UNIVERSIDAD SAN PABLO DE GUATEMALA

Facultad de Ciencias Empresariales

Escuela de Ingeniería
Ingeniería en Ciencias y Sistemas de la Computación.

PATRONES DE ARQUITECTURA

Trabajo presentado en el curso de Ingenieria de Software

Impartido por el Ing. Marcos Alfredo Orozco De Paz

Nombre del estudiante
Henry Oswaldo Morales Vásquez
Carné 2200304

PATRONES DE ARQUITECTURA DE SOFTWARE: TIPOS Y CARACTERÍSTICAS

Los patrones de arquitectura son soluciones generales y reutilizables para problemas comunes en la arquitectura de software. Estos patrones proporcionan una guía para diseñar sistemas robustos y escalables.

PATRÓN EN CAPAS:

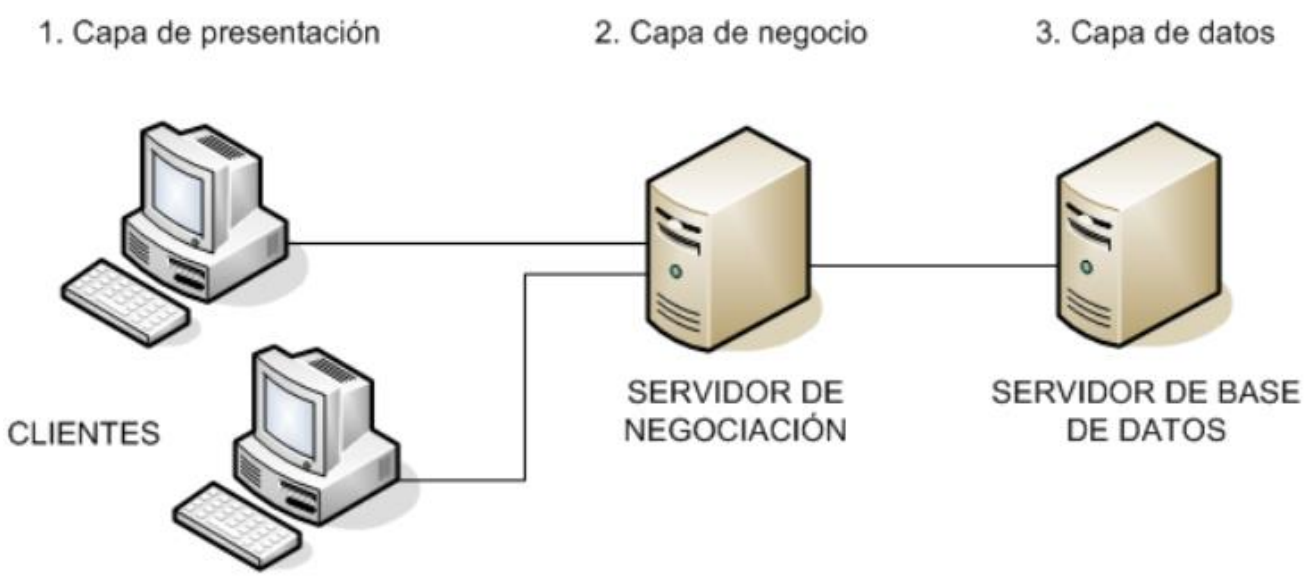
Este patrón de arquitectura de software divide la estructura del software en diferentes capas que comúnmente se conocen como:

- Capa de presentación.
- Capa lógica.
- Capa de datos.

En cada capa el programador organiza un tipo de código especifico; todas las vistas del software en una capa, la interactividad en otra, el desarrollo de las funcionalidades en la tercera y la base de datos en una capa final. De esta forma cada capa se dedica a una parte del software sin perder la relación. Pueden implementarse tantas capas como sean necesario, la comunicación de una capa es con la siguiente y con su anterior ejemplo: la lógica con los datos o la presentación con la lógica.

La idea de este patrón es evitar el acoplamiento, es decir, que un cambio en una línea de código implique cambiar todo lo demás. A través de la programación en capas se puede modificar solo una parte del software y el resto continuar funcionando correctamente.

También puede interpretarse como una arquitectura Cliente-servidor donde las funciones de presentación, lógica de negocio y gestión de datos se separan físicamente. Permite distribuir el trabajo de creación de una aplicación por niveles, de tal modo que cada grupo de trabajo está totalmente abstraído del resto de niveles, basta con conocer la API que existe entre niveles.



Capa de Presentación: Es la que el usuario ve, es donde se presenta el sistema al usuario, le comunica información y a la vez captura la información del usuario en un mínimo de proceso, se conoce también como "interfaz gráfica" dado las características que debe tener; amigable, entendible, fácil de usar para el usuario. Es importante hacer notar que esta capa solo se comunica con la capa de negocio.

Capa de negocio: aquí es donde se alojan los programas que son ejecutados, reciben las peticiones del usuario y se envían las respuestas tras el proceso, se denomina "capa de negocio" porque es donde se establecen las reglas que deben cumplirse, su comunicación es con la capa de presentación para captar los datos , solicitudes y presentar información, y con la capa de datos para solicitar a la base de datos información, gestionar datos, almacenar, borrar datos.

Capa de datos: Aquí residen los datos y se encarga de acceder a ellos. Se conforma por uno o más gestores de base de datos y se encarga de la gestión y administración de los datos. Recibe las peticiones de almacenamiento o recuperación desde la capa de negocio.

Todas la capas pueden estar en el mismo servidor físico, aunque lo usual es que la capa de presentación esté separada de la capa de negocio y datos que podrán estar en el mismo servidor físico.

PATRÓN DE TRES NIVELES:

La diferencia con el patrón de Capas es poca y básicamente es que en el patrón de capas de un software se pueden almacenar todas en un mismo servidor, pero al utilizar el patrón de tres niveles, esto implica que cada capa estará físicamente en un lugar distinto.

De nuevo, la cantidad de capas queda abierta, de acuerdo a como quede mejor organizado el código fuente. Algunas alternativas son:

- Vistas, lógica + Datos
- Vistas + lógica, lógica + datos

Organiza las aplicaciones en sus tres niveles informáticos lógicos y físicos.

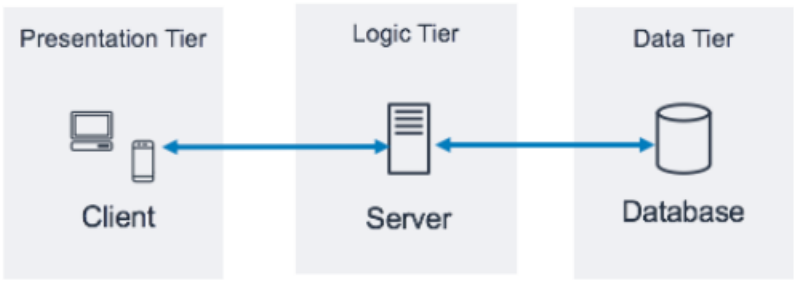
- Nivel de presentación o interfaz de usuario.
- Nivel de aplicativo, procesamiento de datos.
- Nivel de datos, donde se almacena y gestiona la información.

El beneficio de esta arquitectura es que debido a que cada nivel se ejecuta en su propia infraestructura , cada nivel puede ser desarrollado simultáneamente por un equipo destinado a cada capa.

Nivel de presentación: hablamos de la interfaz de usuario, donde el usuario final interactúa con la aplicación, su tarea principal es mostrar información al usuario y recopilar datos de él. Este nivel puede ser ejecutado en un navegador web o en una aplicación desktop, suelen desarrollarse en HTLM, CSS , JAVASCRIPT para el caso web, si es un app desktop existen varios lenguajes para este propósito.

Nivel de aplicación: Es el nivel lógico o medio, núcleo de la aplicación, los datos son recopilados en la capa de presentación y luego procesada en la capa de aplicación, implementa la lógica empresarial. En este nivel también se pueden añadir, suprimir o modificar datos en el nivel de datos. Generalmente se desarrollo en Python, Java, Perl, PHP, Ruby y su comunicación es a través de API´s.

Nivel de datos: Nivel de base de datos, acceso a la información o backend, se gestionan, almacenan la información procesada por la aplicación. Puede estar implementado en un sistema relacional de gestión de base de datos como PostgreSQL, MySQL, MariaDB, Oracle, DB2, SQLSERVER o una base NOSQL como MongoDB.



ARQUITECTURA DE MICROSERVICIOS & ARQUITECTURA ORIENTADA A SERVICIOS:

Ambos se basan en el desarrollo de un software mediante servicios que se ejecutan de forma individual, se comunican entre ellos utilizando código que se programa para que se llamen unos a otros.

Arquitectura de Microservicios: Solo se enfoca en determinar que se desarrolle una aplicación donde cada proceso o funcionalidad se represente por un servicio.

Proporciona un sistema moderno altamente escalable y distribuido.

En la arquitectura de microservicios, una aplicación se divide en una serie de servicios implementables de forma independiente, estos se comunican a través de API. Con ello se logra implementar y escalar cada servicio individual de forma independiente, así como la entrega rápida y frecuente de aplicaciones grandes y complejas.

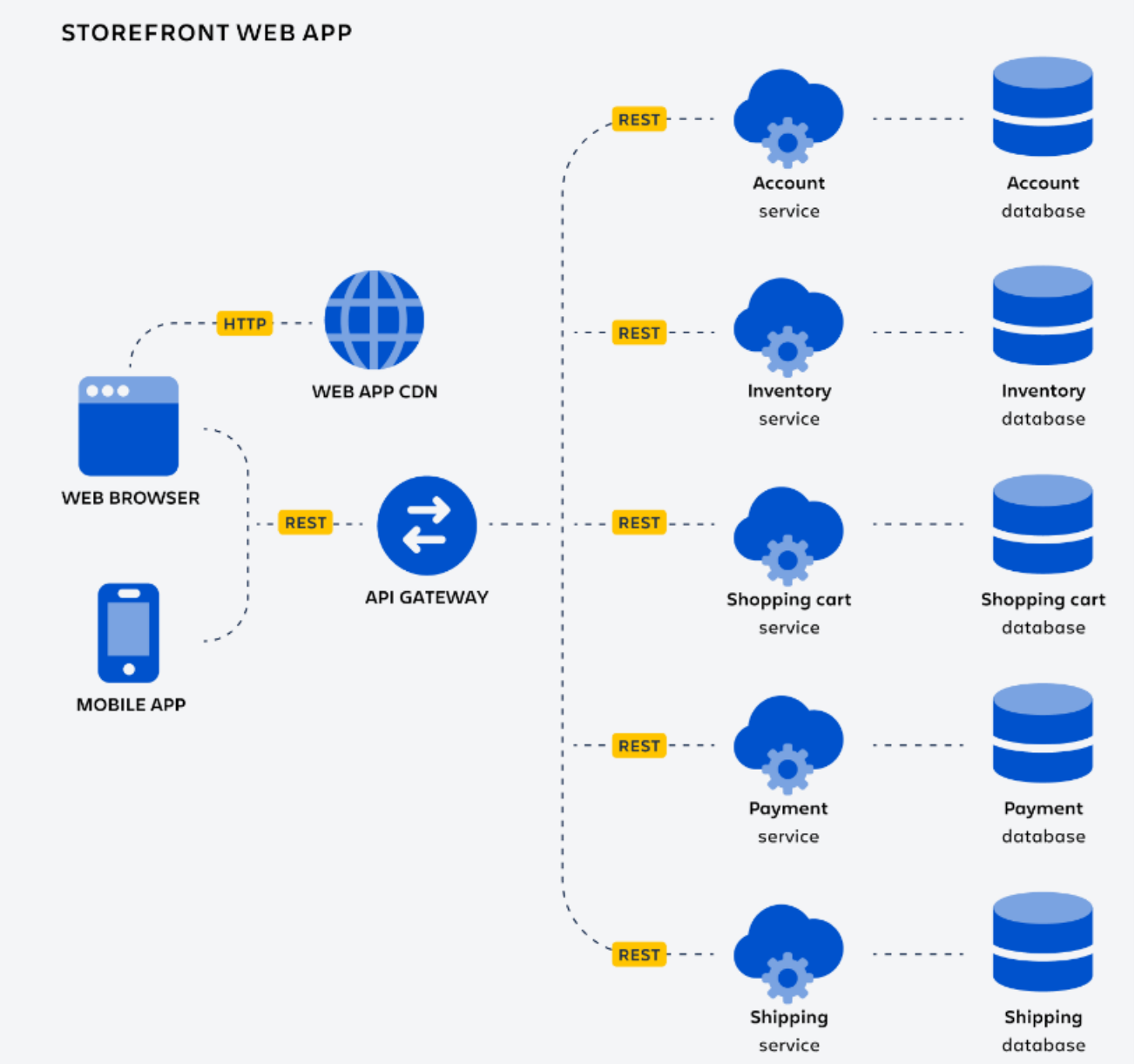
Características fundamentales:

- Varios servicios de componentes:
  Los microservicios se componen de servicios de componentes individuales y poco vinculados que se pueden desarrollar, implementar, operar, cambiar y volver a implementar sin afectar el funcionamiento de otros servicios o la integridad de una aplicación.
- Fácil de mantener y probar:
  Permite a los equipos experimentar con nuevas funciones y revertirlas si no funcionan. Facilita la actualización del código y acelera el tiempo de salida. Simplifica el proceso de aislamiento y corrección de fallos y errores en los servicios individuales.
- Pertenece a equipos pequeños:

Los equipos pequeños e independientes suelen crear un servicio dentro de microservicios, esto los anima a adoptar prácticas de metodologías ágiles y Devops. Dado que los equipos pueden trabajar de forma independiente y moverse rápidamente acortando el ciclo de desarrollo.

- Se organiza en trono a capacidades empresariales:

  El enfoque de microservicios permite organizar los servicios en torno a las capacidades empresariales, los equipos son multifuncionales y disponen de una gama completa de habilidades necesarias para el desarrollo.



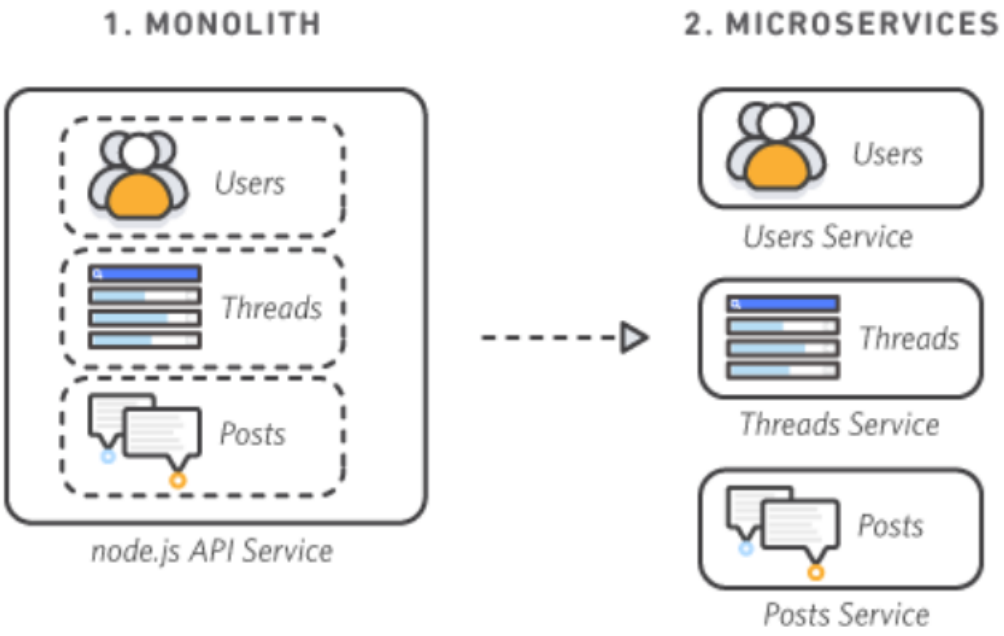
Arquitectura Orientada a Servicios: Los servicios funcionan de forma independiente y proporcionan intercambio de datos o funcionalidades a sus consumidores. El consumidor solicita la información y envía los datos de entrada al servicio. Entonces el servicio procesa los datos, realiza la tareas y devuelve una respuesta.

Un ejemplo es un aplicación que utiliza un servicio de autorización, se le proporciona al servicio el nombre de usuario y la contraseña, el servicio verificará las credenciales y devolverá una respuesta.

Protocolos de comunicación:

Los servicios se comunican mediante reglas que se establecen y que determinan la transmisión de los datos en la red. Estas reglas son llamadas protocolos de comunicación. Algunos estándares para implementar SOA son:

- RESTful HTTP
- Apache THRIFT
- Apache ActiveMQ
- Servicio de mensaje java JMS

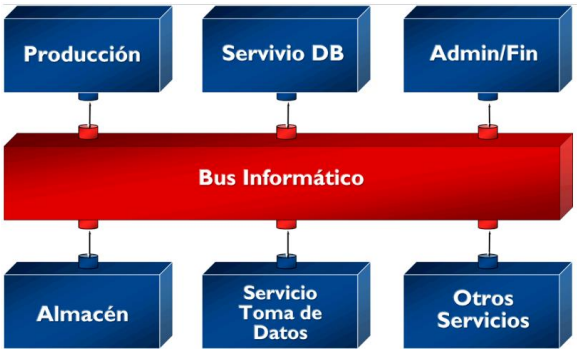


ESB en la arquitectura orientada a servicios:

Un bus de servicio empresarial (ESB) es un software que puede utilizar durante la comunicación con un sistema que tiene varios servicios. Este software establece la comunicación entre los servicios y sus consumidores, de forma independiente de la tecnología. El tener un ESB proporciona capacidades de comunicación y transformación a través de una interfaz de servicio reutilizable. Un ESB se puede considerar como un servicio centralizado que dirige las solicitudes de servicio al servicio adecuado transformando la solicitud al formato aceptable para la plataforma y lenguaje de programación que subyacen en el servicio.

Limitaciones:

- Escalabilidad limitada: se ve afectada cuando los servicios son compartidos por varios recursos y se necesita coordinar para realizar la respectiva funcionalidad.
- Aumenta de las interdependencias: estos modelos pueden volverse cada vez más complejos y desarrollar interdependencias entre servicios, puede llegar a ser complicado modificar o corregir errores si son varios servicios los que se llaman entre sí.

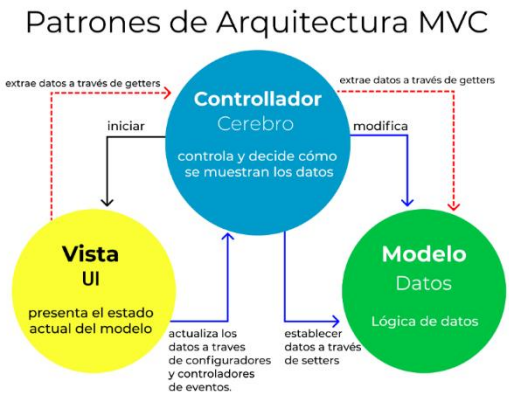


MODELO VISTA CONTROLADOR (MVC):

Este modelo se basa en el patrón Modelo-Vista-Controlador (MVC), divide el desarrollo del sistema en "conceptos", su objetivo es separar lógicamente el negocio de la presentación. Pudiera parecer al N-capas pero con la diferencia al respecto de este, la lógica puede estar presente en todos los componentes y cada componente puede comunicarse con los demás indistintamente, sin tener que pasar por alguno de ellos de manera obligatoria.

Este marco de trabajo es muy popular en frameworks. Este patrón convierte el desarrollo de aplicaciones complejas en un proceso más manejable, permitiendo el trabajo simultaneo en la aplicación.

- Modelo => El backend que contiene la lógica de datos.
- Vista => el Frontend o interfaz gráfica de usuario GUI.
- Controlador => Cerebro de la aplicación que controla como se muestran los datos.



Hoy en día el patrón MVC es utilizado ampliamente en desarrollos Web porque permite que la aplicación sea escalable , mantenible y fácil de expandir. Ayuda a dividir el código frontend y backend en componentes separados, así es más fácil administrar y hacer cambios a cualquiera de los lados sin que se interfieran entre sí.

MODEL VISTA PRESENTADOR (MVP):

Es un patrón arquitectónico utilizado en el desarrollo de software, especialmente en aplicaciones de interfaz de usuario UI. El MVP se basa en el principio de separación de preocupaciones y promueve un código más modular y mantenible. Este patrón proporciona una estructura clara que facilita el desarrollo y la prueba de aplicaciones.

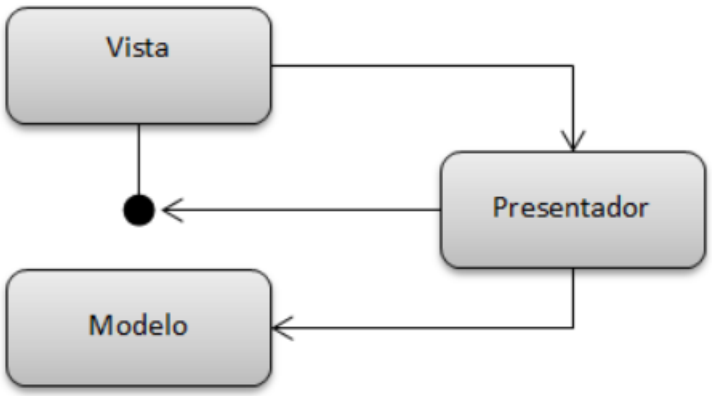
Componentes:

- Modelo: Representa la capa de datos y lógica de negocio de la aplicación. Gestiona el acceso a la base de datos, servicios web, puede incluir lógica para procesar y manipular datos antes de enviarlos a la vista.

- Vista: Es la capa de presentación y se encarga de mostrar la interfaz de usuario. Recibe la información del presentador y actualiza la interfaz de usuario según sea necesario. Contiene la lógica del negocio.
- Presentador: Actúa como intermediario entre el modelo y la vista , recupera datos del modelo y los presenta a la vista de manera adecuada. Maneja las interacciones del usuario y actualiza el modelo según sea necesario.

Características:

- Separación de responsabilidades: MPV facilita la separación clara de responsabilidades entre las capas del modelo, vista y presentador.
- Testabilidad: Debido a la clara separación de responsabilidades, las unidades de código (modelo, vista y presentador) son más fáciles de probar y de forma independiente, se pueden realizar pruebas unitarias en el presentador sin la necesidad de interactuar directamente con la vista o con el modelo.
- Reutilización de código: La modularidad del patrón MVP facilita la reutilización de componentes en diferentes partes de la aplicación o incluso en proyectos diferentes.
- Adaptabilidad a cambios: Dado que el modelo y la vista son independientes entre sí, es más fácil realizar cambios en la interfaz de usuario o en la lógica de negocio sin afectar al otro componente.

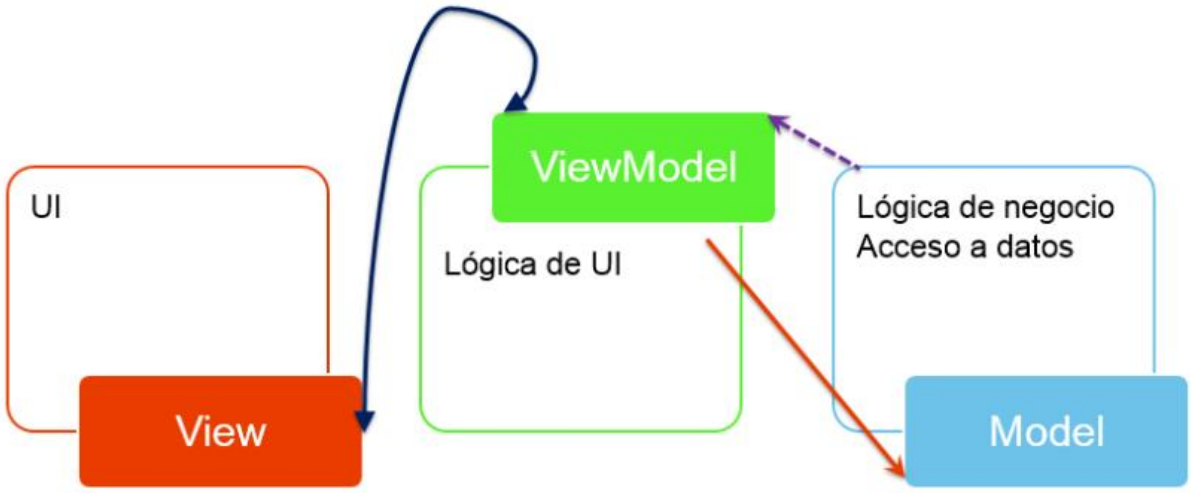


MODELO VISTA – VISTAMODELO (MVVM):

Este patrón ayuda a separar la lógica del negocio de la interfaz de usuario, facilitando las pruebas, mantenimiento y la escalabilidad de los proyectos. MVVM es una forma de crear aplicaciones cliente que aprovecha las características principales de la tecnología WPF.

Consta de tres componentes:

- Model: un modelo representa un objeto de la vida real del dominio de la lógica de negocio del sistema. Ejemplo una factura o un usuario. Es responsable de exponer los datos de una manera que WPF pueda consumir fácilmente.
- View: la vista se define en XAML(lenguaje marcado de WPF) y no debe tener ninguna lógica en el código subyacente. Se une a la ViewModel solo mediante el enlace de datos (binding).
- Viewmodel: Es un modelo de vistas de aplicación, se trata de abstracción de datos de las vistas. Conecta únicamente los datos que requieren las vistas y realiza la lógica necesaria para la preparación de dichos datos. Exponer los datos relevantes para la vista, expone los comportamientos de las vistas.



Este modelo se usa en aplicaciones donde intervengan Silverligth, Windows App, WPF , cómo todo patrón va a facilitar el trabajo en equipo e inclusive el ingreso de nuevos miembros al equipo ya que al conocer cómo funciona las interacciones el MVVM podrá ubicarse más fácilmente.

ARQUITECTURA DE MICROKERNEL:

Un kernel o núcleo, es el nombre que recibe la parte del sistema operativa que se encarga de manejar los procesos que se ejecutan en la computadora y planifica como usar los recursos.

Este tipo de patrón de arquitectura de software se usa en sistemas para manejar errores de software o del hardware de forma independiente, dividiendo los sistemas en secciones de forma tal que si ocurre un fallo no se propague al resto del software.

Ventajas:

- Reducción de complejidad.
- Decentralización.
- Acción depuradora para el trabajo con los controladores de los SO.

Desventajas:

- Sobrecarga de memoria o mal funcionamiento de integraciones.

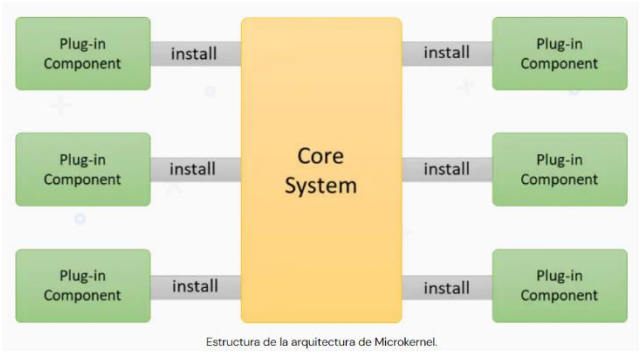
Algunos sistemas operativos que lo implementan:

- amigaOS
- SO3
- MINIX
- AIX
- ChorusOS

Es también conocida como arquitectura plug-in, permite crear aplicaciones extensible mediante la cual es posible agregar nueva funcionalidad mediante la adición de pequeños plugins que extienden la funcionalidad inicial al sistema.

En esta arquitectura las aplicaciones se dividen en dos tipos de componentes:

- Sistema Core : contiene los elementos mínimos para hacer que la aplicación funcione y cumpla con el propósito para el cual fue diseñada.
- Plugins o módulos: añaden o instalan al componente Core para extender su funcionalidad. Puede haber un solo modulo Core y n-cantidad de plugins.

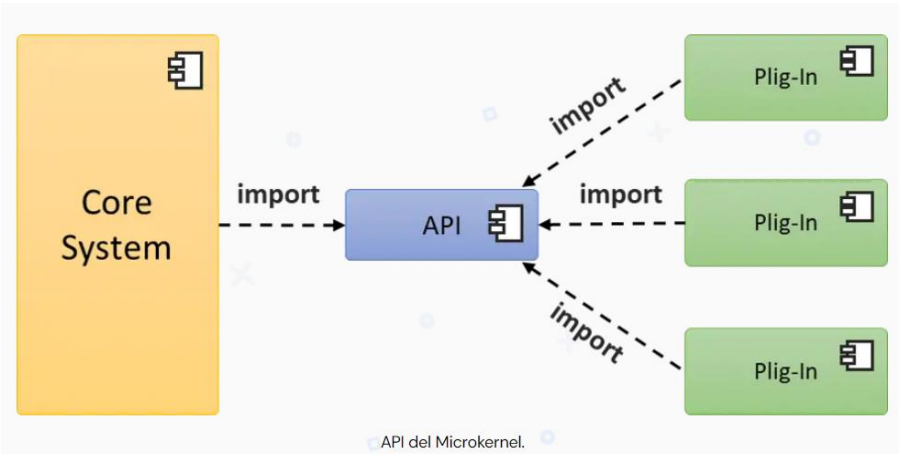


Estructura de la arquitectura de Microkernel.

El objetivo central de este estilo es permitir la extensión de la funcionalidad pero respetando el principio Open-closed, abierto a extender la funcionalidad pero cerrado para modificar su funcionalidad principal.

Los sistemas que utilizan esta arquitectura no son fáciles de desarrollar pues se necesita crear aplicaciones que sean capaces de agrandar dinámicamente su funcionalidad a medida que nuevos plugins son instalados, al mismo tiempo que se debe tener mucho cuidado de que los plugins no modifiquen la esencia de la aplicación. El solo hecho que un sistema acepte plugins es complicado.

Para permitir la construcción de plugins, el Core debe proporcionar API´S o una definición la cual el plugin debe implementar, de esta manera el Core y el API provee para que los desarrolladores ceen plugins.



API del Microkernel.

ARQUITECTURA EN PIZARRA:

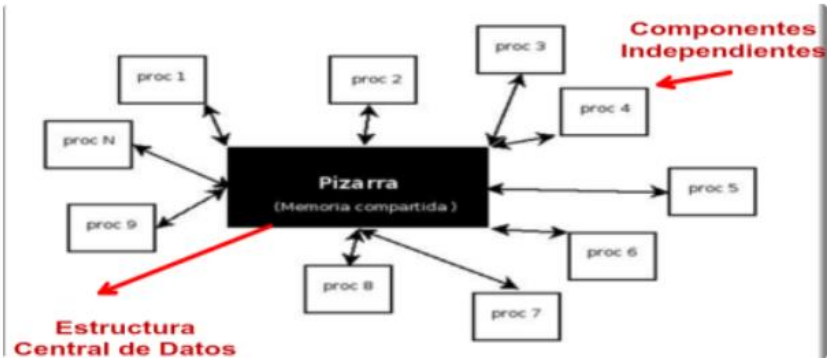
Este patrón sigue el esquema de la pizarra real, la cual tiene un estado inicial relacionado con un problema, y un estado final que es alcanzado cuando se tiene una solución.

La pizarra maneja agentes que se encargan de trabajar de manera independiente:

- La pizarra escribe un conjunto de tareas para los agentes, dirigidas a diferentes áreas dentro del software.
- Cada agente lee en la pizarra la tarea, la ejecuta y escribe resultado en ella.
- Otros agentes pueden utilizar respuestas encontradas para poder resolver su tarea.
- Al final, la pizarra alcanza la solución basada en los resultados de cada tarea.

Es importante destacar que las tareas de los agentes y los agentes en sí, no tiene que poseer ninguna relación. Sin embargo la forma de trabajo debe ser una misma lógica y los resultados deben ser escritos de una forma común definida en el software.

El patrón "blackboard" es habitualmente utilizado en sistemas expertos, multiagentes y basados en conocimiento.



ARQUITECTURA LIMPIA:

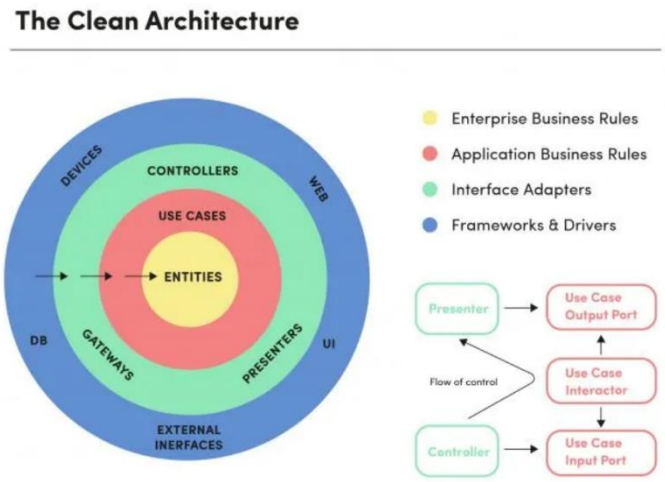
Esta filosofía de diseño fue presentada por Robert C Martin en 2017, básicamente se refiere a la organización de código en componentes o módulos separados y cómo estos elementos se relacionan entre sí.

El objetivo principal de Clean Architecture es brindar una metodología para desarrollar código que funcione mejor, que tenga pocas dependencias y sea fácil de entender y mantener, esto reduce costos y maximiza la productividad del programador.

Este patrón se ha vuelto bastante popular y muchos equipos de desarrollo la están adoptando, principalmente porque ayuda a diseñar aplicaciones con muy bajo acoplamiento y alta cohesión, de manera que las aplicaciones e vuelven más probables y flexibles para cambiar a medida que crece el proyecto.

Características:

- Diagrama de capas circulares concéntricas que recuerdan a The Onion Architecture.
- Las capas internas representan políticas abstractas de alto nivel, mientras la externas son los detalles técnicos de implementación.



En el centro del círculo se tiene el dominio o reglas comerciales, las cuales son la razón del existir de un sistema de software, son la principal funcionalidad de una aplicación.

El código que representa las reglas de negocio deberían ser el corazón del sistema, con preocupaciones menores conectadas a ellas. Las reglas comerciales deben ser el código más independiente e inmutable del sistema.

Cuando más exterior se vaya en el círculo, los elementos se vuelven menos críticos y más propensos a cambios. La presentación y los datos son menos importantes ya que son implementaciones que eventualmente puede ser reemplazadas.

La regla de dependencia:

Esta es la regla principal de esta metodología, y es que las dependencias del código solo pueden provenir de los niveles externos hacia adentro, y las capas internas no deben tener conocimiento de las funciones de las capas externas.

Al asegurarse de que las reglas de negocio y la lógica de dominio estén completamente desprovistas de dependencias externas, logrará lo que se denomina una separación limpia de preocupaciones.

REFERENCIAS BIBLIOGRAFICAS

Patrones de arquitectura de software: tipos y características | Saasradar

Wikipedia. (s.f.). Arquitectura multicapa. En Wikipedia, la enciclopedia libre. Recuperado de https://es.wikipedia.org/wiki/Arquitectura_multicapa

IBM. (s.f.). Three-Tier Architecture. Recuperado de https://www.ibm.com/mx-es/topics/three-tier-architecture

Atlassian. (s.f.). Microservices architecture. Recuperado de https://www.atlassian.com/es/microservices/microservices-architecture

Wikipedia. (s.f.). Arquitectura Orientada a Servicios [Imagen]. Recuperado de https://es.m.wikipedia.org/wiki/Archivo:Arquitectura_Orientada_a_Servicios.png

FreeCodeCamp. (s.f.). El modelo de arquitectura View Controller Pattern. Recuperado de https://www.freecodecamp.org/espanol/news/el-modelo-de-arquitectura-view-controller-pattern/

Reactive Programming. (s.f.). Microkernel. Recuperado de https://reactiveprogramming.io/blog/es/estilos-arquitectonicos/microkernel

Clicko. (s.f.). Patrón diseño MVVM usando WPF (Parte 1). Blog de Clicko. https://blog.clicko.es/patron-diseno-mvvm-usando-wpf-parte-1/

Shirivo. (2015, Junio 30). Modelo, Vista, Vista Modelo… ¿eso es un patrón? Blog de Shirivo. https://shirivo.wordpress.com/2015/06/30/modelo-vista-vista-modelo-eso-es-un-patron/

Nescalro. (s.f.). Entendiendo a la Arquitectura Limpia. Medium. https://nescalro.medium.com/entendiendo-a-la-arquitectura-limpia-7877ad3a0a47