreparedStatement statement = connection.prepareStatement(query);
        statement.setInt(1, id);
        ResultSet rs = statement.executeQuery();

        if (rs.next()) {
            return new UserDTO(rs.getString("username"), rs.getString("email"));
        }
        return null;
    }
}
```

Fuente: elaboración propia

Service layer (capa de servicio)

La capa de servicio actúa como puente entre el DAO y el controlador. Contiene la lógica de negocio de la aplicación y, con frecuencia, convierte las entidades de la base de datos en DTO antes de enviarlas al controlador.

Gracias a esta capa, el controlador se mantiene limpio y enfocado únicamente en el manejo de la entrada y salida del usuario, sin preocuparse por la lógica de negocio ni por la transformación de datos.

Figura 15. Ejemplo de implementación de service layer en Java

```java
public class UserService {
    private UserDAO userDAO;

    public UserService(UserDAO userDAO) {
        this.userDAO = userDAO;
    }

    public UserDTO getUser(int id) throws SQLException {
        return userDAO.getUserById(id);
    }
}
```

Fuente: elaboración propia

Vista: lo que ve el usuario

La vista se centra en la presentación. Ya sea que utilicemos JSP (JavaServer Pages), *Thymeleaf* o un *frontend* moderno como *React*, esta capa se encarga de mostrar los datos al usuario de forma clara y coherente.

En la figura 16 se observa un ejemplo de vista implementada con JSP. La vista obtiene el *UserDTO* desde el controlador y lo utiliza para completar la interfaz de usuario, mostrando únicamente la información necesaria de manera organizada.

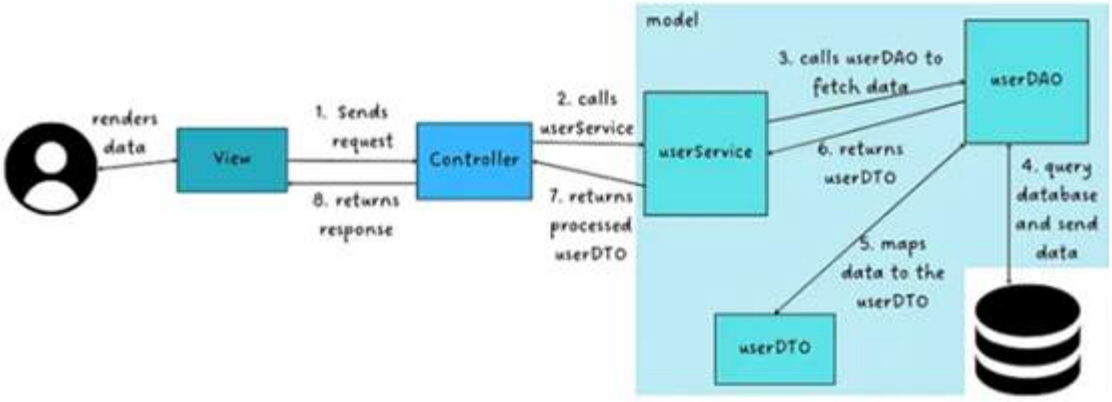
Figura 16. Ejemplo de implementación de vista con JPS

```jsp
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<html>
<head><title>User Profile</title></head>
<body>
    <h2>User Details</h2>
    <p>Username: ${user.username}</p>
    <p>Email: ${user.email}</p>
</body>
</html>
```

Fuente: elaboración propia

¿Cómo funcionan conjuntamente DAO y DTO en una aplicación?

Figura 17. Flujo MVC con integración de DAO y DTO



MVC Workflow with DAO and DTO Integration

Fuente: Liyanage, 2025, https://short.do/UPE939

Supongamos que un usuario accede a /user?id=1 en el navegador. Esto es lo que ocurre paso a paso:

1. El controlador recibe la solicitud y la gestiona, llamando a UserService.
2. El servicio (UserService) solicita a UserDAO obtener los datos del usuario desde la base de datos.
3. El DAO consulta la base de datos y devuelve un DTO con la información del usuario.
4. El servicio recibe el DTO y lo envía de vuelta al controlador.
5. El controlador adjunta el DTO a la solicitud y la reenvía a la vista.
6. La vista representa los datos del usuario en la página.

Cada componente tiene una función clara, lo que hace que el código sea más limpio, modular y fácil de depurar o modificar.

CONTINUAR

Referencias

Hermozo, D. (2024). *DTO y DAO en el Desarrollo de Software.* LinkedIn. https://www.linkedin.com/pulse/dto-y-dao-en-el-desarrollo-de-software-angel-hermozo-zemte?originalSubdomain=es

Hernández, R. (s.f.). *El patrón modelo-vista-controlador: Arquitectura y frameworks explicados.* https://www.freecodecamp.org/espanol/news/el-modelo-de-arquitectura-view-controller-pattern/

[Imagen sin título sobre composite view], (s.f.). https://anacsantosa.blogspot.com/2013/07/123-composite-view.html

[Imagen sin título sobre estructura de DAO], (s.f.). https://miprimerblog-temas.blogspot.com/2013/07/patrones-de-diseno.html

[Imagen sin título sobre patrón fast – lane reader], (s.f.). https://castellanosmiguel.blogspot.com/2013/07/fast-lane-reader-definicion-estepatron.html

Liyanage, D. (2025). Understanding MVC, DAO, and DTO: A Clean Way to Structure Your Application. https://medium.com/@dinithi.adithya/understanding-mvc-dao-and-dto-a-clean-way-to-structure-your-application-566437a0dbba

Oracle, (s.f.). Estructura arquitectónica de Java ES. https://docs.oracle.com/cd/E19528-01/820-3091/aauav/index.html

CONTINUAR

Lección 3 de 3

Descarga en PDF