Patrones de diseño de “software”

Breve descripción:

Los patrones de diseño de “software” surgen a partir de experiencias desarrolladas en la industria y representan un conjunto de formas estandarizadas, probadas y repetibles, que permiten resolver uno o varios problemas particulares que se presentan en el diseño del “software”.

Abril 2024

Tabla de contenido

Introducción

Le damos la bienvenida al componente formativo “Patrones de diseño de software”. Recordemos que los patrones de diseño surgen como un concepto, inicialmente, para el área de arquitectura e ingeniería civil, pues era en estas disciplinas donde se presentaban casos particulares que requerían de soluciones a problemáticas particulares.

El concepto de patrón de diseño se empezó a asociar al diseño orientado a objetos y, en lugar de referirse a la forma de colocar paredes, puertas o ventanas, se refería a la forma en que se construyen clases, objetos, interfaces, y a la forma en cómo estos deben interactuar.

El uso de patrones de diseño permite acelerar el proceso de desarrollo, al proporcionar paradigmas probados y comprobados en el diseño de “software”. Estandarizar las soluciones a problemas comunes permite también mejorar la comunicación entre desarrolladores, ya que se emplean nombres bien conocidos y entendidos a la hora de describir un problema de diseño y cómo abordar su solución.

A continuación, se abordan varios de los patrones de diseño más representativos en la industria del “software”, los cuales nos permitirán, a futuro, implementar soluciones más robustas, de acuerdo con altos estándares de calidad y recomendaciones ampliamente conocidas en la industria, lo que adicionalmente nos dará un plus como desarrolladores de “software” en una industria cada vez más competitiva globalmente.

1. Conceptos GOF

Los patrones GoF se presentan como una forma indispensable de abordar la programación y surgen debido a: Erich Gamma, Richard Helm, Ralph Jonson y John Vlissides, quienes tratan el tema en su libro "Design Patterns—Elements of Reusable Software"; por esta razón, a estos patrones se les conoce con el nombre de la pandilla de los cuatro ("GoF, gang of four"). Según estos autores, los patrones de diseño se basan principalmente en dos principios de diseño orientado a objetos:

- Programa una interfaz, no una implementación.
- Favorece la composición de objetos sobre la herencia.

Para EcuRed (s. f.), los patrones de diseño tienen las siguientes características:

- Constituyen soluciones concretas que, aunque se representan de forma genérica, son aplicados para resolver problemas reales.
- Son especificaciones técnicas basadas en los principios de la programación orientada a objetos y, dependiendo del lenguaje de programación, la forma de implementarlos puede variar.
- Son frecuentemente utilizados, ya que se construyen a partir de la experiencia acumulada en la industria del desarrollo de "software".
- Los patrones generalmente hacen referencia al uso de interfaces, clases y objetos que deben ser ajustados de acuerdo con la solución concreta a desarrollar.
- No necesariamente el uso de un patrón implica el uso de palabras claves, reservadas o de librerías especializadas.

- Favorecen la implementación de las características de la programación orientada a objetos, como la encapsulación, las jerarquías y el polimorfismo.

Los patrones que describen GOF suelen clasificarse en tres grandes categorías según su finalidad: patrones comportamentales, patrones creacionales y patrones estructurales. Una breve explicación de estos podrá consultarse a continuación, pues en el transcurso de este componente serán explicados con mayor detalle:

Patrones GoF

- **Creacionales:** abstraen el proceso de crear instancias y configuración de objetos.
- **Estructurales:** se ocupan de agrupar clases y objetos para formar estructuras más complejas. Separan la interfaz de la implementación.
- **Comportamentales:** describen la responsabilidad de los objetos y la forma en que se comunican entre ellos.

Para conocer cada uno de estos patrones GOF, lo invitamos a revisar su definición:

Tabla 1. Patrones GoF

Nombre	Definición
Fábrica abstracta (“Abstract factory”)	Crea diferentes familias de objetos relacionados o dependientes, como, por ejemplo, la creación de diferentes elementos de una interfaz gráfica.
Método de fabricación (“Factory method”)	Define una interfaz para crear un objeto, pero permite que las subclases determinen la clase a implementar.

Nombre	Definición
Prototipo (“Prototype”)	Se basa en la clonación de objetos copiándolos de un prototipo.
“Singleton”	Restringe una clase a una sola instancia y provee un punto global de acceso a la clase.
Constructor (“Builder”)	Separa la construcción de objetos complejos de su representación.
Adaptador (“Adapter”)	Convierte una interfaz de una clase en otra.
Puente (“Bridge”)	Desacopla una abstracción de su implementación, permitiendo modificarlas independientemente.
Objeto Compuesto (“Composite”)	Construye objetos complejos a partir de otros más simples, utilizando para ello la composición recursiva y una estructura de árbol.
Envoltorio (“Decorator”)	Permite añadir dinámicamente funcionalidad a un objeto existente, evitando heredar sucesivas clases para incorporar la nueva funcionalidad.
Fachada (“Façade”)	Permite una interfaz unificada para un conjunto de interfaces en un subsistema.
Peso Mosca (“Flyweight”)	Emplea un uso compartido para eliminar o reducir la redundancia cuando se tiene gran cantidad de objetos con información idéntica.
Apoderado (“Proxy”)	Proporciona un objeto sustituto para controlar el acceso a otro.
Cadena de responsabilidad (“Chain of responsibility”)	Permite que más de un objeto tenga la posibilidad de atender una petición.
Orden (“Command”)	Encapsula una petición como un objeto y permite “deshacer” la petición.
Intérprete (“Interpreter”)	Permite construir un intérprete de lenguaje para una gramática simple y sencilla.

Nombre	Definición
Iterador (“Iterator”)	Define una interfaz que declara los métodos necesarios para acceder secuencialmente a una colección de objetos sin exponer su estructura interna.
Mediador (“Mediator”)	Coordina las relaciones entre sus asociados. Permite la interacción de varios objetos, sin generar acoples fuertes en esas relaciones.
Recuerdo (“Memento”)	Almacena el estado interno de un objeto y lo restaura posteriormente.
Observador (“Observer”)	Dependencia de uno a muchos para notificar cambios de estado de un objeto.
Estado (“State”)	Permite a un objeto alterar su comportamiento dependiendo del estado interno del mismo.
Estrategia (“Strategy”)	Define familias de algoritmos, los encapsula y maneja su selección.
Método plantilla (“Template method”)	Define el esqueleto de un algoritmo y permite que clases derivadas redefinan ciertos pasos.
Visitante (“Visitor”)	Operaciones aplicadas a elementos de una estructura de objetos.

2. Patrones de arquitectura

Para hablar de patrones, primero es necesario hablar de arquitectura de "software". Esta consiste en una disciplina dentro del desarrollo de "software". Como ya se sabe, este desarrollo es algo intangible y puede ser tan complejo como las necesidades que se quieren resolver, por lo tanto, es importante tener en cuenta que, cuando se piensa en la escalabilidad del "software" y su usabilidad, utilizando los recursos disponibles y sacándole el máximo provecho a esta arquitectura, posibilitando un ciclo de vida mucho más largo, se debe pensar que una aplicación de "software" es una especie de lego, donde todas sus piezas se encajan, para poder cumplir con diversas funciones.

Dentro de estos patrones de arquitectura, se encuentran diferentes tipos, que permiten generar "software" con mayor rendimiento. Algunos de estos son:

2.1. Patrón multicapa

Son patrones de X niveles y se organizan en capas horizontales. Muchas de las aplicaciones informáticas han tomado este patrón como arquitectura, funciona conectando los componentes, pero estos no dependen uno del otro. Cada una de estas capas cumple con una función específica en el desarrollo de la aplicación. Su objetivo primordial es la separación de la lógica de negocios de la lógica de diseño, es decir, separar la capa de datos de la capa de presentación al usuario.

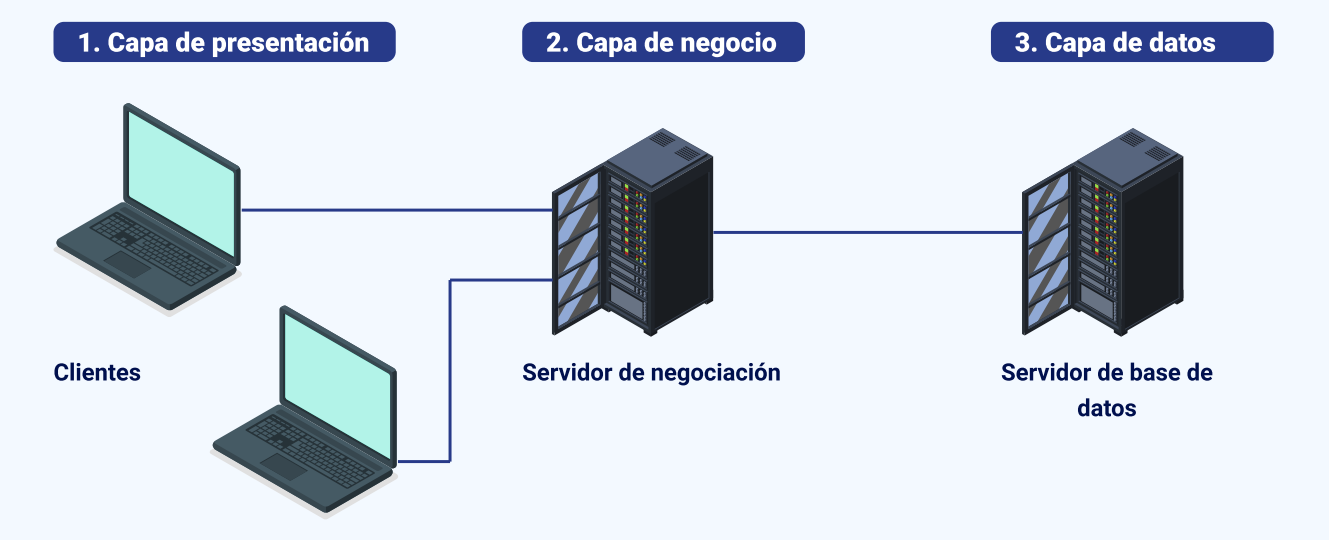
La ventaja que ofrece consiste en que el desarrollo se puede llevar a cabo en varios niveles. En caso de realizar algún cambio, solo se afecta el nivel requerido, sin tener que revisar todo el código.

Ejemplo:

El modelo de interconexión de sistemas abiertos, donde la capa de presentación se encarga de todo el diseño y desarrollo visual de parte del usuario, mientras que la capa de negocio se encarga de ejecutar las reglas y la lógica del negocio. Cada capa tiene funciones específicas, una capa informa cómo debe comportarse la venta de un producto, teniendo en cuenta la lógica y reglas establecidas para registrar esa venta, de esta forma, el patrón permite delegar funcionalidades propias de cada capa.

En la siguiente figura, se ilustra la arquitectura en tres capas. Fue creada en Microsoft Visio 2003, para ayudar a entender gráficamente y de una forma más fácil este tema.

Figura 1. Arquitectura de tres capas



Este patrón puede dividirse entre capas tantas veces se decida según su arquitectura, solo se debe tener en cuenta que, aunque es una buena forma de desarrollar, si existen muchas capas, también podría ser contraproducente, así que

depende del juicio del arquitecto de “software” la distribución de capas que contemple.

Ventajas

- Facilidad al momento de realizar pruebas.
- Facilidad para el mantenimiento de la aplicación.
- Se puede desarrollar por separado, delegando capas por desarrollador.
- Permite actualizar y mejorar la aplicación por separado.

2.2. Patrón Modelo Vista Controlador

Este modelo, también identificado como MVC, es una arquitectura que también se separa por capas, solo que en este caso son tres capas las que se dedican o son responsables de delegar funcionalidades específicas, como se describe a continuación:

Modelo: es una representación de los datos, maneja la lógica del negocio y la persistencia.

- Accede a la capa de la base de datos.
- En el modelo se definen las reglas del negocio.
- Contiene la información extraída de la base de datos para después ser manipulada.

Figura 2. Patrón Modelo Vista Controlador

Nota. Marrero, D. (2016). Patrón MVC.

Vista: se encarga de la interfaz de usuario, la interacción del usuario con el sistema y el despliegue de información.

- Muestra la información solicitada al modelo.
- Recibe los eventos que ejecuta del usuario en la interacción de la vista y se conecta con el controlador.
- Recibe información del usuario para ser enviada al modelo y que sea almacenada.

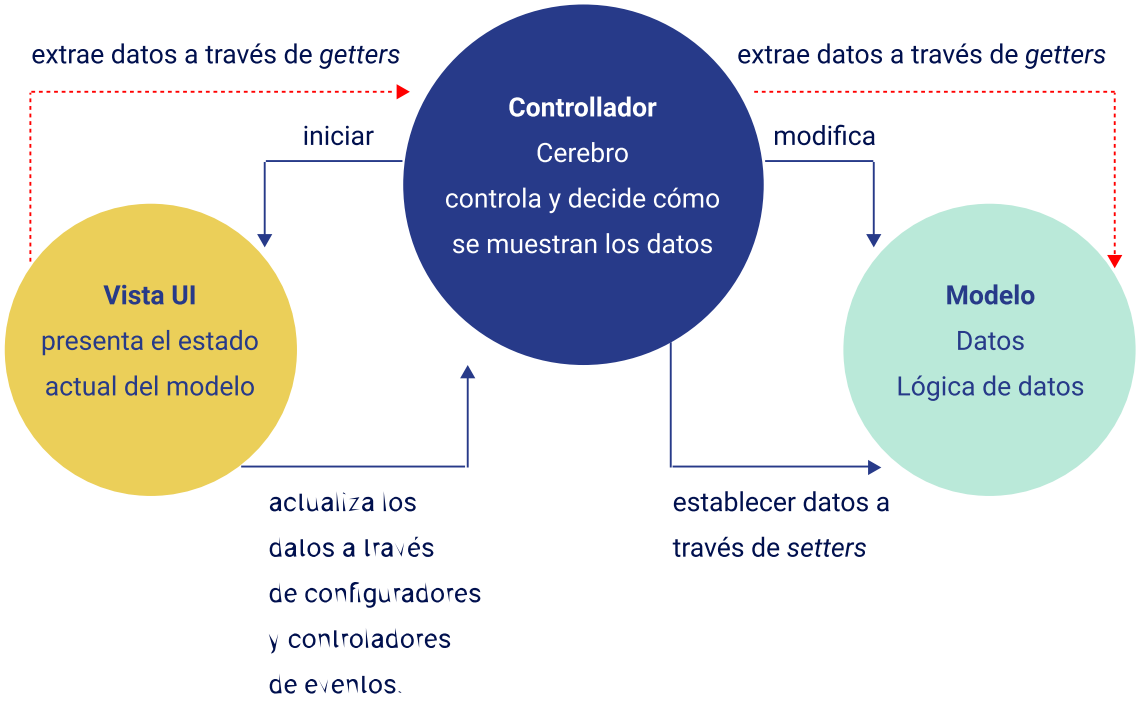
Figura 3. Esquema del patrón de diseño Modelo-Vista-Controlador (MVC)

Nota. Bahit, E. (2011). POO y MVC en PHP.

Controlador: el controlador no es más que el interlocutor entre la vista y el modelo, gestionando y adaptando la información entre ellas.

- Recibe peticiones desde la vista por medio de eventos.
- Envía peticiones al modelo de datos de información.
- Organiza la información para el despliegue en la vista.

Figura 4. Patrón Modelo-Vista-Controlador (MVC)



Nota. Hernández, R. (2021). El patrón modelo-vista-controlador: Arquitectura y "frameworks" explicados.

Es así como se describen algunas de las responsabilidades de cada capa, para lo que hay que seguir unos pasos específicos al momento de iniciar sesión en una aplicación:

a) El usuario ingresa los datos de acceso, "login" y "password", en la vista. Esta información se solicita al usuario dentro de dos cajas de texto.
b) El usuario da clic en el botón ingresar, siendo esta una petición mediante un evento desde la vista.

c) La vista envía esta petición mediante el evento al controlador y recibe los datos ingresados por el usuario, por medio de un arreglo.

d) El controlador envía esta información y manda una petición de consulta al modelo.

e) El modelo, que es una representación de los datos, verifica si el usuario existe y la contraseña es la correcta.

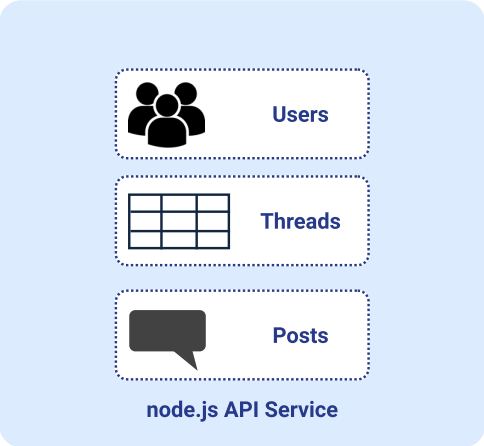
f) El modelo envía el resultado al controlador, que fue quien hizo la petición de consulta.

g) El controlador devuelve el resultado a la vista y, si concuerda, permite el ingreso al usuario; si no, le devuelve un mensaje de error.

2.3. Arquitectura monolítica

La arquitectura monolítica es aquella en la que el "software" está estructurado de forma tal que todos los aspectos funcionales del mismo quedan acoplados y sujetos en un mismo programa. En esta arquitectura, cada proceso o microservicio es un elemento independiente. Es utilizada en aplicaciones en las que todo se desarrolla en una misma parte, aunque también se utilizan módulos para desarrollar.

Figura 5. Monolith



En el momento de compilarse, lo hace de forma completa, como una sola, y se comprende en una interfaz del lado del cliente, una aplicación del lado del servidor y una base de datos. Al igual que las otras arquitecturas, tiene sus ventajas y desventajas:

Ventajas

- Fácil de probar y de “buggear”.
- Fácil de desplegar.
- Fácil de desarrollar.

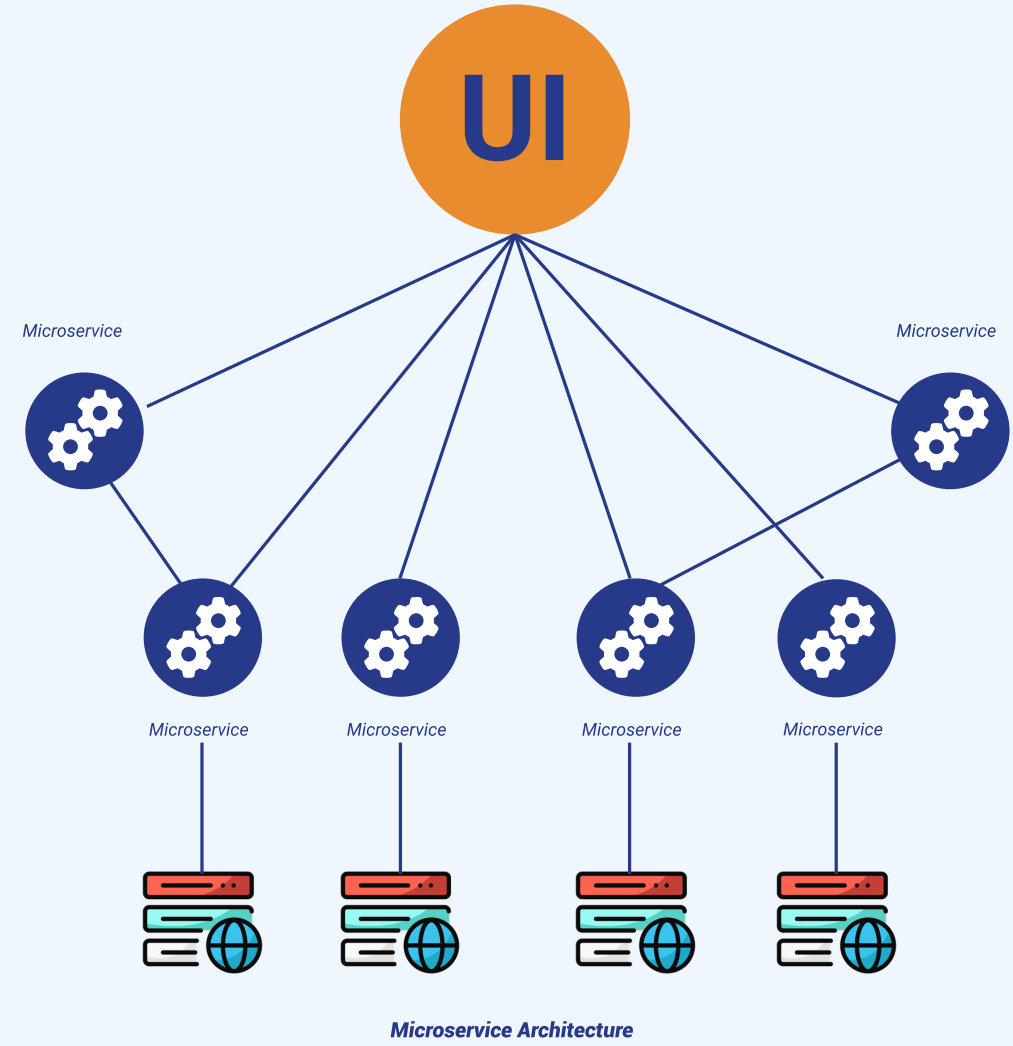
Desventajas

- Alto acoplamiento: si se hace un cambio en alguna parte del código, puede cambiar el comportamiento en otra parte del mismo.
- Código muy grande.
- Costos altos al momento de su escalabilidad.
- Difícil de mantener.
- Dificultad en las mejoras y actualizaciones.

2.4. Microservicios

Esta arquitectura se enfoca en crear pequeños programas API, que permiten que los servicios se comuniquen, posibilitando que las aplicaciones con estas arquitecturas puedan evolucionar y adaptarse muchísimo mejor. Además, se pueden incluir servicios de otras plataformas u otras aplicaciones, ya que pueden ser consumidos para la manipulación de la información.

Figura 6. Microservicios



Nota. Imagen tomada de Hiberus. (2021)

Dentro de sus características está que cada servicio que se desarrolla se puede hacer de forma independiente, sin que afecte el funcionamiento del resto de la aplicación; además, se crean para solucionar problemas específicos, haciéndolos

especializados, y, en caso de que un servicio de estos se vuelva muy grande, puede seguirse dividiendo en más microservicios.

Entre los beneficios de los microservicios, se tienen:

Los microservicios: en el mundo del desarrollo de "software", los microservicios están revolucionando la forma en que las empresas crean y despliegan sus aplicaciones. Hoy, exploraremos cómo esta arquitectura puede ser un catalizador de cambio, ofreciendo múltiples beneficios que potencian la agilidad, la escalabilidad y la eficiencia.

La agilidad: en el enfoque de microservicios, los desarrolladores se enfrentan a problemas más manejables, aplicando la estrategia de "divide y vencerás". Esto no solo facilita una mejor comprensión y solución de los problemas, sino que también permite que los equipos se dividan según los microservicios en los que trabajan, integrando posteriormente sus funcionalidades, permitiendo a los equipos avanzar más rápido y con mayor precisión.

La escalabilidad: con los microservicios, agregar nuevas funciones se convierte en una tarea sin mayores inconvenientes. Esta flexibilidad en el escalado permite a las organizaciones adaptarse rápidamente a las demandas cambiantes del mercado o a los nuevos requisitos de los usuarios, sin comprometer el rendimiento.

La sencillez en la implementación: al centrarse en problemas más pequeños, los equipos logran una mejor comprensión de los desafíos, lo que facilita entregas continuas y permite corregir errores de manera oportuna.

La libertad tecnológica: los microservicios operan como aplicaciones pequeñas e independientes, lo que significa que no están atados a una única arquitectura. Esta

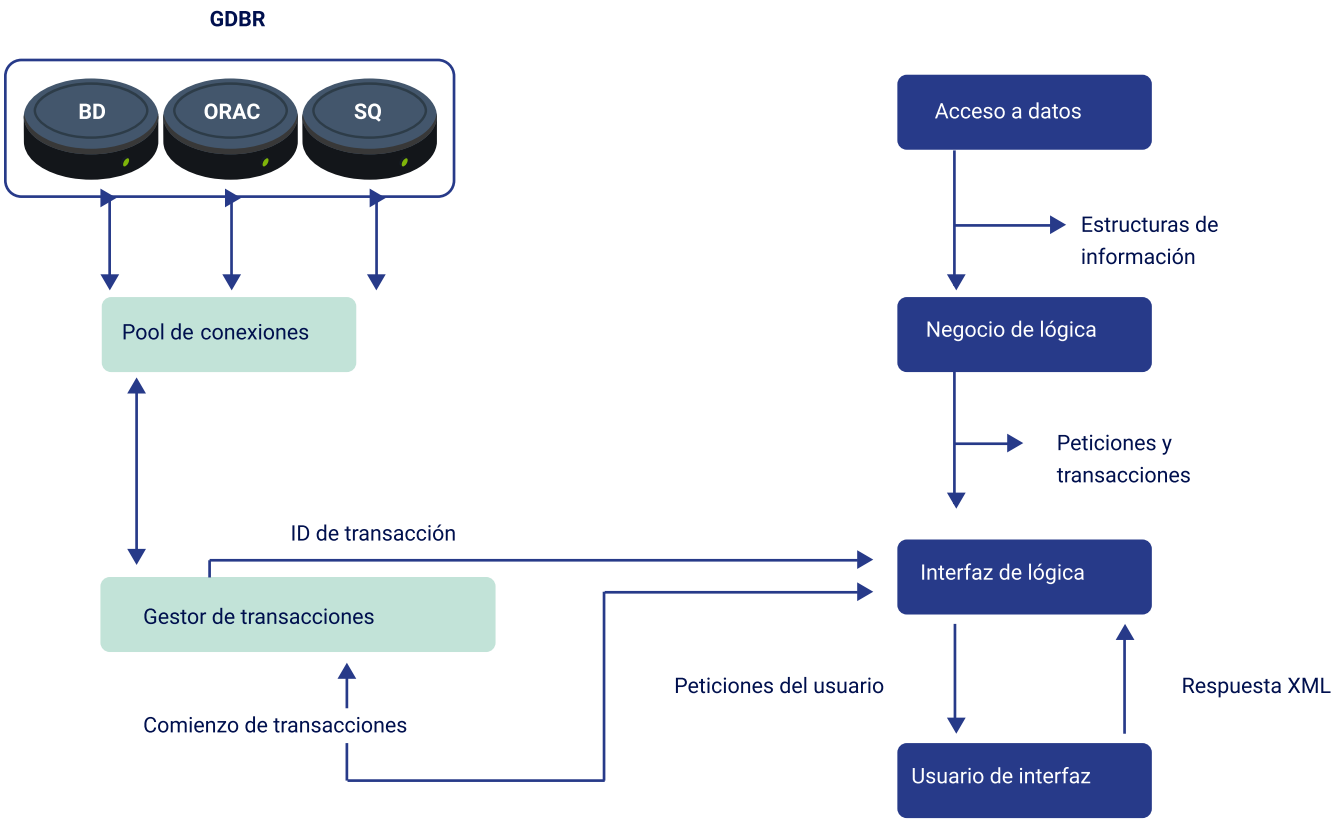
diversidad permite a los equipos utilizar las tecnologías más adecuadas para cada servicio, fomentando la innovación y mejorando la eficiencia.

La reutilización de código: los microservicios pueden ser diseñados para ser utilizados en diferentes partes de una aplicación, lo que evita la duplicación de esfuerzos y promueve una base de código más limpia y mantenible.

2.5. Diseño de la arquitectura

Una arquitectura de "software" describe los componentes básicos de un sistema de "software" y su combinación interna. En el marco del desarrollo de "software", esta arquitectura es la decisión más temprana del diseño de una aplicación. Determina los criterios de calidad, como la mantenibilidad, modificabilidad, seguridad y rendimiento, además, le permite analizar de forma más clara cómo debe comportarse la aplicación que se va a desarrollar, teniendo en cuenta el acceso a ella y su capacidad.

Figura 7. Diseño de la arquitectura



2.6. Estilos arquitectónicos

Los estilos son un complemento de los patrones de diseño, como los arquitectónicos de "software", y lo importante aquí es que se preocupan por los "stakeholders". Por lo tanto, los estilos se preocupan por el tratamiento estructural del "software".

Cuando se aplican estilos arquitectónicos en el desarrollo de "software", se le da un valor distintivo a la calidad del "software". Los estilos son un conjunto de componentes, como bases de datos o módulos de cómputo, que son utilizados para realizar una función y, por medio de conectores, permiten la comunicación, coordinación y cooperación entre ellos, "el todo es la suma de sus partes".

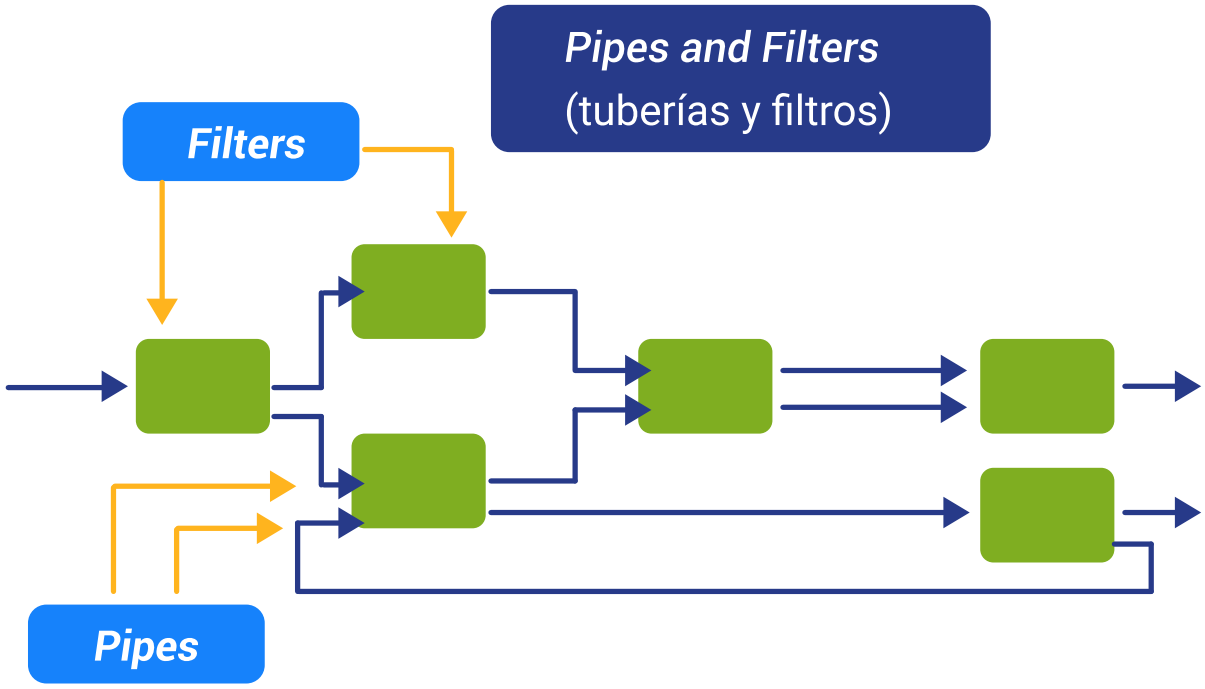
Un estilo arquitectónico se define teniendo claros los elementos, la forma y la razón por la cual se seleccionan dichos elementos. Dentro de los elementos, se deben tener en cuenta el procesamiento, los datos y las conexiones. Entre la forma, se deben considerar las propiedades y relaciones que estos elementos deben tener para cumplir con su funcionalidad. Por último, la razón no es más que el análisis mediante el cual se seleccionaron esos elementos para satisfacer las necesidades del cliente o del desarrollador.

Hay diferentes tipos de estilos arquitectónicos, algunos de los más conocidos son:

Arquitectura centrada en datos

Esta arquitectura se selecciona cuando se quiere resolver una necesidad de transformación de datos en cada filtro de información, y estas transformaciones son independientes de cada filtro. Estos filtros son componentes y las tuberías son los medios de comunicación entre los componentes, como se representa en la siguiente imagen.

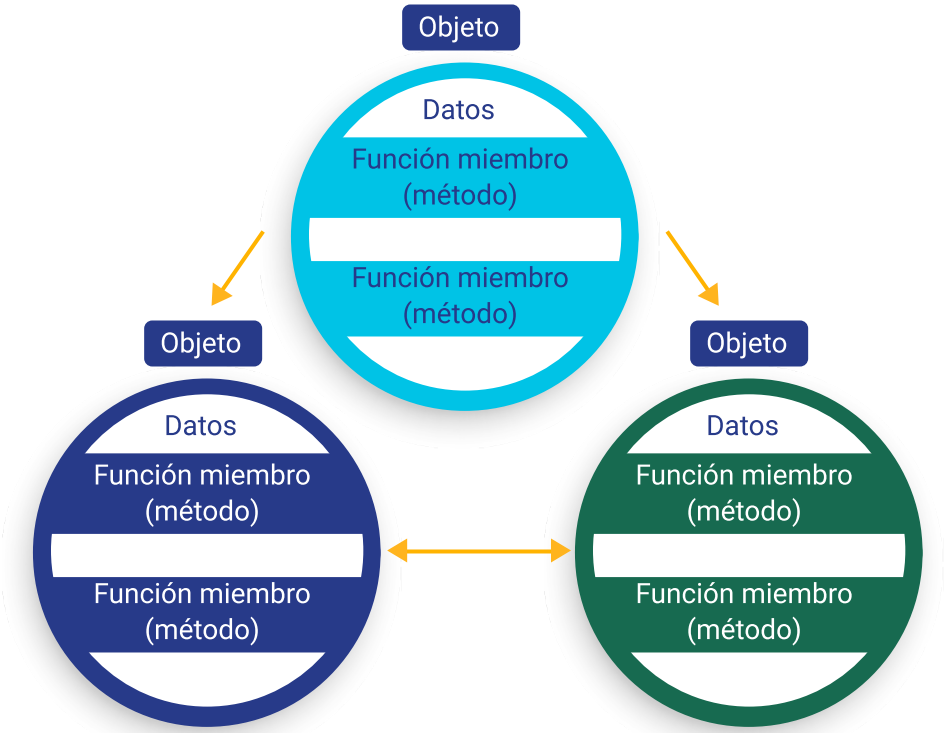
Figura 8. Tuberías y filtros



Arquitectura Abstractos de datos y POO

Se piensa en esta arquitectura si la necesidad de la solución informática es integrar varias fuentes de datos desde diferentes aplicaciones o sobre la misma.

Figura 9. Principios de la programación orientada a objetos

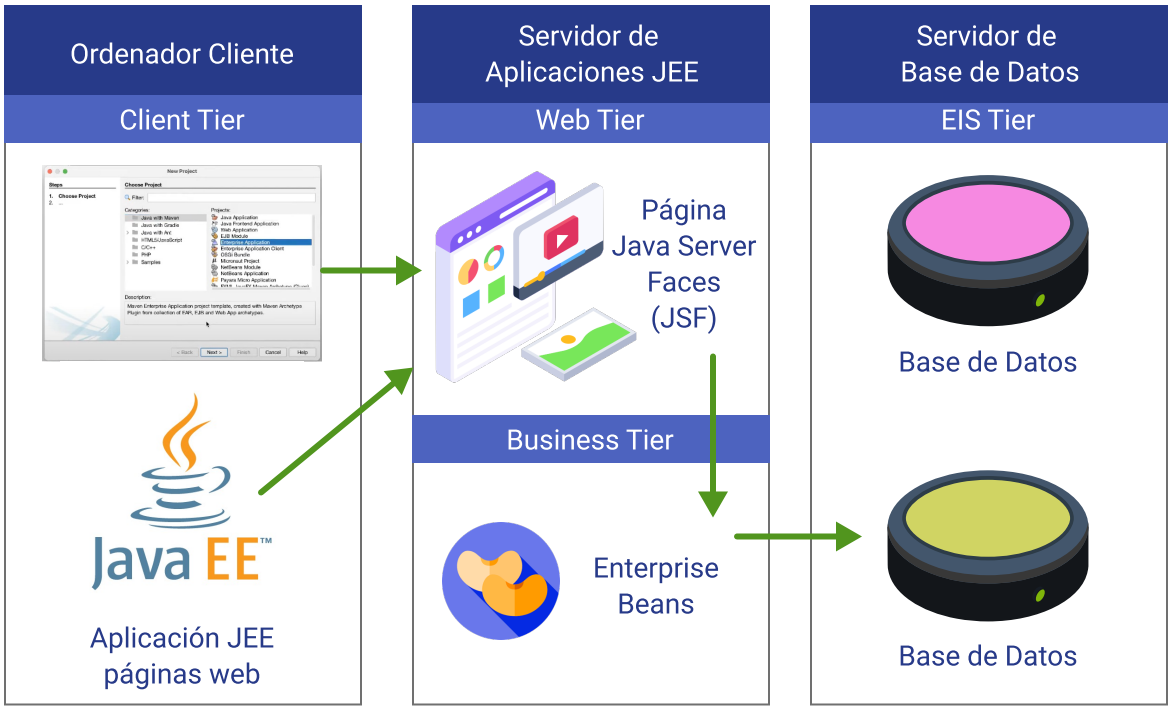


Nota. Rojas, Y. (2021). Base de Datos.

Arquitectura de llamada y retorno

Refleja la estructura del lenguaje de programación. Permite al diseñador del "software" construir una estructura de programa relativamente fácil de modificar y ajustar a escala. Se basa en la bien conocida abstracción de procedimientos/funciones/métodos.

Figura 10. Aplicación empresarial Java EE

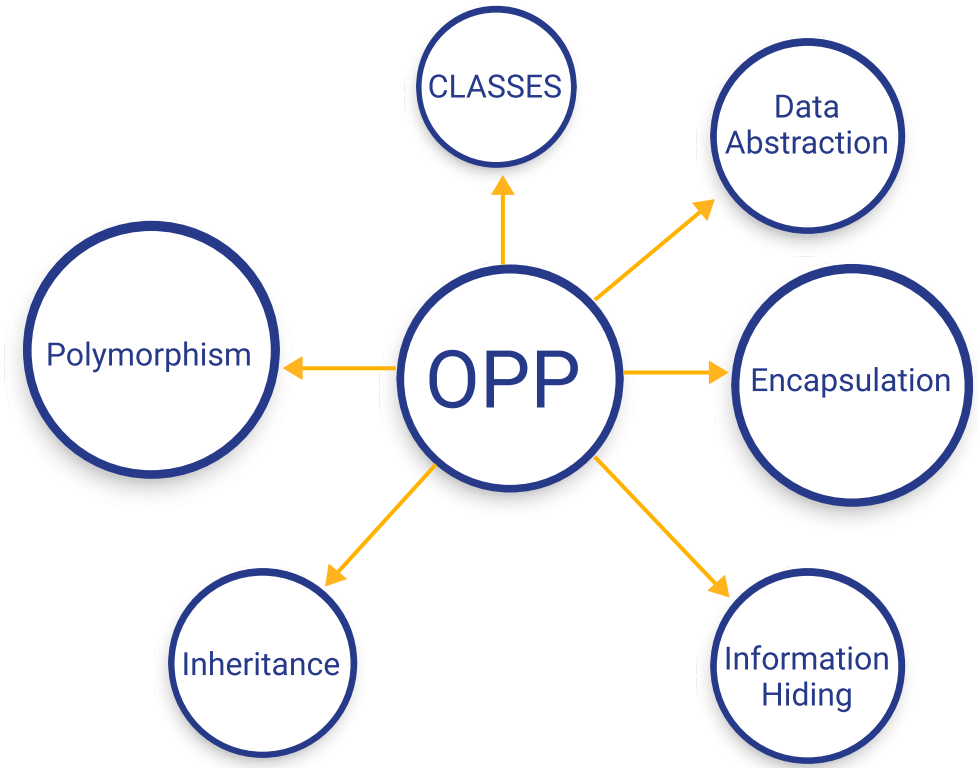


Nota. Ayala, J. (s.f.). Modelo de Aplicación de Java Empresarial.

Arquitectura orientada a objetos

Los componentes de un sistema encapsulan los datos y las operaciones que se deben realizar para manipular los datos. La comunicación y la coordinación entre componentes se consiguen a través del paso de mensaje.

Figura 11. Programación Orientada a Objetos (OOP)

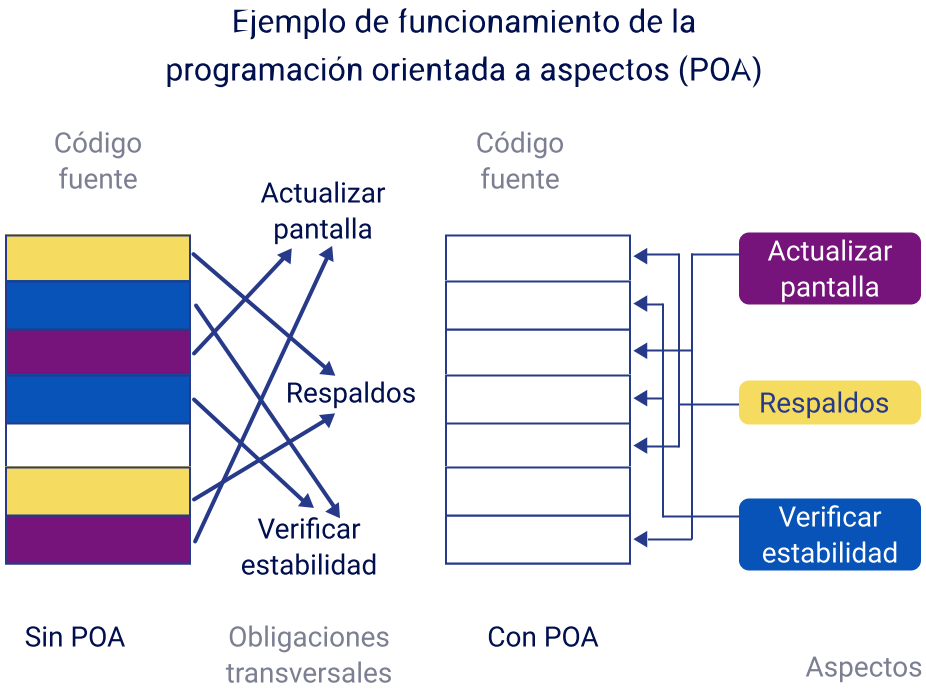


Nota. Cuéllar, J. (2010). Estilos y patrones básicos en arquitectura de software.

Arquitectura orientada a aspecto

El uso de AOP puede aislar las diversas partes de la lógica empresarial, reduciendo así el acoplamiento entre las diversas partes de esta lógica, mejorando la reutilización del programa y mejorando la eficiencia del desarrollo, al mismo tiempo.

Figura 12. Programación tradicional Vs. Programación Orientada a Aspectos (POA)

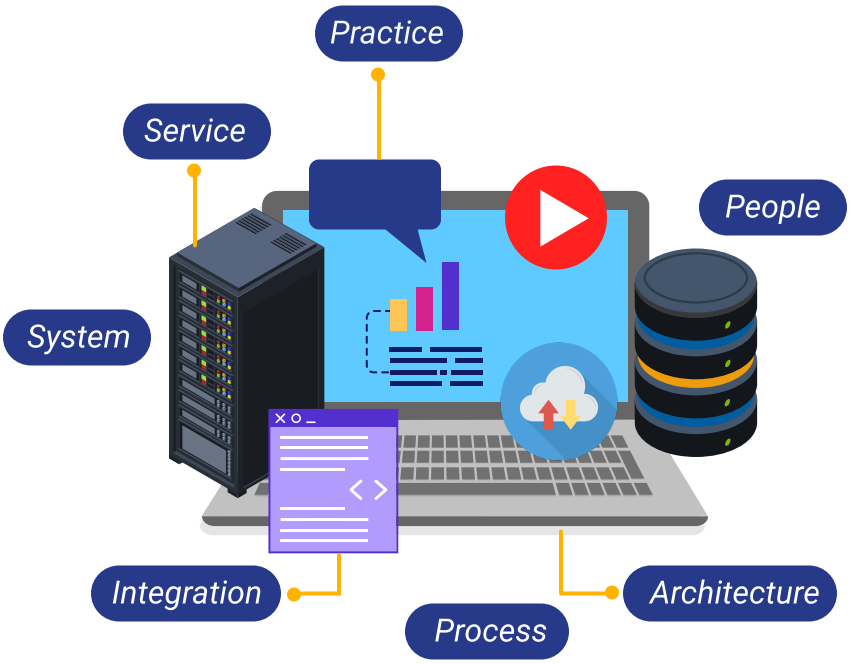


Nota. Ruelas, U. (2017). Poa-aop-programación-orientada-a-aspectos-aspect-oriented-programming-ejemplo-qué-es-paradigma.

Arquitectura orientada a servicios (SOA)

La Arquitectura Orientada a Servicios es un estilo de arquitectura de TI que se apoya en la orientación a servicios. Y esta orientación a servicios es una forma de pensar en: servicios, su construcción y sus resultados.

Figura 13. Esquema de la Arquitectura Orientada a Servicios, SOA

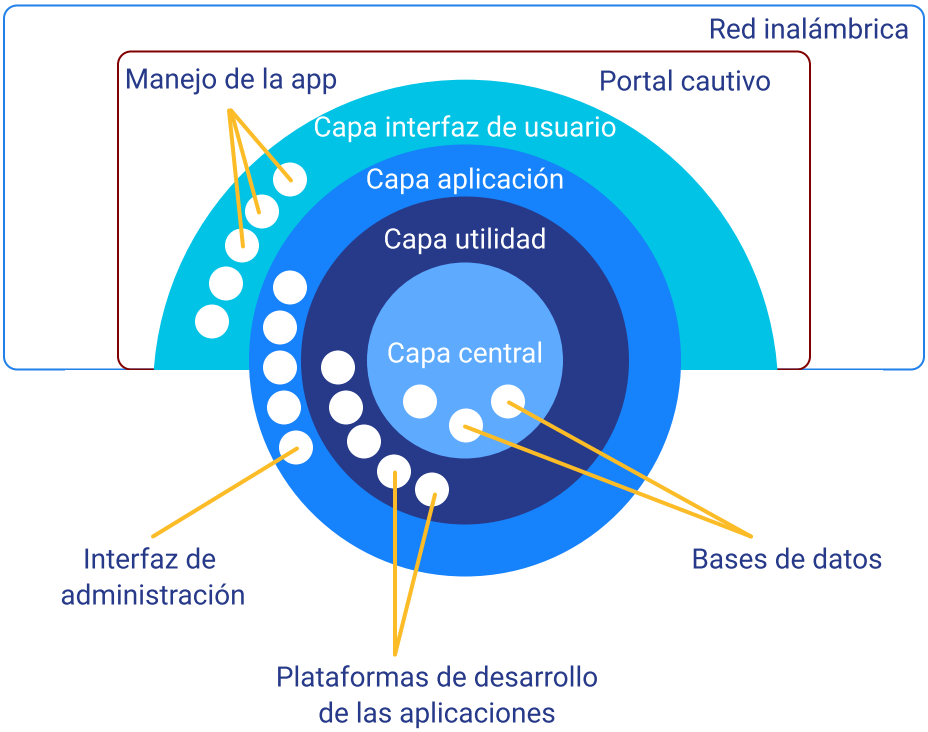


Nota. Fusap. (2021). INTEGRACIÓN SOA-OSB: TELECOM.

Arquitectura estratificada

Se crean diferentes capas y cada una realiza operaciones que progresivamente se aproximan más al cuadro de instrucciones de la máquina. En la capa externa, los componentes sirven a las operaciones de interfaz de usuario.

Figura 14. Estructura de una plataforma de desarrollo de aplicaciones



Nota. Zambrano, L. (2017). Diseño de la arquitectura estratificada de la solución.

3. Patrones comportamentales

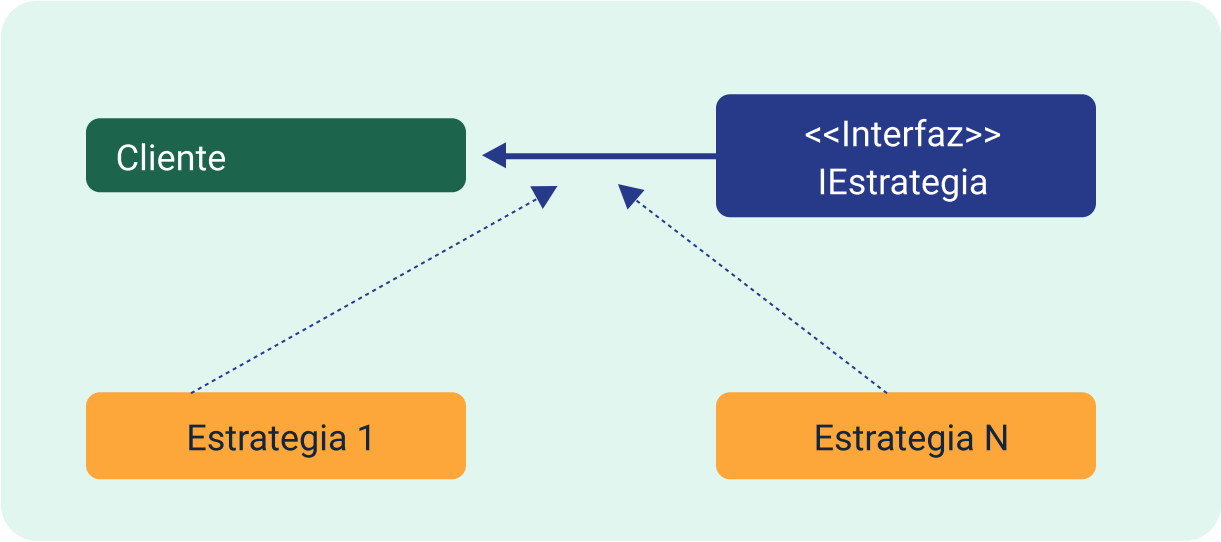
Como se había mencionado en los conceptos GOF, los patrones de diseño comportamentales se centran en definir la forma en cómo los objetos interactúan entre ellos por medio de mensajes.

Los patrones de comportamiento gestionan algoritmos, relaciones y responsabilidades entre objetos.

3.1. Estrategia

El patrón estrategia permite encapsular un conjunto de algoritmos de forma que puedan ser seleccionados dinámicamente durante el tiempo de ejecución de acuerdo con las acciones del cliente. Este patrón es una de las formas en las que se ven reflejadas fácilmente las características de la programación orientada a objetos, particularmente, lo referente a encapsulamiento y polimorfismo (Landa, 2018c).

Figura 15. Ejemplo de Patrón estrategia



Este patrón de diseño es útil cuando una misma funcionalidad puede ser provista usando diferentes mecanismos, algoritmos o estrategias, que serán seleccionadas

dependiendo de las acciones realizadas por el cliente en el momento que está ejecutando el programa.

Para un mejor entendimiento del patrón, suponga que quiere implementar una calculadora, la cual provee un conjunto de operaciones (suma, resta, multiplicación y división) que serán usadas por el cliente según su deseo. Cada una de estas operaciones representa una estrategia diferente y será el cliente quien invocará la ejecución de cada una de ellas dependiendo de su deseo, por medio de una interfaz que, usando las propiedades del polimorfismo, se transformará para poder responder a cada solicitud.

Ejemplo

Consideremos el ejemplo de implementar una calculadora que ofrece un conjunto de operaciones (suma, resta, multiplicación y división) para que el cliente las utilice según necesite. Cada una de estas operaciones constituye una estrategia distinta. Será el cliente quien decida cuál de ellas ejecutar, basándose en sus requerimientos, a través de una interfaz. Esta interfaz, aprovechando las propiedades del polimorfismo, se adapta para responder adecuadamente a cada tipo de solicitud.

3.2. Comando

El patrón comando permite aislar los objetos que realizan una petición de los objetos concretos encargados de recibir y realizar dicha acción. Esto permite, entre otras cosas, que las peticiones puedan ser enviadas a varios receptores y, si se maneja el estado de las solicitudes, controla acciones de tipo Undo y Redo.

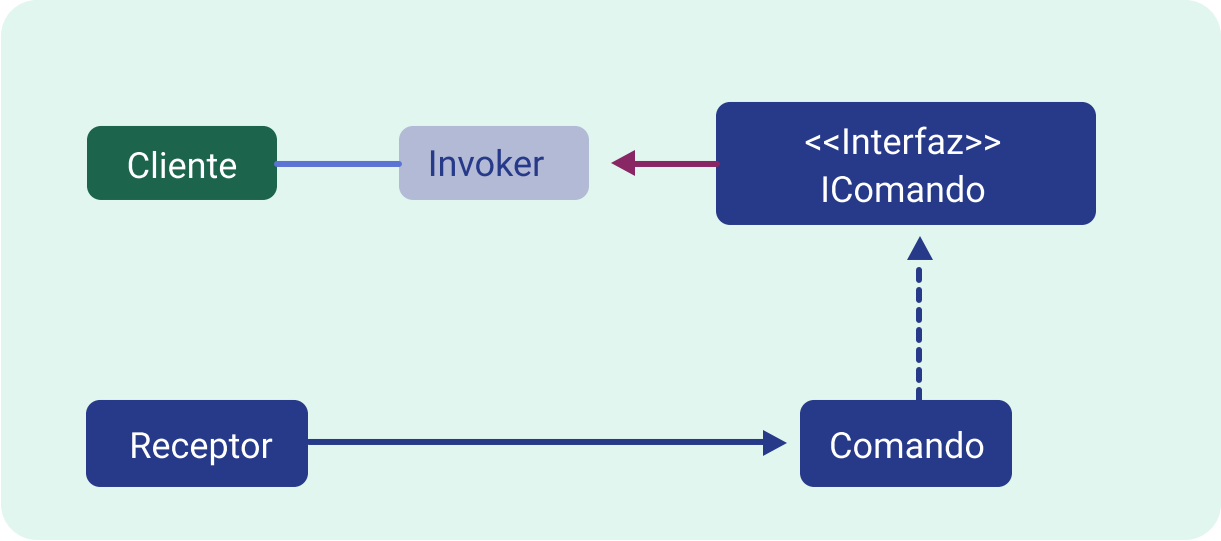
El patrón comando necesita la implementación de varios elementos (Landa, 2018c), así:

- **Comando:** es la clase que sirve de puente entre el cliente y los receptores.

- **Invoker:** elemento usado por los clientes y que le solicita al comando llevar a cabo una acción.
- **Cliente:** invoca la ejecución de las acciones desde el Invoker.
- **IComando:** interfaz donde se especifican las operaciones a ejecutar.
- **Receptor:** clase que realiza la acción.
- **Ejecutar:** operación que necesita ser llevada a cabo.

En la siguiente figura, se enuncia cada uno de los componentes del patrón y cómo interactúan entre ellos.

Figura 16. Diagrama patrón comando



Ejemplo

Para entender mejor este patrón, imagine el siguiente contexto: una persona (cliente) quiere hacer uso del televisor y, para poder realizar esto, hace todas las solicitudes de servicios por medio del control remoto (Invoker). El control remoto se comunica con una interfaz que se encarga de responder a las solicitudes de cada uno de los comandos que el usuario puede hacer, como, por ejemplo, prender el televisor,

apagar el televisor, subir el volumen, etc. Cada comando realiza una acción particular sobre el televisor (Receptor).

3.3. Iterator

Este patrón de diseño está orientado al trabajo con colecciones y facilita el acceso a todos los elementos de la colección sin tener la necesidad de conocer su estructura.

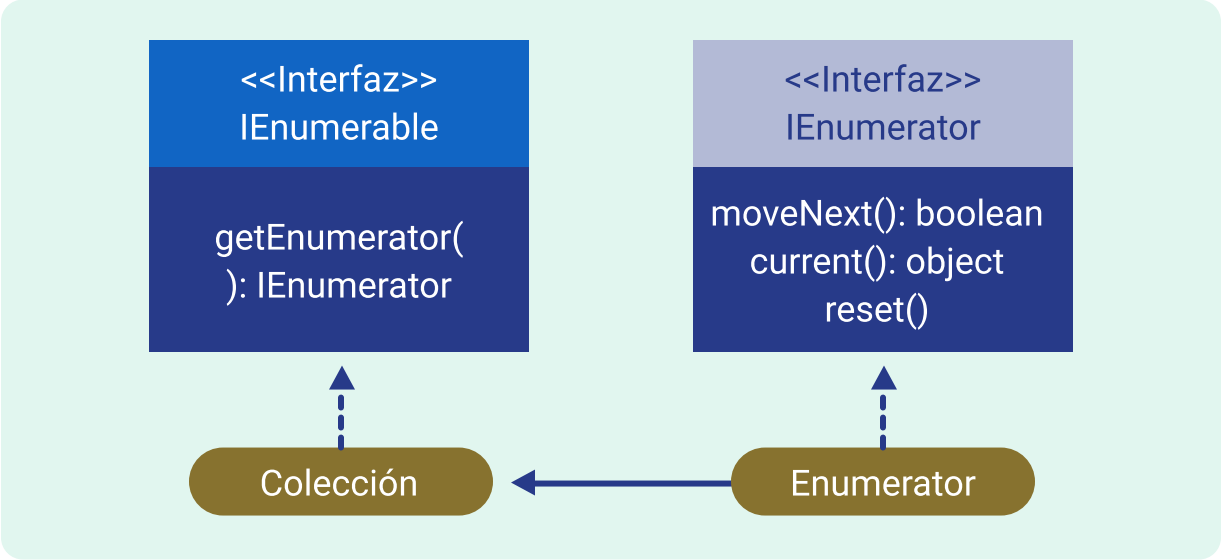
En este patrón, se reconocen dos elementos clave: los enumeradores y los iteradores:

- **Los enumeradores:** como su nombre lo indica, es el encargado de establecer la secuencia con la cual se puede conocer el siguiente elemento de la estructura.
- **Los iteradores:** corresponden a un mecanismo que permite recorrer la estructura de acuerdo con su secuencia de inicio a fin.

El enumerador, se encarga de implementar un conjunto de métodos estándar para poder establecer la secuencia con la que se debe recorrer la estructura. Entre los métodos más comunes, se encuentra, por ejemplo, el método moveNext(), el cual indica si existe o no un próximo elemento por recorrer; el método Current(), que devuelve el valor actual de la colección según la posición actual en la secuencia; y el método Reset(), que permite iniciar nuevamente la secuencia desde su punto de partida. El iterador necesita del enumerador para poder hacer el proceso de recorrido (Landa, 2018c).

En la siguiente figura, se enuncia cada uno de los componentes del patrón y cómo interactúan entre ellos.

Figura 17. Diagrama patrón iterador



Este tipo de operaciones son tan comunes en los sistemas actuales que los lenguajes de programación ya poseen una implementación propia del patrón iterador.

4. Patrones creacionales

Una de las tareas habituales en el proceso de construcción de "software" es distribuir responsabilidades entre un conjunto de módulos o clases, siguiendo los principios establecidos por los paradigmas de programación, como la programación orientada a objetos. No obstante, al abordar un requerimiento específico, será necesario instanciar objetos de distintos tipos que, asumiendo sus responsabilidades, ejecutan cierta lógica.

Los patrones creacionales proporcionan diversos mecanismos de creación de objetos, que aumentan la flexibilidad y la reutilización del código existente de una manera adecuada a la situación. Esto le da al programa más flexibilidad para decidir qué objetos deben crearse para un caso de uso dado.

4.1. Singleton

Este patrón de diseño creacional se encarga de definir la forma en que se puede garantizar que exista una única instancia de una clase particular en el contexto de la aplicación. Esto es útil para casos en los que, por cuestiones de manejo de memoria o de la lógica del negocio, se requiere que sea el mismo objeto quien responda todos los mensajes, independientemente del contexto actual de la aplicación.

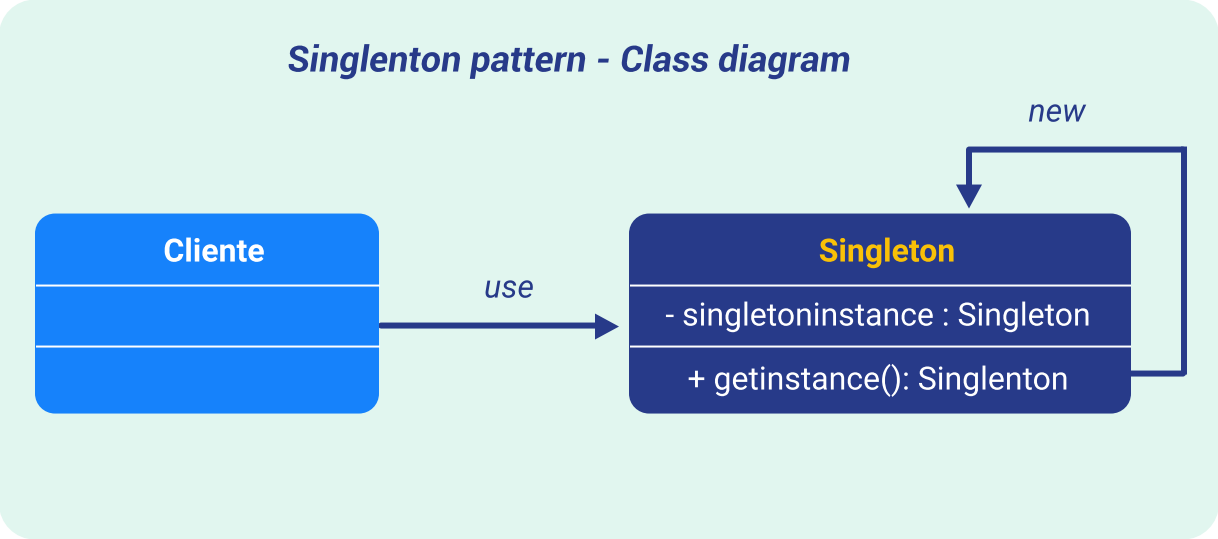
Este patrón se ve reflejado en el cuerpo de la misma clase que se requiere sea instancia una sola vez.

Ejemplo

Puede ser el manejo de conexiones a bases de datos. En algunos casos, no sería conveniente instanciar nuevos objetos de conexión cada vez que se genere un evento.

A continuación, se enuncia cada uno de los componentes del patrón relacionados entre el cliente y los requerimientos.

Figura 18. Diagrama patrón Singleton



4.2. Fábrica abstracta

Es un patrón de diseño creacional que permite producir familias de objetos relacionados sin especificar sus clases concretas. A continuación, podrá apreciar en detalle sus características.

Este patrón de diseño permite las instancias de una familia de objetos relacionados o dependientes, sin necesidad de entrar en detalles particulares de su implementación, de forma que tanto el cliente como los elementos creados por la fábrica son totalmente independientes unos de otros.

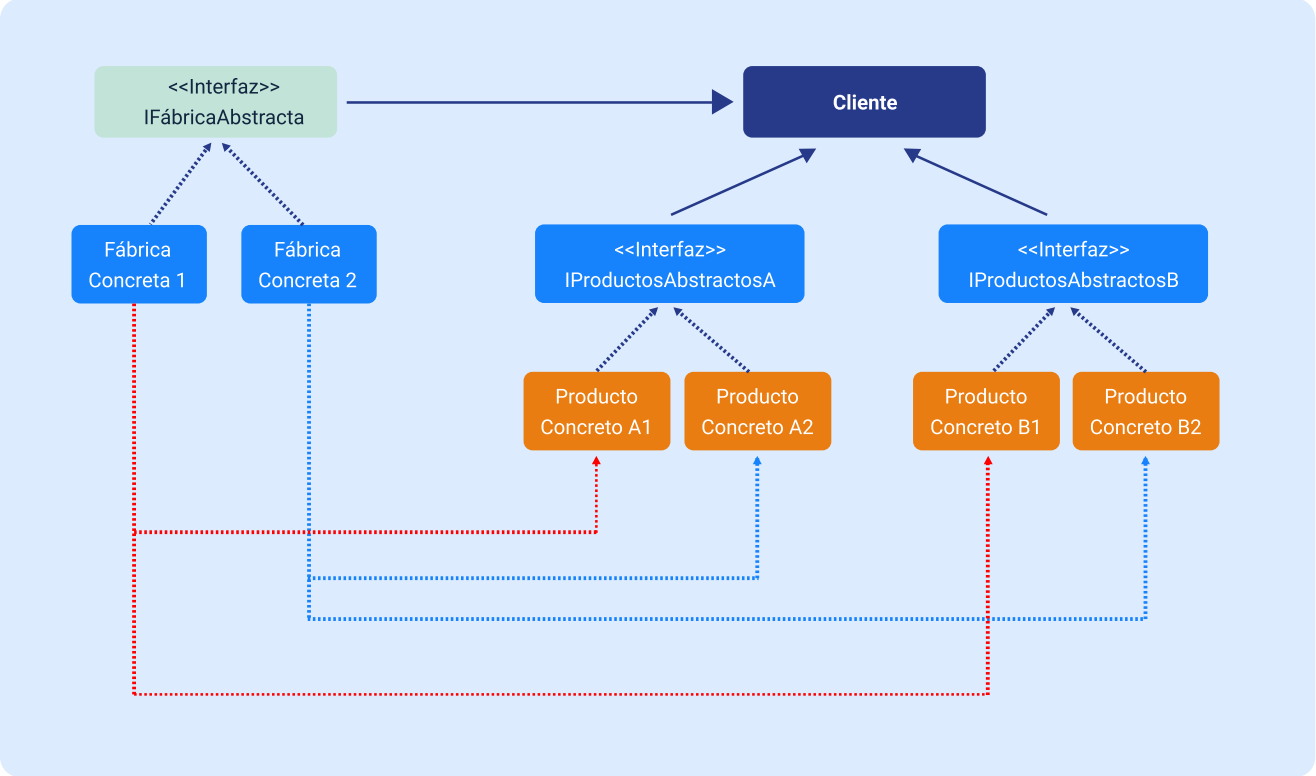
Este patrón define una interfaz de tipo fábrica, la cual se refina por medio de la creación de fábricas concretas, las cuales pueden producir diferentes tipos de objetos y en diferentes combinaciones, según las necesidades del cliente.

Los elementos importantes de este patrón se describen a continuación (Landa, 2018c):

- **Interfaz de la fábrica abstracta**, la cual relaciona las operaciones de creación de los productos abstractos.
- **Fábricas concretas**, las cuales implementan las operaciones de creación definidas en la interfaz de fábrica abstracta.
- **Interfaz de producto**, la cual relaciona los comportamientos de los productos o elementos.
- **Productos concretos**, los cuales implementan los comportamientos definidos en la interfaz de producto.

En la siguiente figura se enuncia cada uno de los componentes.

Figura 19. Componentes de un patrón de fábrica abstracta



5. Patrones estructurales

Los patrones estructurales son un conjunto de conceptos y principios fundamentales en el diseño de "software" que proporcionan una guía invaluable para organizar y definir los componentes de los objetos de manera eficiente y coherente. Estos patrones se basan en la idea de establecer relaciones y estructuras entre los diferentes elementos de un sistema de "software", permitiendo una arquitectura más robusta, flexible y fácil de mantener.

5.1. Fachada

El patrón fachada se utiliza cuando el sistema está compuesto por varios subsistemas y se hace complejo gestionar los mensajes que debe realizar el cliente en cada uno de estos subsistemas.

Este patrón permite generar al cliente una vista de alto nivel que simplifica el control y el envío de mensajes a los subsistemas, ocultando los detalles relacionados con la gestión de las clases e instancias, como se detalla en el siguiente recurso educativo.

Existen diferentes variaciones del patrón fachada (Landa, 2018c):

- **Opaco:** una de las variaciones más utilizadas y en la cual los clientes no pueden acceder a los subsistemas, solo se puede hacer mediante el objeto fachada.
- **Transparente:** el cliente tiene la posibilidad de acceder a los subsistemas por medio de la fachada, pero también puede hacerlo de forma directa.

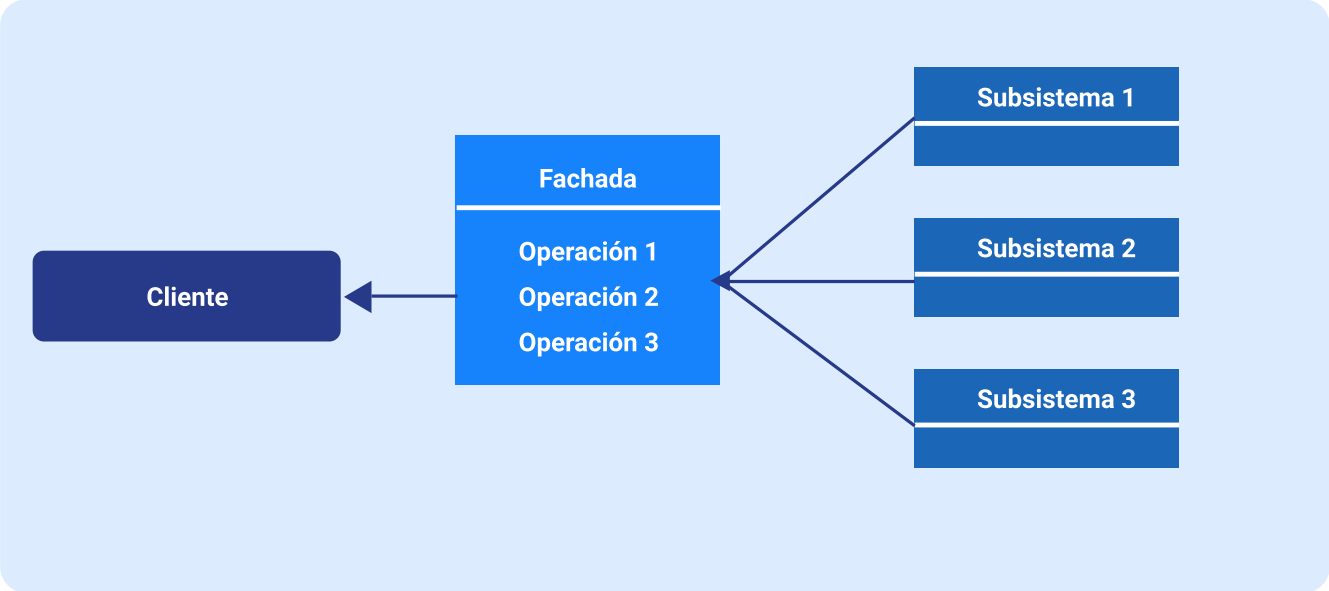
- **Estática:** en esta variación, la fachada se implementa como una clase estática, por lo cual, no se hace necesaria la instancia de un objeto concreto de la fachada.

En este patrón se reconocen tres partes fundamentales (Landa, 2018c):

- **Fachada:** clase que provee las operaciones de alto nivel que serán usadas por el cliente.
- **Subsistemas:** clases que proveen las funcionalidades que son expuestas por la fachada.
- **Cliente:** hace uso de las operaciones de alto nivel por medio de la fachada.

En la siguiente figura, se enuncia cada uno de los componentes del patrón.

Figura 20. Componentes del patrón fachada



5.2. Delegate

El patrón delegate se usa cuando se quieren reutilizar y extender funcionalidades de una clase sin hacer uso de la herencia. Este patrón permite, de cierta forma, implementar algo similar a la herencia múltiple, que no es admitida por algunos lenguajes de programación, pero, adicionalmente, permite tener un control más detallado sobre este proceso, ya que se puede ocultar parte de los elementos heredados o, incluso, compartir elementos que no son posibles de heredar bajo el mecanismo de herencia tradicional.

Este patrón lo que busca es evitar asumir todas las responsabilidades en una sola instancia y delegar las actividades en otras instancias que son especializadas en resolver dichas tareas.

6. Vistas estáticas

La vista estática está encargada de modelar los conceptos significativos del dominio de la aplicación desde sus propiedades internas y las relaciones existentes. Se denomina vista estática porque no modela el comportamiento del sistema ni muestra las variaciones que se puedan presentar por efecto del tiempo.

Los elementos fundamentales de la vista estática son las clases, que describen los conceptos del dominio del problema, y las relaciones, que pueden ser de tipo asociación, generalización y de dependencia. Entre los diagramas de UML que se utilizan para representar la vista estática del sistema, se encuentran (ITCA-FEPADE, s. f.):

- Diagrama de clases
- Diagrama de objetos
- Diagrama de componentes

A continuación, se detallan algunos diagramas de UML que permiten tener una visión más generalizada de los sistemas de información en desarrollo o desarrollados por el equipo de trabajo.

7. Diagrama de despliegue

Los diagramas de despliegue hacen parte de los tipos de diagrama propuestos por UML y su objetivo es la representación de la arquitectura del sistema en términos de “hardware” y “software” físico, y los medios por los cuales se conectan. Este tipo de diagrama es muy útil para el proceso de despliegue del sistema.

Los diagramas de despliegue utilizan un conjunto de elementos gráficos que tienen una representación y significado estandarizados. A continuación, se detalla cada uno de ellos (Cinergix Pty. Ltd., 2021):

- **Nodos:** representan un elemento que puede ser “hardware” o “software” y se grafican mediante un cubo.
- **Artefactos:** representan elementos concretos generados en el proceso de desarrollo, como, por ejemplo, bibliotecas, archivos, etc.
- **Asociación de comunicación:** representa el camino de comunicación entre los nodos y se grafica mediante una línea continua que une los nodos en cuestión.
- **Dispositivos:** es un tipo especial de nodo que representa un recurso computacional del sistema, como, por ejemplo, un servidor.
- **Especificaciones de despliegue:** representan configuraciones que se deben tener en cuenta para desplegar un artefacto en un nodo.
- **Figura ilustrativa:** este un tipo de ejemplo de un diagrama de despliegue para un sistema de gestión de biblioteca.

8. Diagrama de componentes

El diagrama de componentes es uno de los diagramas propuestos en UML que representa una vista estática del sistema de información y hace parte de los diagramas estructurales. Este diagrama proporciona una vista de alto nivel de los componentes dentro del sistema y generalmente se construye posterior a la construcción del diagrama de clases.

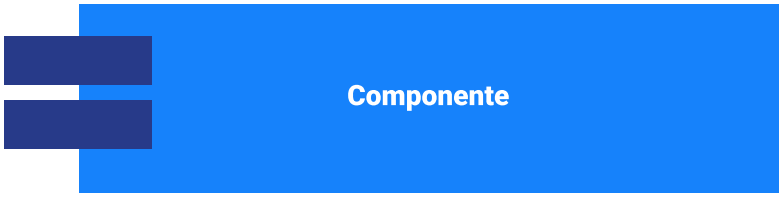
Un componente puede ser “software”, como, por ejemplo, las bases de datos o una interfaz de usuario; y también puede ser “hardware”, como un dispositivo o incluso una unidad de negocio, como, por ejemplo, la nómina, el inventario, proveedores, etc.

Este tipo de diagrama es muy útil para arquitecturas orientadas a servicios, permite mostrar la estructura general del código, por lo que puede ser usado para mostrar las funciones del sistema que se construye a cualquier parte interesada.

Los elementos que conforman un diagrama de componente son los siguientes (Diagramas UML, 2019):

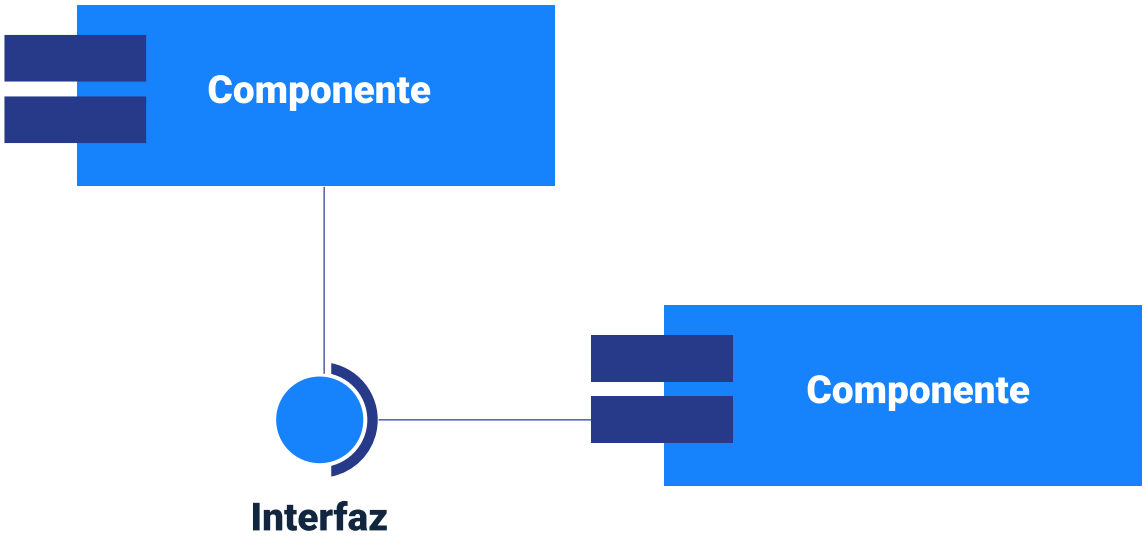
- **Componente:** es una abstracción de un nivel más alto que las clases y representa unidades lógicas del sistema.

Figura 21. Componente



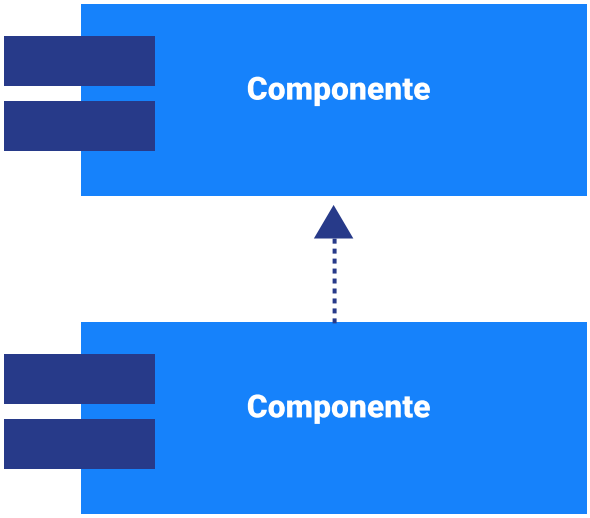
- **Interfaz:** las interfaces siempre se asocian con los componentes y representan el lugar que debe ser usado por otros componentes para poder establecer comunicación con él.

Figura 22. Interfaz



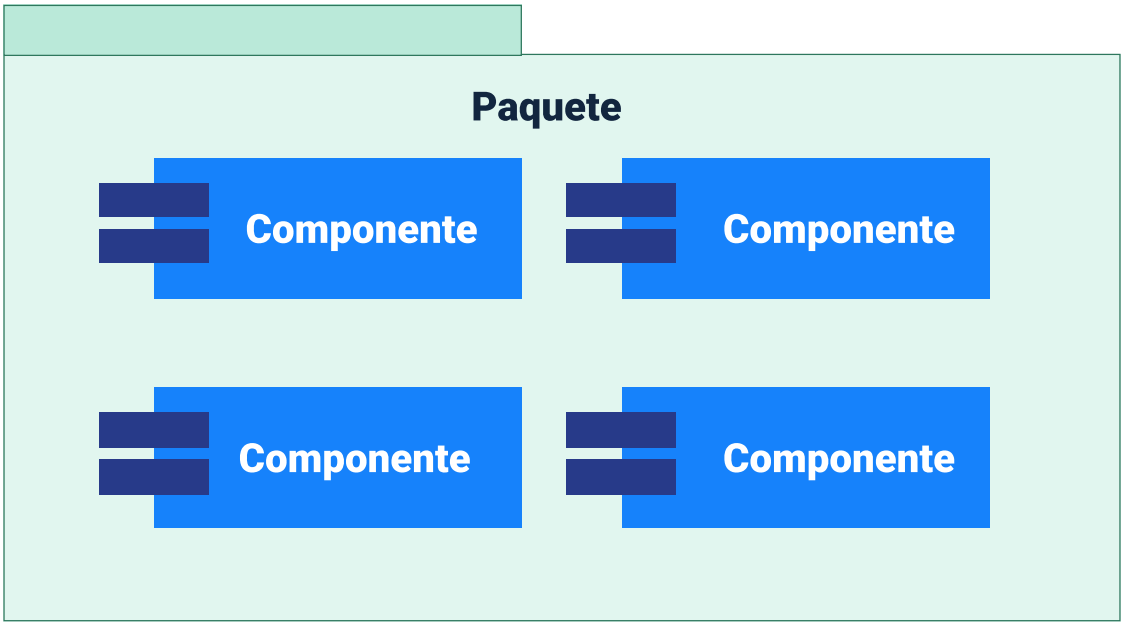
- **Relación de dependencia:** representa una relación más general entre dos componentes, para indicar que un componente requiere de otro para poder ejecutar su trabajo.

Figura 23. Relación de dependencia



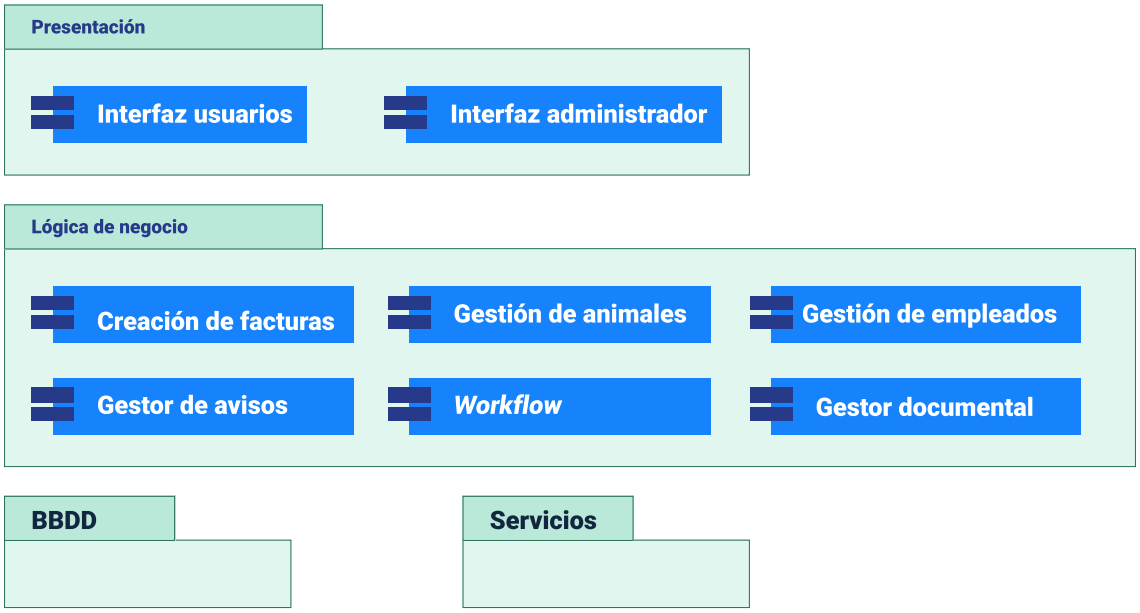
- **Paquetes:** es posible utilizar los paquetes para agrupar lógicamente un conjunto de componentes dentro de un subsistema.

Figura 24. Paquetes



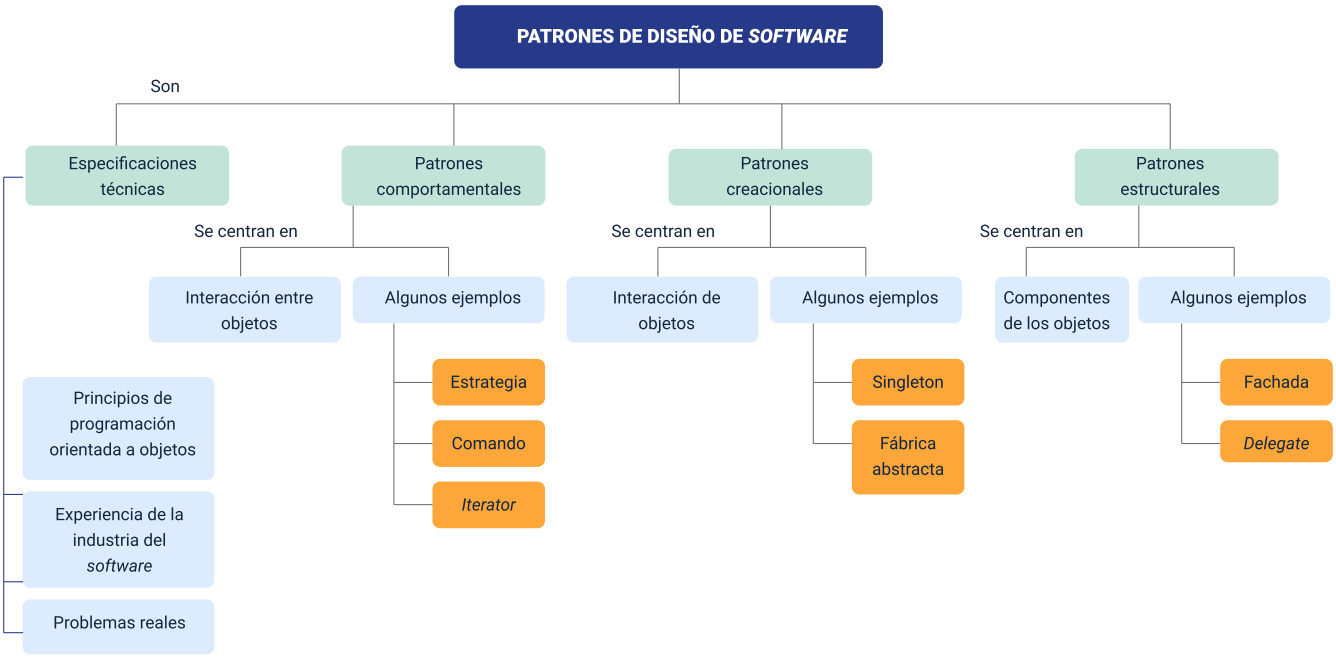
- **Ejemplo:** En la figura, se muestra un ejemplo de un diagrama de componentes para un sistema de gestión de una clínica veterinaria.

Figura 25. Ejemplo



Síntesis

A continuación, se muestra un mapa conceptual de los elementos más importantes desarrollados en este componente:



Material complementario

Tema	Referencia	Tipo de material	Enlace del recurso
Patrones de diseño	Leiva, A. [DevExperto]. (2020). Patrones de diseño “software”: Repaso completo en 10 minutos.	Video	https://youtu.be/6BHOeDL8vls
Diagrama de despliegue	Landa, N. [nicosiored]. (2018b). Diagrama de Despliegue - 22 - Tutorial UML en español.	Video	https://youtu.be/NSB0ATJUavA
Diagrama de componentes	Landa, N. [nicosiored]. (2018a). Diagrama de Componentes I - 20 - Tutorial UML en español.	Video	https://youtu.be/oOycG_n1ARs
Arquitectura Multicapa	Garrido, A. [Universitat Politécnica de Valéncia - UPV]. (2017). Arquitectura del “software” multicapa.	Video	https://www.youtube.com/embed/kHvxX1E9vIU
Patrón Modelo Vista Controlador	Fazt. (2017). ¿Qué es el patrón Modelo Vista Controlador?, Explicación simple.	Video	https://www.youtube.com/embed/ANQDmqBYwns
Arquitectura Monolítica	Lazy Loading. (2021). Arquitectura monolítica vs microservicios.	Video	https://www.youtube.com/embed/99YMeCBk3jw
Estilos de Arquitectura	Mercury 7w7. (2020). Estilos de Arquitectura de “software”.	Video	https://www.youtube.com/watch?v=PK9TTcTosTw
Arquitectura Microservicios	MegaPractical. (2016). Arquitectura Orientada a Servicios (SOA), “Enterprise	Video	https://www.youtube.com/embed/o_Br2vZ4uQY

Tema	Referencia	Tipo de material	Enlace del recurso
	Service Bus" TIBCO en español.		

Glosario

Acoplamiento: es la forma y nivel de interdependencia entre módulos de "software"; una medida de qué tan cercanamente conectados están dos rutinas o módulos de "software".

API: es un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el "software" de las aplicaciones. API significa interfaz de programación de aplicaciones. Las API permiten que sus productos y servicios se comuniquen con otros, sin necesidad de saber cómo están implementados.

Aplicación: es un programa informático diseñado como una herramienta para realizar operaciones o funciones específicas. Generalmente, son diseñadas para facilitar ciertas tareas complejas y hacer más sencilla la experiencia informática de las personas.

Base de datos: es una recopilación organizada de información o datos estructurados, que normalmente se almacena de forma electrónica en un sistema informático.

"**Browser**": es el término inglés que se utiliza para identificar a un navegador web o navegador de Internet. Consiste en un "software", programa o incluso aplicación que ofrece al usuario el acceso a la red.

Delegar: dar [una persona o un organismo] un poder, una función o una responsabilidad a alguien para que los ejerza en su lugar o para obrar en representación suya.

Escalabilidad: capacidad de adaptación y respuesta de un sistema con respecto al rendimiento del mismo a medida que aumentan de forma significativa el número de usuarios del mismo.

Ícono: signo que representa un objeto o una idea con los que guarda una relación de identidad o semejanza formal.

Implementación: poner en funcionamiento o aplicar métodos, medidas, etc., para llevar algo a cabo.

Interfaz: la conexión física y funcional que se establece entre dos aparatos, dispositivos o sistemas que funcionan independientemente uno del otro. En este sentido, la comunicación entre un ser humano y una computadora se realiza por medio de una interfaz.

Patrón: quitar de una cosa lo malo, lo que es extraño o lo que no sirve, para dejarla pura. Los Patrones de diseño ("Design Patterns") son una solución general, reutilizable y aplicable a diferentes problemas de diseño de "software".

Servidor: un servidor es un conjunto de computadoras capaz de atender las peticiones de un cliente y devolverle una respuesta en concordancia.

Sintaxis: es la "parte de la gramática que estudia el modo en que se combinan las palabras y los grupos que estas forman para expresar significados, así como las relaciones que se establecen entre todas esas unidades".

Sitio web: se relacionan con las entidades clave de los proyectos de desarrollo: requerimientos, actividades y resultados.

UML: lenguaje unificado de modelado. Relaciona un conjunto de diagramas estandarizados para la representación de sistemas de información desde diferentes tipos de vista.

Web: conjunto de información que se encuentra en una dirección determinada de Internet.

Referencias bibliográficas

Amazon Web Services [AWS]. (s. f.). Microservicios. https://aws.amazon.com/es/microservices/

Ayala, J. (s.f.). Modelo de Aplicación de Java Empresarial. https://jmaw.blogspot.com/2012/09/modelo-de-aplicacion-de-java-empresarial.html

Bahit, E. (2011). POO y MVC en PHP. https://www.slideshare.net/eugeniabahit/poo-y-mvc-en-php-por-eugenia-bahit

Blancarte, O. (2020). Arquitectura Monolítica. "Reactive Programming". https://reactiveprogramming.io/blog/es/estilos-arquitectonicos/monolitico

Cinergix Pty. Ltd. (2021). La Guía Fácil de los Diagramas de Despliegue UML. Blog de Creately. https://creately.com/blog/es/diagramas/tutorial-de-diagrama-de-despliegue/

Cuéllar, J. (2010). Estilos y patrones básicos en arquitectura de "software". https://josecuellar.net/estilos-patrones-basicos-arquitectura-software/

Diagramas UML. (2019). Diagrama de componentes. https://diagramasuml.com/componentes/

EcuRed. (s. f.). Patrones Gof. https://www.ecured.cu/Patrones_Gof

Fusap. (2021). INTEGRACIÓN SOA-OSB: TELECOM. https://www.fusap.com.ar/desarrollos.html

Gamma, E., Helm, R., Johnson, R., Vlissides, J. y Booch, G. (1994). “Design Patterns: Elements of Reusable Object-Oriented Software”. Addison-Wesley Professional.

Hernández, R. (2021). El patrón modelo-vista-controlador: Arquitectura y “frameworks” explicados. https://www.freecodecamp.org/espanol/news/el-modelo-de-arquitectura-view-controller-pattern/

Hiberus. (2021, marzo 4). De una arquitectura tradicional a una arquitectura microservicios. Hiberus Tecnología. https://www.hiberus.com/crecemos-contigo/de-una-arquitectura-tradicional-a-microservicios/

Landa, N. [nicosiored]. (2018c). Patrones de Diseño de “Software” [Video]. YouTube. https://www.youtube.com/playlist?list=PLM-p96nOrGcbqbL_A29b0z3KUXdq2_fpn

Landeta_P. (2013). 2 2 estilos arquitectónicos. Slideshare. https://es.slideshare.net/landeta_p/2-2-estilos-arquitectonicos

Marrero, D. (2016). Patrón MVC. https://daniellamikaelson.files.wordpress.com/2016/07/20160327_56f72c4319db8.jpg?w=840

Novoseltseva, E. (2020). Los 5 principales patrones de Arquitectura de “Software”. Apiumhub. https://apiumhub.com/es/tech-blog-barcelona/principales-patrones-arquitectura-software/

Rojas, Y. (2021). Base de Datos. https://guiadelempresario.com/wp-content/uploads/2021/05/Base-de-datos-orientada-a-objetos.jpeg

Ruelas, U. (2017). Poa aop programación orientada a aspectos “aspect oriented programming” ejemplo qué es paradigma. https://codingornot.com/que-es-la-programacion-orientada-a-aspectos-aop/poa-aop-programacion-orientada-a-aspectos-aspect-oriented-programming-ejemplo-que-es-paradigma

Zambrano, L. (2017). Diseño de la arquitectura estratificada de la solución. https://www.researchgate.net/figure/Figura-1-Diseno-de-la-arquitectura-estratificada-de-la-solucion_fig1_318509442

Créditos

Nombre	Cargo	Centro de Formación y Regional
Milady Tatiana Villamil Castellanos	Responsable del Ecosistema	Dirección General
Olga Constanza Bermúdez Jaimes	Responsable de Línea de Producción	Centro de Servicios de Salud - Regional Antioquia
Jonathan Guerrero Astaiza	Experto Temático	Centro de Teleinformática y Producción Industrial - Regional Cauca
David Eduardo Lozada Cerón	Experto Temático	Centro de Teleinformática y Producción Industrial - Regional Cauca
Paola Alexandra Moya	Evaluadora Instruccional	Centro de Servicios de Salud - Regional Antioquia
Andrés Felipe Herrera Roldán	Diseñador de Contenidos Digitales	Centro de Servicios de Salud - Regional Antioquia
Jhon Jairo Urueta Álvarez	Desarrollador Fullstack	Centro de Servicios de Salud - Regional Antioquia
Edgar Mauricio Cortés García	Actividad Didáctica	Centro de Servicios de Salud - Regional Antioquia
Luis Gabriel Urueta Álvarez	Validador de Recursos Educativos Digitales	Centro de Servicios de Salud - Regional Antioquia
Jaime Hernán Tejada Llano	Validador de Recursos Educativos Digitales	Centro de Servicios de Salud - Regional Antioquia
Margarita Marcela Medrano Gómez	Evaluador para Contenidos Inclusivos y Accesibles	Centro de Servicios de Salud - Regional Antioquia
Daniel Ricardo Mutis Gómez	Evaluador para Contenidos Inclusivos y Accesibles	Centro de Servicios de Salud - Regional Antioquia