CURSO PROFESIONAL DE ARQUITECTURA DE SOFTWARE

ATRIBUTOS DE CALIDAD

DEFINICIÓN

En el curso de **Fundamentos de Arquitectura de Software** mencionamos los atributos de calidad, en este módulo los estudiaremos a detalle, aprenderemos a medirlos, mejorarlos y detectarlos.

Un ejemplo de atributo de calidad es la seguridad, otro es rendimiento, entonces para definirlos:
Son las expectativas de usuario, en general implícitas, de cuán bien funcionará un producto.

Notas adicionales:

- Son las expectativas de usuario, en general implícitas, de cuan bien funcionará un producto.

- Los atributos tienen identidad en si mismos y son las cualidades de la que todos hablamos, cuando un sistema es bueno o malo en algún aspecto.

  Ejemplo1: Seguridad, aquí podríamos hablar de diferente estrategias o implementaciones que nos ayuden a mejorar la seguridad de un sistema.

  Ejemplo2: Rendimiento, que tan rápido corresponde un sistema, respecto a algún estimulo.

ATRIBUTOS: IDONEIDAD FUNCIONAL

La Idoneidad Funcional es lo que conecta a lo que el usuario quiere hacer (tareas u objetivos a resolver con el sistema) y como están implementadas funcionalmente en nuestro sistema.

Completitud funcional:

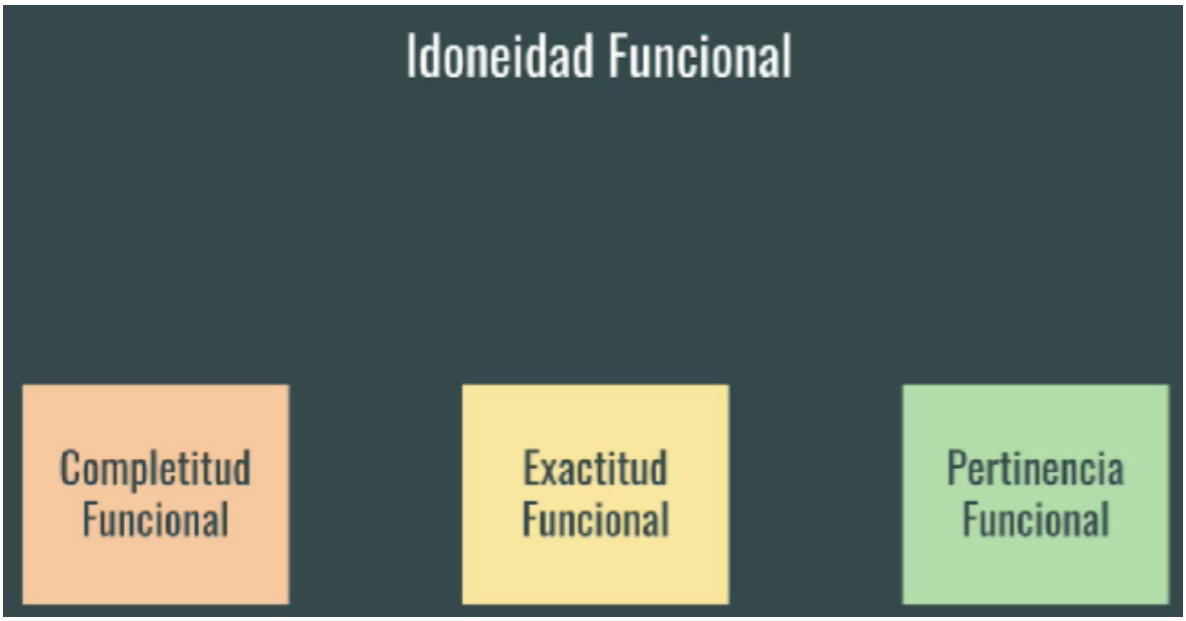
- Cuan completa esta la implementación con respecto a lo que se espera del sistema.
- Requerimientos Funcionales vs Funcionalidades implementadas
- Ejemplo: Login con redes sociales, podríamos hacerlo incrementalmente (empezamos con una red social y luego vamos agregando otras). Entonces el usuario no sentiría que está "completo" hasta que no tenga la posibilidad de iniciar sesión hasta que no tenga todas las redes sociales disponibles.

Exactitud funcional:

- Cuan preciso es el sistema para implementar lo requerido.
- Resultados Esperados vs Resultados Obtenidos
- Ejemplos: resultados numéricos, grado de éxito/fracaso, éxito/no éxito

Pertinencia funcional:

- Cuan alineado esta lo que se implementó con lo requerido.
- Objetivos Cumplidos vs Objetivos Esperados
- Ejemplo Aplicaciones CRUD: suelen considerar cada entidad o concepto del problema como un elemento al que se lo puede crear, eliminar, modificar o leer. Esto le permite a frameworks y librerías el brindar herramientas de desarrollo rápido para estas acciones genéricas, pero a medida que nos encontramos con la complejidad esencial de nuestro problema, los ABMC suelen ser incompletos y difíciles de usar porque, al final, delegan la complejidad en el usuario que sabe qué tiene que crear o modificar para que su problema se resuelva.



ATRIBUTOS: EFICIENCIA DE EJECUCIÓN

Este es otro atributo de calidad, se trata de cuan bueno o eficiente es el sistema, a la hora de responder a lo que el usuario necesita. y asu vez teniendo los recursos del sistema, cuan bueno es que los aprovecha o desaprovecha.

Tiempo a Comportamiento: Este nos dice cuan bueno es el sistema respondiendo al usuario; Específicamente cuanto tarde el sistema y cuanto esperamos que ese sistema tarde.
Se mide: a traves de la medición de la respuesta, dado un estímulo y luego compararlo con el tiempo que esperamos que tarde el sistema.

Uso de Recursos: Cuanto el sistema aprovecha esos recursos en su contexto, sea la RAM, el SO, ETC. el objetivo es saber cuan bien o mal se están usando.
Se mide: medir cuanto ocupa el sistema del RAM o CPU y si soporta o tiene la capacidad entera de recursos para soportar.

Capacidad: Cuanto soporta el sistema en cantidad de pedidos, es decir tiene un límite en la cantidad de una cantidad máxima de usuarios usando el sistema a la vez.
Se mide: Cuanto esperamos que el sistema responda en ciertos casos, por ejemplo, en carga, escritura o lectura y luego de saber cuánto esperamos que responda. probar cuanto responde realmente.

ATRIBUTOS: COMPATIBILIDAD

Este agrupa los atributos de cuanto el sistema coexiste o interopera con otros sistemas, es decir; **Cuanto puede el sistema vivir en un contexto más grande.**

Interoperabilidad: Cuan fácil es comunicarse con este sistema, cuanto define su sistema de comunicación, incluso cuanto después este sistema puede comunicarse con otros.
Se mide: A través de casos puntuales, por ejemplo, una plataforma de pagos. que tan fácil o difícil es que nuestro sistema se integre son dicha plataforma.

Coexistencia: Esta dada por cuanto el sistema soporta o no el estar en un contexto dado con otro sistema también. ¿El sistema puede coexistir dentro del mismo servidor, la misma red? ¿Eso genera fallos al sistema?
Se mide: A través de la cantidad de fallos que tenemos, sin que esos fallos sean generados por nuestra aplicación. Es decir, si nuestra aplicación convive con otras y estas otras nos quita recursos o porque la aplicación le genera un fallo de segmentación en el sistema operativo. (Cualquier cosa externa).

ATRIBUTOS: USABILIDAD

- **Reconocimiento de Idoneidad:** Cuanto nos damos cuenta de que un sistema es lo que nosotros necesitamos usar para resolver nuestro objetivo. Para medirlo necesitamos saber si el dominio de nuestro problema está asociado con el sistema.

  Ejemplo: Wordpress.

- **Curva de Aprendizaje:** Es que tan fácil o difícil es aprender a usar el sistema. Se mide por la cantidad de ayuda que necesitemos para poder hacer las tareas.

  Ejemplo: Lenguaje de Gestos.

- **Operabilidad:** Que cantidad de pasos o esfuerzo hay que hacer para cumplir un objetivo.

  Ejemplo: Evitar formularios largos.

- **Protección de Errores:** Es como el sistema le comunica al usuario de cuantas veces se equivocó (Una manera de darle feedback al usuario).

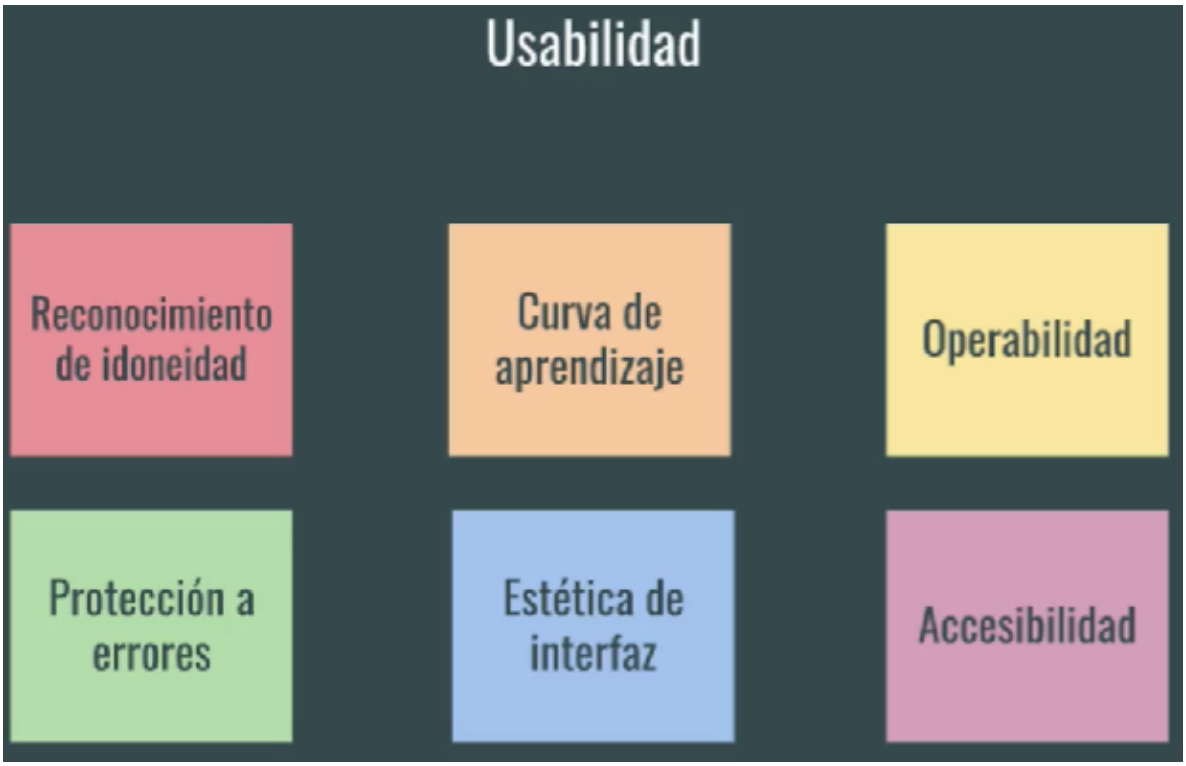
  Ejemplo: Ser más específico con el feedback.

- **Estética de Interfaz:** La mejor manera de medirlo es preguntarles a los usuarios a través de encuestas subjetivas.

  Ejemplo: En caso tal sea una prima se deben considerar puestos como los de UI y UX.

- **Accesibilidad:** Se trata de cuan permisible es el sistema de utilizar para personas con discapacidades. Esto es muy difícil de medir, ya que se necesita experiencia por el usuario.

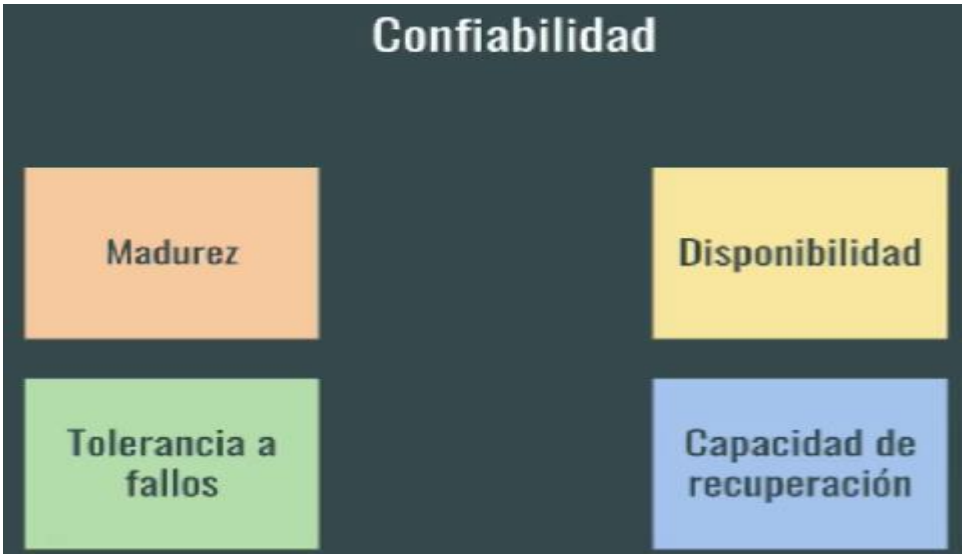
  Ejemplo: Utilizar Propiedades del desarrollo para tener mayor inclusión con las personas con discapacidad.


Usabilidad
Reconocimiento de idoneidad
Curva de aprendizaje
Operabilidad
Protección a errores
Estética de interfaz
Accesibilidad

ATRIBUTOS: CONFIABILIDAD

Se trata de cuanto el sistema nos permite utilizarlo a través del tiempo de forma normal.

- **Madurez:** Cuanto falla el sistema. Para medirla, se toma el tiempo entre cada fallo que haya tenido el sistema. Cuanto más tiempo pase, más maduro es el sistema. Ejemplo, sistemas bancarios.
- **Disponibilidad:** Cuanto tiempo esta fuera de servicio el sistema con respecto a su ciclo de vida normal. Para medirlo, igualmente tomamos el tiempo que estuvo fuera y lo expresamos en una forma de porcentaje. Ejemplo, los contratos o acuerdos.
- **Tolerancia a fallos (Resiliencia):** Como el sistema se mantiene dando el servicio a pesar de que tenga un fallo o haya un fallo con la conexión a un sistema externo. Para medirlo hay que generar los fallos y ver cómo se comporta. Ejemplo, aplicaciones móviles en algunos países donde se genera time out.
- **Capacidad de recuperación:** Cuanto tiempo el sistema puede seguir estando disponible, luego de algún fallo. Para medirlo, guardamos el tiempo que el sistema vuelve a dar el servicio una vez que salió por un fallo. Ejemplo, aplicaciones distribuidas como AWS, Docker, entre otros.



ATRIBUTOS: SEGURIDAD

Cuanto el sistema puede proteger, saber identificar y conectar a sus usuarios. y al conectar usuarios con información.

- **Confidencialidad:** El sistema debe saber, quienes tienen permiso a que información.
- **Integridad:** Cuanto el sistema toma recaudo para proteger esa información de atacantes o usuarios que no deberían tener acceso a ella. Ejemplo, sistemas bancarios.
- **Comprobación de Hechos:** hablamos de territorio legal, de cómo hacer para que el sistema garantice de que algo paso.
- **Traza de Responsabilidad:** Esta conectada a la comprobación de hechos, pero esta más dada a como se conecta una acción del sistema con su responsable. Sea este el usuario o el sistema mismo.
- **Autenticidad:** Verifica que el usuario que está tratando de ingresar al sistema o realizar una acción, sea quien dice que ser. Ejemplo, aplicaciones web, móviles, sistemas gubernamentales.

ATRIBUTOS: MANTENIBILIDAD

Nos referimos a todas esas cosas que hacen que un sistema pueda cambiar. Pueda evolucionar y a su vez pueda ser reparado.

Modularidad. Habla de la capacidad de un sistema en ser separado en partes dónde cada una de esas partes sea independiente de las otras. Ejemplo, aplicaciones redistribuidas.

Reusabilidad. Es una característica que habla sobre cuánto podemos aprovechar el esfuerzo que hicimos en desarrollar un módulo o una pieza de software y reutilizarla en otro lado, es decir, volver a usarla para otro propósito diferente o para una funcionalidad ligeramente diferente. Ejemplo, Código Open Source.

Capacidad de análisis. Conexión entre el código y los requerimientos. Ejemplo, desarrollo orientado al requerimiento.

Capacidad de modificación. Cuán fácil o difícil es ir al código y cambiar el comportamiento. Ejemplo, test automatizados.

Capacidad de prueba. Habla sobre cuán fácil o difícil es crear estas pruebas para que el sistema garantice que hace lo que se requiere que haga. Para tener una mejor capacidad de prueba tenemos que darle más importancia a nuestra estructura del código, a nuestras operaciones y cuán atómicas son y cuán independientes son unas de otras. Ejemplo, diseñas acciones de métodos de objetos que sean puros, que no tengan efectos colaterales.

ATRIBUTOS: PORTABILIDAD

- **Adaptabilidad:** La podemos medir analizando cuán fuertemente depende nuestro sistema de un entorno especifico.

- **Capacidad de Instalación:** La podemos medir analizando cuán fuertemente necesitamos requerimientos en el el entorno de despliegue.

- **Capacidad de Reemplazo:** La podemos medir conociendo los nuevos requerimientos o facilidades que hay hoy y configurarlo a favor de nuestro sistema.

TENSIONES ENTRE ATRIBUTOS

Se debe enfocar a dos máximo tres atributos ya que el intentar mantener todos puede tender al fracaso del proyecto, esto no es viable, por lo cual se debe priorizar en base a los requerimientos (funcionales, no funcionales, del proyecto, del negocio) cuales son los atributos que más valor aportan al sistema. Si de da más valor a un atributo, perderás valor en otro.

	Disponibilidad	Uso de recursos	Capacidad de instalación	Integridad	Interoperabilidad	Capacidad de modificación	Eficiencia de ejecución	Portabilidad	Madurez	Reusabilidad	Robusteza	Protección de errores	Escalabilidad	Seguridad	Usabilidad	Capacidad de prueba
Disponibilidad									+		+					
Uso de recursos	+				-	-	+	-			-		+		-	
Capacidad de instalación	+								+					+		
Integridad			-		-		-			-		+		+	-	-
Interoperabilidad	+		-	-			-	+	+		+	-		-		
Capacidad de modificación	+		-				-		+	+			+			+
Eficiencia de ejecución		+			-	-		-			-		-		-	
Portabilidad		-			+	-	-			+				-	-	+
Madurez	+	-		+		+	-				+	+		+	+	+
Reusabilidad		-		-	+	+	-	+						-		+
Robusteza	+	-	+	+	+		-		+			+	+	+	+	
Protección de errores		-		+	+		-				+			+	-	-
Escalabilidad	+	+		+			+	+	+		+					
Seguridad	+			+	+		-	-	+		+	+			-	-
Usabilidad		-	+				-	-	+		+	+				-
Capacidad de prueba	+		+	+		+			+	+	+	+		+	+	

https://medium.com/analysts-corner/those-other-requirements-quality-attributes-and-their-inescapable-tradeoffs-31dc0691974d

ANALIZANDO PLATZISERVICIOS

- Cuando mas grande es el sistema mas capacidad de prueba debe tener para asegurar que ningún cambio rompe funcionalidades anteriores.
- A medida que avanza un proyecto se pueden ir usando los atributos de calidad para tomar mejores decisiones de diseño a la hora de implementar una solución.
- No hay que implementar todos los atributos a la vez pero una vez que se vayan usando hay que vigilar que no se genere tensión entre los atributos y que un debilite a otro.

Notas adicionales:

Aplicando atributos de calidad en base a la madurez del proyecto

Hay que priorizar los atributos y atacarlos uno a uno.

Startup

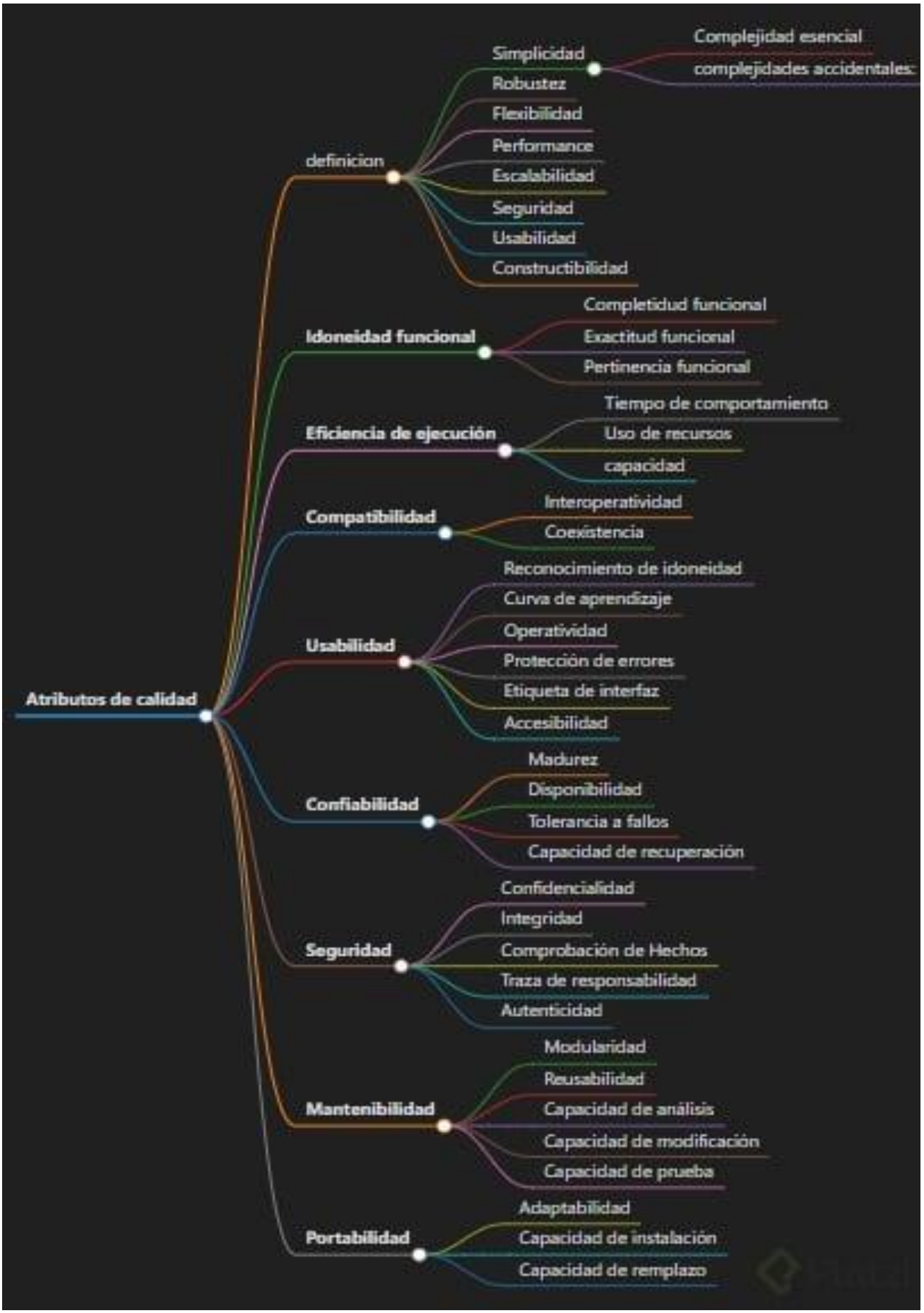
- Confiabilidad: Madurez y disponibilidad
- Seguridad: Autenticidad y confidencialidad
- Compatibilidad: Interoperabilidad

En crecimiento

- Eficiencia de ejecución: Uso de recursos y capacidad
- Compatibilidad: Interoperabilidad
- Seguridad: Comprobación de hechos, traza de responsabilidad, confidencialidad

Gran escala

- Usabilidad: Accesibilidad, reconocimiento de idoneidad, operabilidad
- Mantenibilidad: Modularidad, capacidad de prueba, capacidad de modificación
- Confiabilidad: Tolerancia a fallos, capacidad de recuperación


Atributos de calidad
definicion
Simplicidad
Complejidad esencial
complejidades accidentales:
Robustez
Flexibilidad
Performance
Escalabilidad
Seguridad
Usabilidad
Constructibilidad
Idoneidad funcional
Completidud funcional
Exactitud funcional
Pertinencia funcional
Eficiencia de ejecución
Tiempo de comportamiento
Uso de recursos
capacidad
Compatibilidad
Interoperatividad
Coexistencia
Usabilidad
Reconocimiento de idoneidad
Curva de aprendizaje
Operatividad
Protección de errores
Etiqueta de interfaz
Accesibilidad
Confiabilidad
Madurez
Disponibilidad
Tolerancia a fallos
Capacidad de recuperación
Seguridad
Confidencialidad
Integridad
Comprobación de Hechos
Traza de responsabilidad
Autenticidad
Mantenibilidad
Modularidad
Reusabilidad
Capacidad de análisis
Capacidad de modificación
Capacidad de prueba
Portabilidad
Adaptabilidad
Capacidad de instalación
Capacidad de remplazo

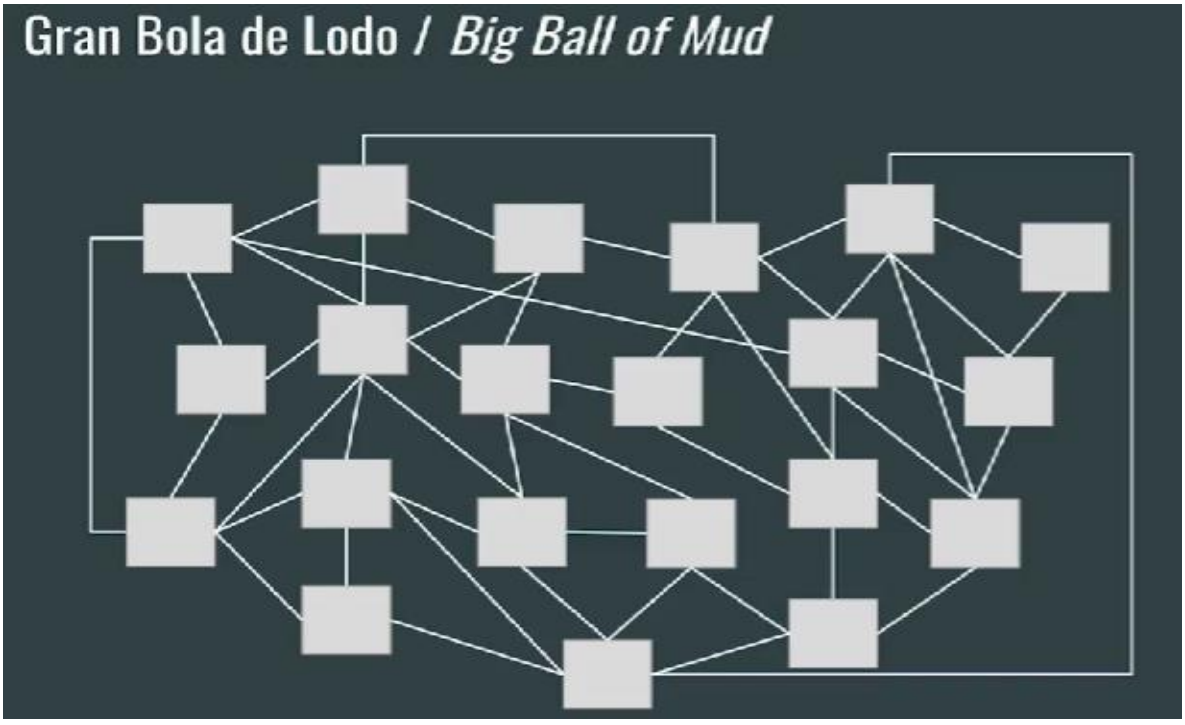
PATRONES DE ARQUITECTURA

PATRONES MONOLÍTICOS VS DISTRIBUIDOS

¿Qué es un patrón de Arquitectura?
Decisiones de diseño ya tomadas para generar un esquema, estructura o tipo de comunicación entre componentes.

- Monolíticos. Artefacto resultante se despliega como una sola unidad
- Distribuidos. Arquitecturas distribuidas, Cada componente se puede desplegar independientemente.
  Cada componente del patrón distribuido es un componente monolítico o lo que es igual a *Los patrones distribuidos son el conjunto de patrones monolíticos*

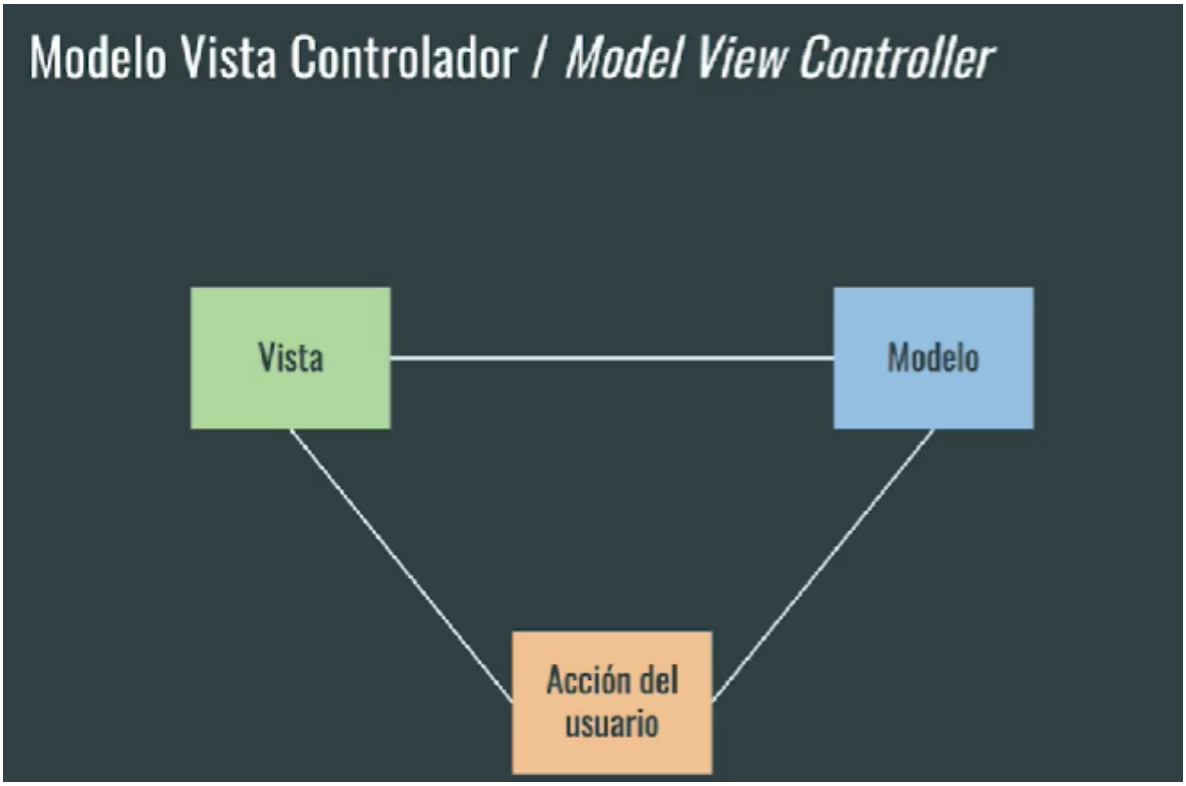
Patrón Gran Bola de Lodo / Big Ball of Mud
Surge cuando un equipo no considera la arquitectura como algo relevante. No hay criterio. El sistema es un caos.



PATRONES: MODELO VISTA CONTROLADOR

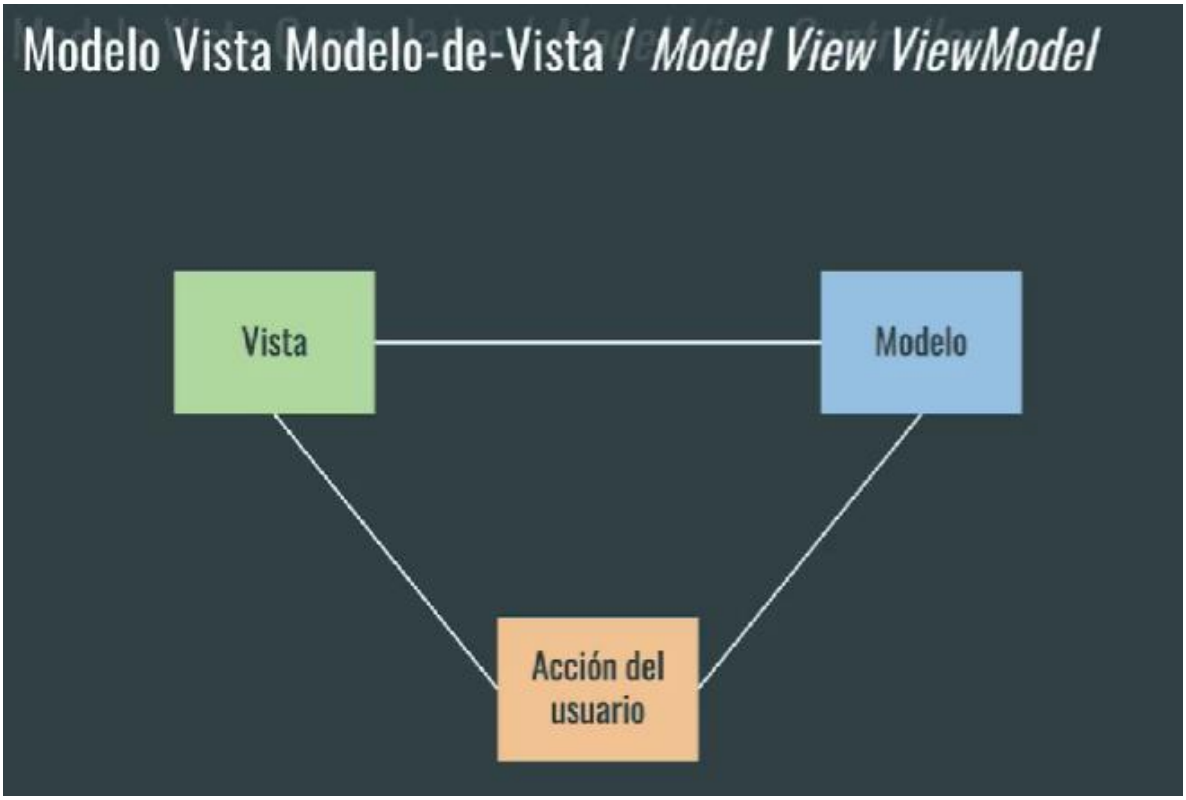
Modelo Vista Controlador (MVC), el cual establece la **separación de responsabilidades** de la siguiente manera:

- Modelo: El estado del sistema
- Vista: La presentación
- Controlador: Las acciones del usuario.

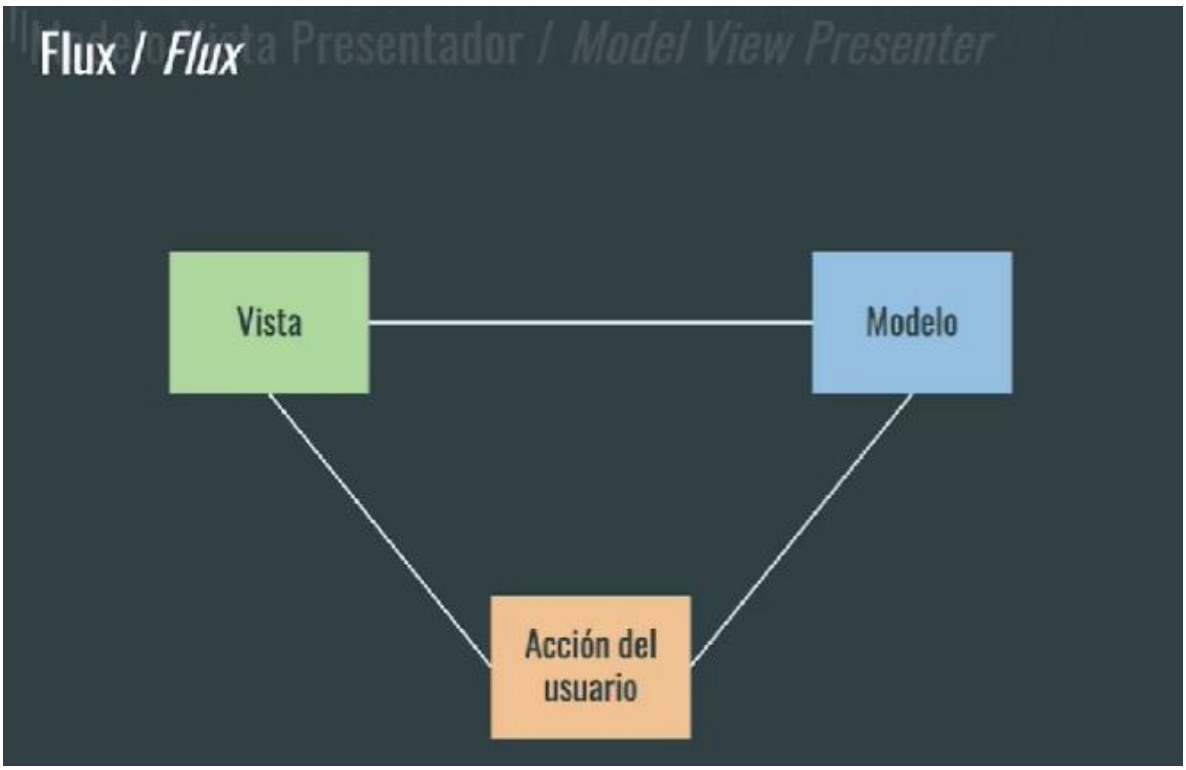


Algunas variantes de este modelo son:

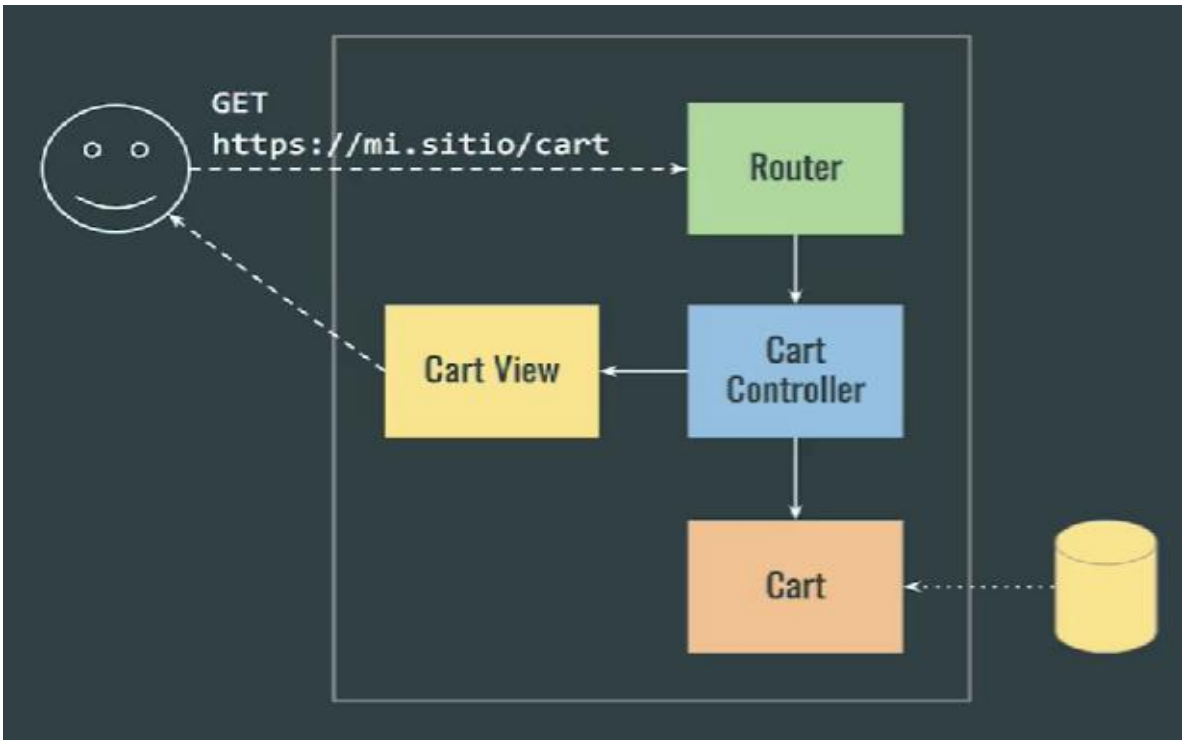
Modelo Vista VistaModelo (MVVM) Donde el **ViewModel** contiene toda la lógica de presentación. Permite abstracción de lógica de la vista. Ejemplo: C# y ASP.NET.



Modelo Vista Presentador (MVP): Donde el **Presenter** lleva toda la lógica de presentación. Es un coordinador o intermediario, es quien va a mediar entre la vista y el modelo, sin que estos interactúen entre sí.
Flux: Maneja el flujo de datos (Cómo fluyen los datos). Aquí se maneja la comunicación en una sola vía. Se elimina la comunicación bidireccional que había entre modelo y controlador. Se deja de lado esa comunicación triangular del MVC.



Ejemplo mas común en aplicaciones web:



PATRONES: CAPAS

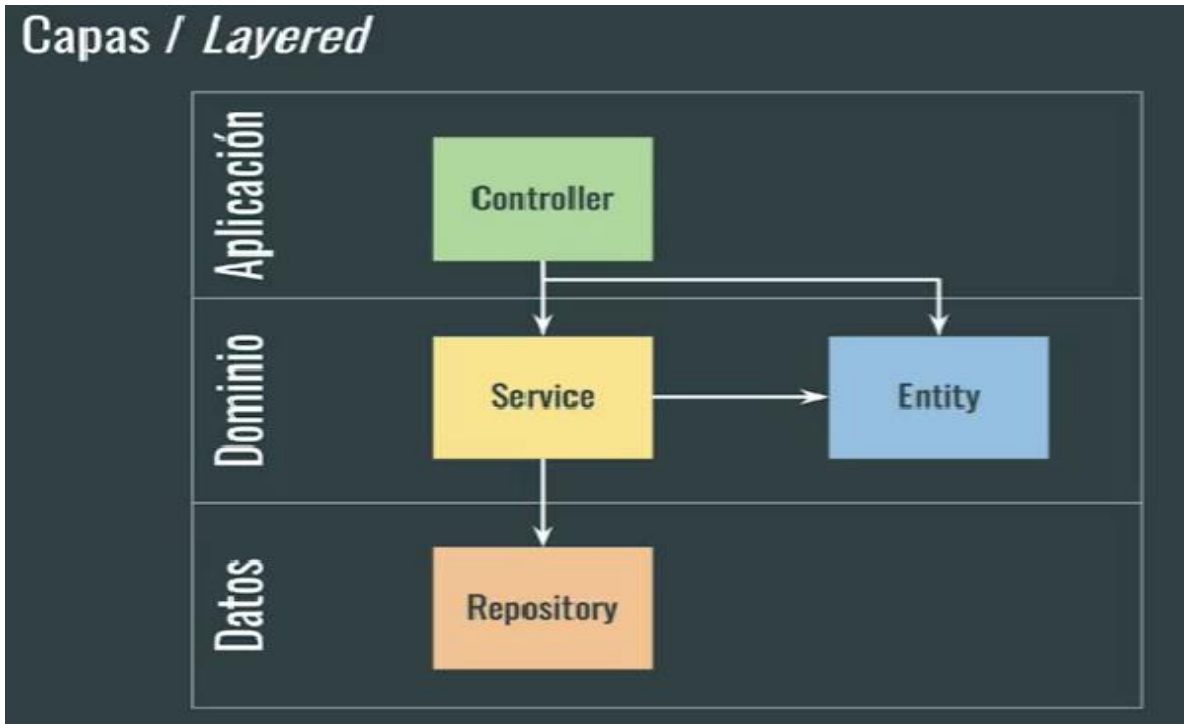
- Separación en capas, donde cada una es responsable de cierto concepto global de la aplicación.

- La cantidad de capas depende de cada aplicación. Es común ver 3 o 4 capas. La comunicación siempre debe ser de arriba hacia abajo.

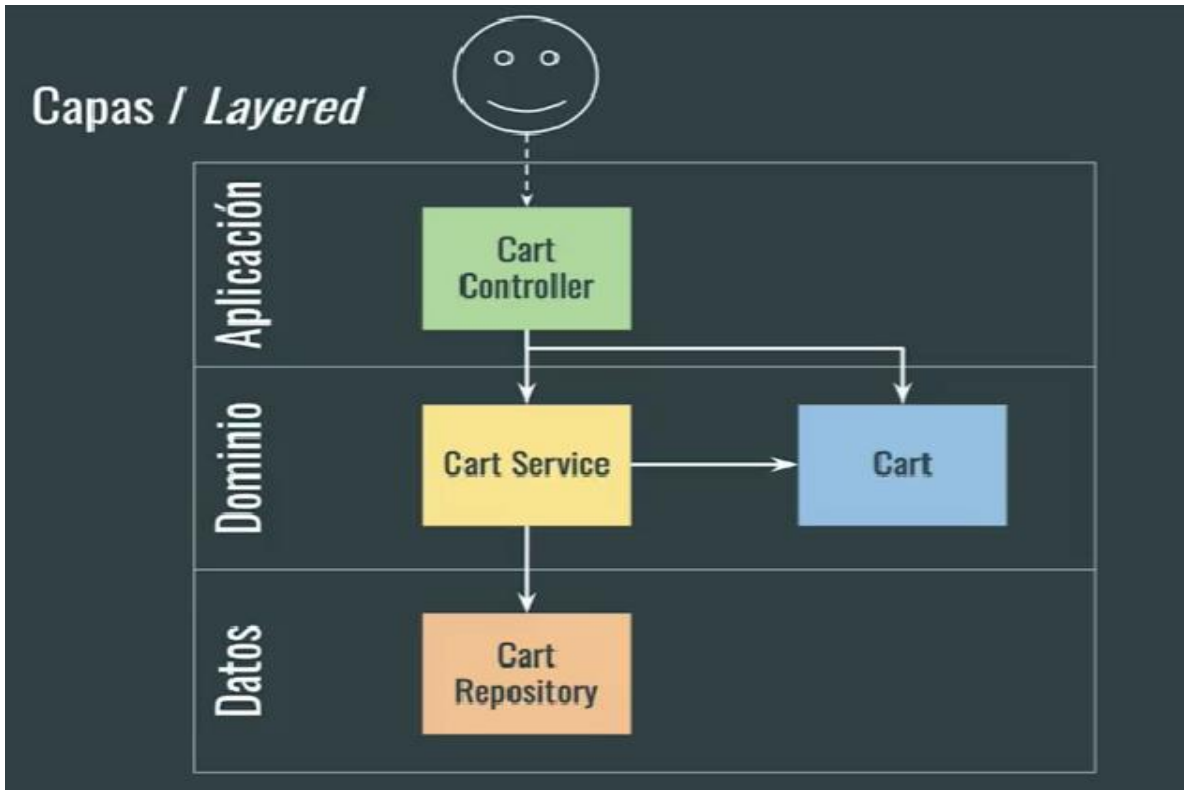
- La arquitectura en capas se implementa en un monolito. La arquitectura resultante se despliega toda junta.

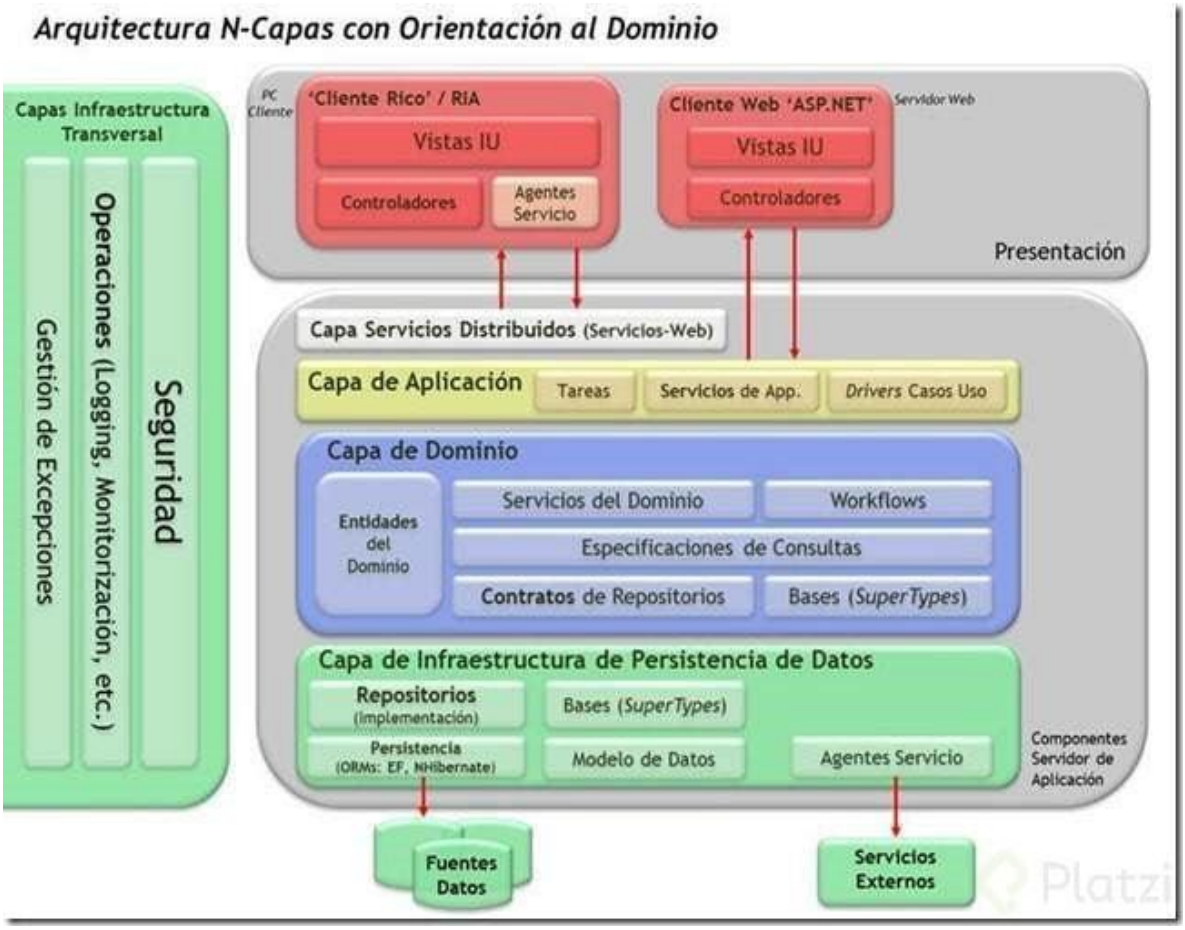
Las 4 capas más comúnmente encontradas de un sistema de información general son las siguientes.

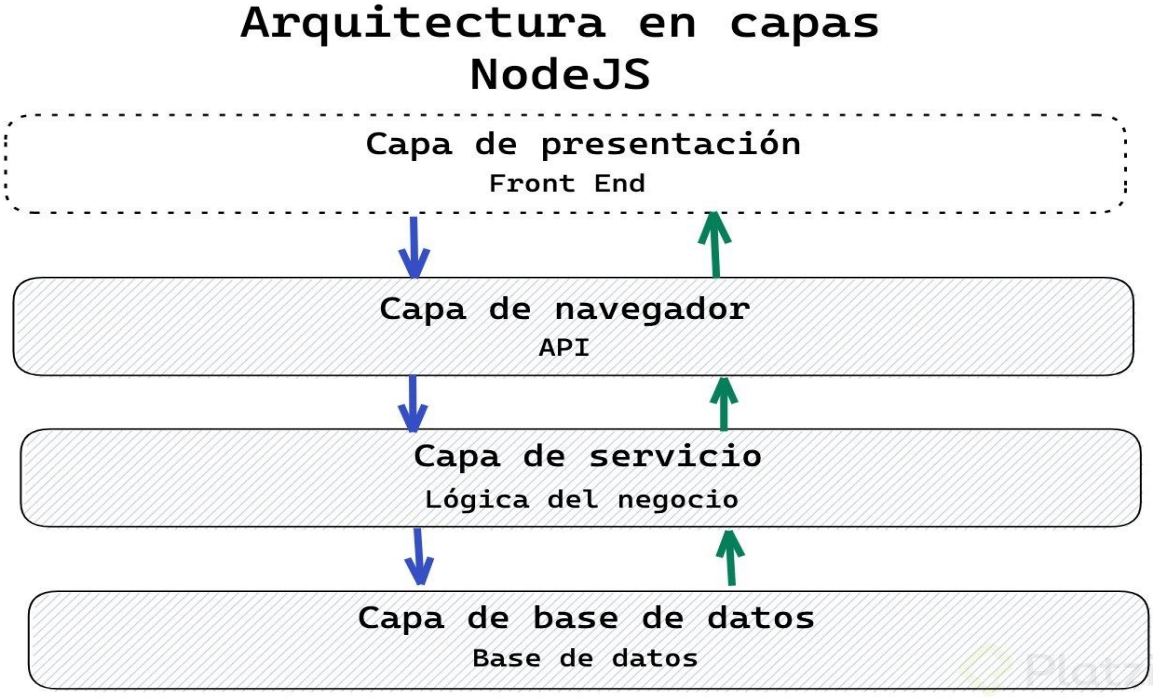
- Capa de presentación (también conocida como capa UI)
- Capa de aplicación (también conocida como capa de servicio)
- Capa de lógica de negocios (también conocida como capa de dominio)
- Capa de acceso a datos (también conocida como capa de persistencia)



Ejemplo de una aplicaciones web con peticiones HTTP:



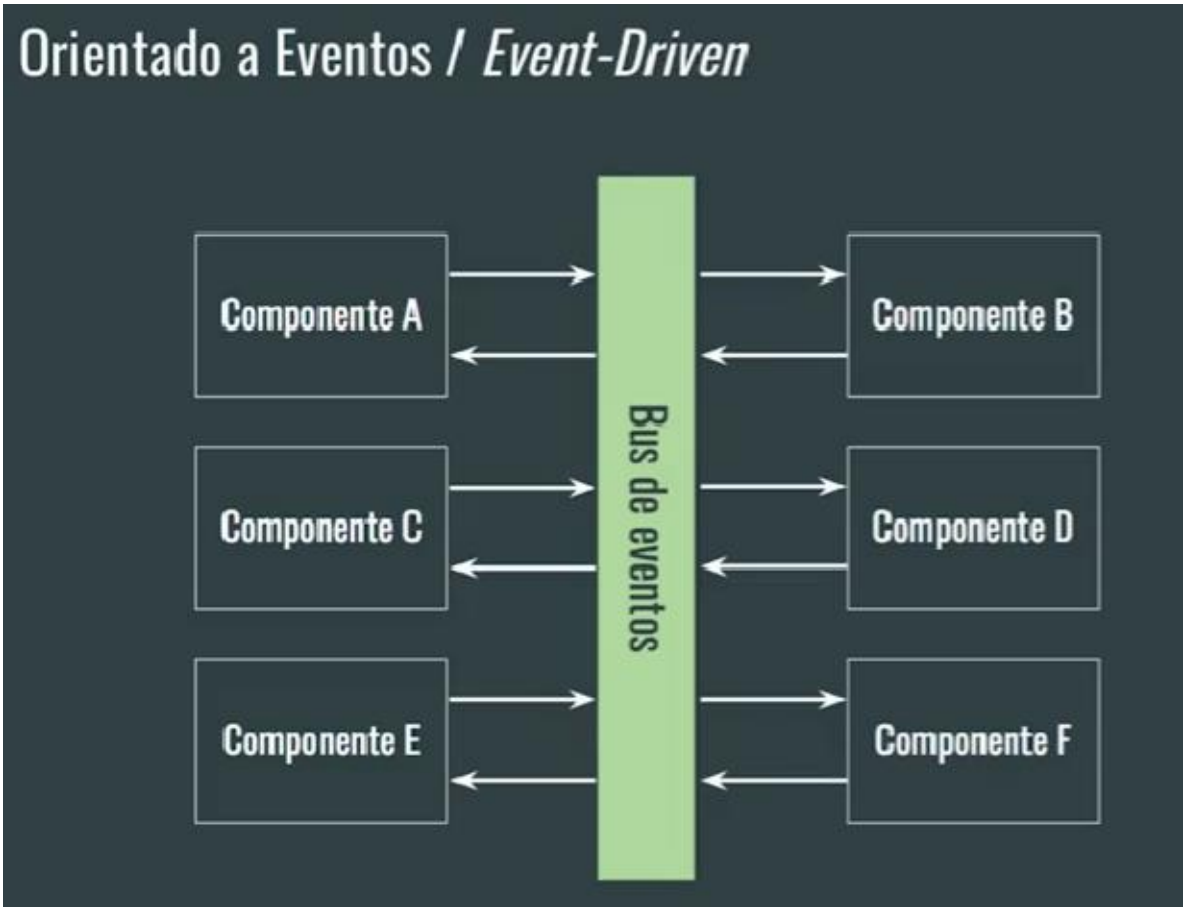

Arquitectura N-Capas con Orientación al Dominio
Capas Infraestructura Transversal
Gestión de Excepciones
Operaciones (Logging, Monitorización, etc.)
Seguridad
PC Cliente
'Cliente Rico' / RIA
Vistas IU
Controladores
Agentes Servicio
Cliente Web 'ASP.NET'
Servidor Web
Vistas IU
Controladores
Presentación
Capa Servicios Distribuidos (Servicios-Web)
Capa de Aplicación
Tareas
Servicios de App.
Drivers Casos Uso
Capa de Dominio
Entidades del Dominio
Servicios del Dominio
Workflows
Especificaciones de Consultas
Contratos de Repositorios
Bases (SuperTypes)
Capa de Infraestructura de Persistencia de Datos
Repositorios (Implementación)
Bases (SuperTypes)
Persistencia (ORMs: EF, NHibernate)
Modelo de Datos
Agentes Servicio
Componentes Servidor de Aplicación
Fuentes Datos
Servicios Externos
Platzi


Arquitectura en capas
NodeJS
Capa de presentación
Front End
Capa de navegador
API
Capa de servicio
Lógica del negocio
Capa de base de datos
Base de datos
Platzi

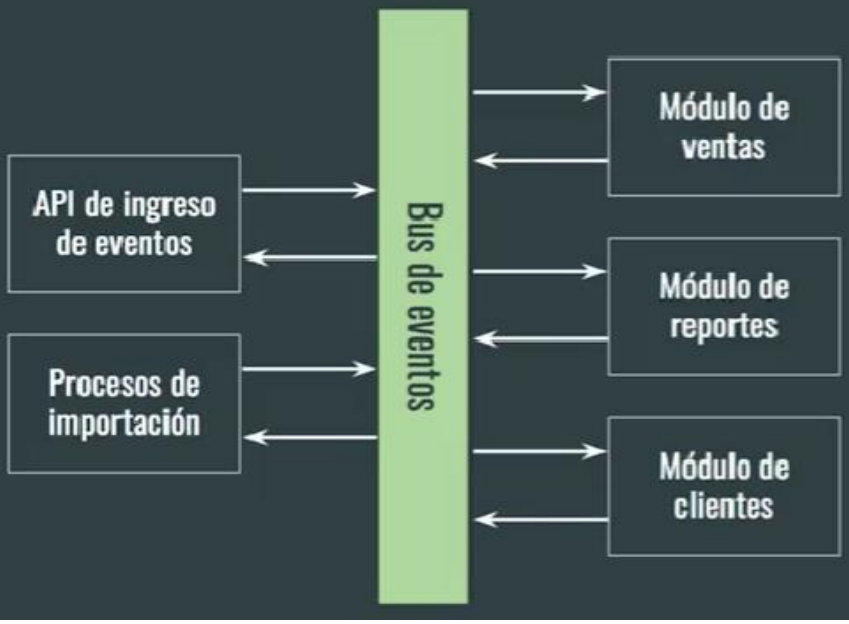
PATRONES: ORIENTADO A EVENTOS / PROVISIÓN DE EVENTOS

Orientado a eventos. Trata sobre cómo conectar componentes a través de eventos. Cada componente publica eventos a un bus de eventos común y los componentes interesados a estos eventos pueden estar suscritos y luego responder al respecto. No hay otra forma de comunicación, el bus de eventos pasa ser el método principal de comunicación entre componentes. Algo complejo es saber si una acción que hicimos tuvo el resultado que esperábamos, en general suelen ser eventualmente consistentes, lo que significa que cuando hacemos una escritura el sistema no nos garantiza que va a estar disponible hasta que ese evento no se distribuya en todas las partes que lo necesita.

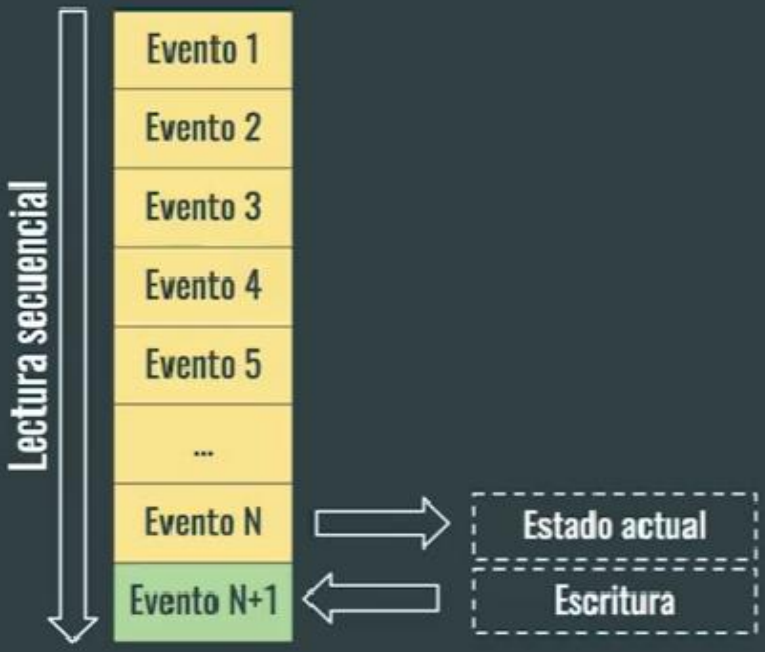
Provisión de eventos. En vez de que nuestra aplicación tenga el estado actual del sitio, podríamos tener solamente guardados los eventos que nos importan.



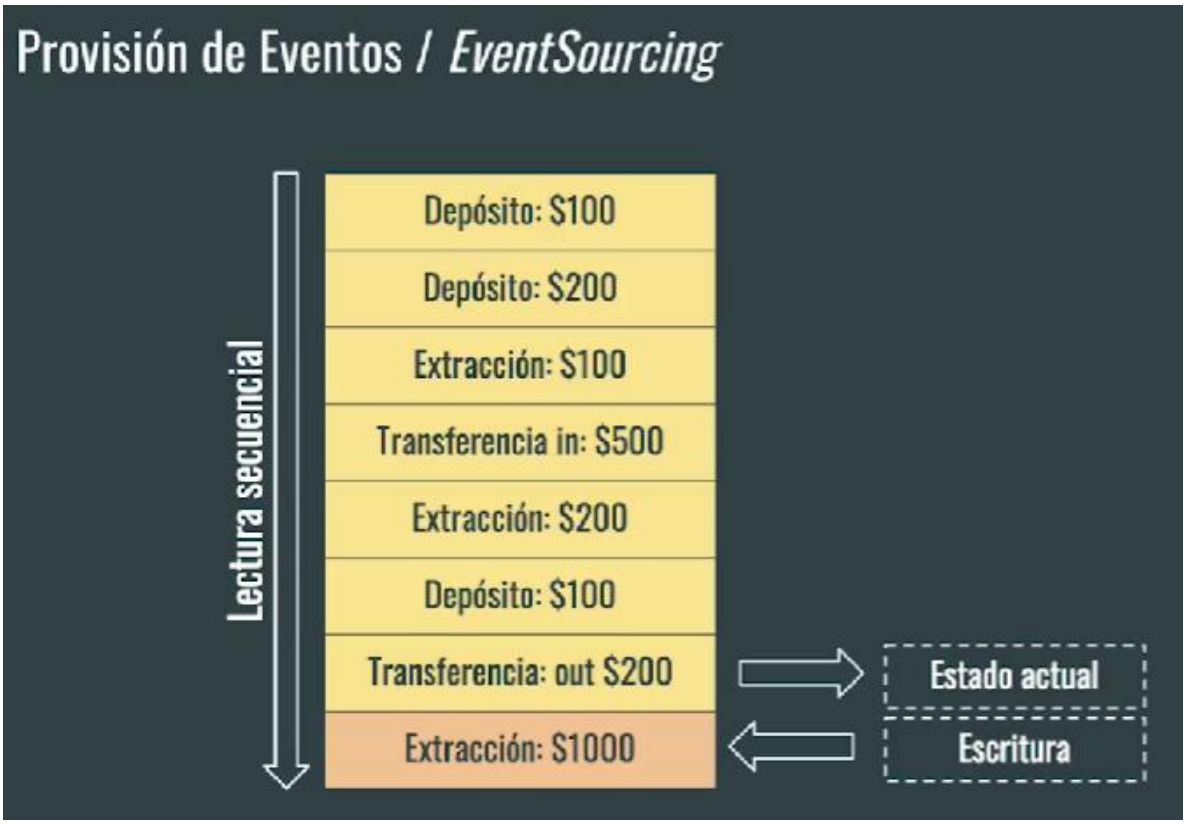
Orientado a Eventos / *Event-Driven*



Provisión de Eventos / *EventSourcing*



Ejemplo de transacciones bancarias:



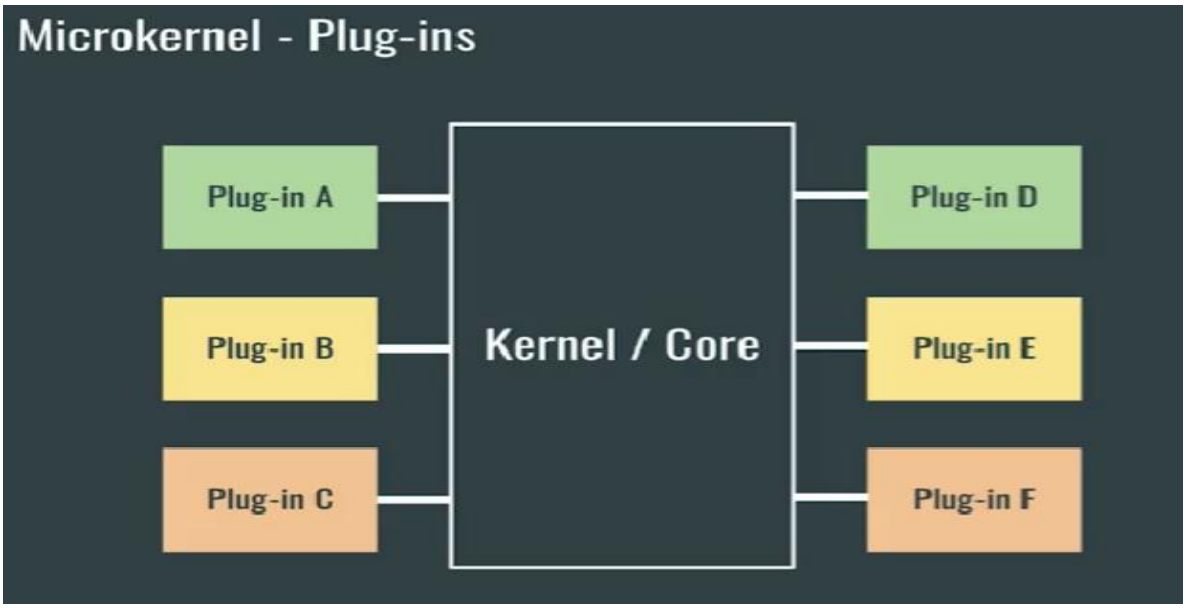
Siempre se debe estudiar el caso de uso y evaluar si es el adecuado para el desarrollo de mi solución, ya que **cada patrón tiene sus pros y contras**

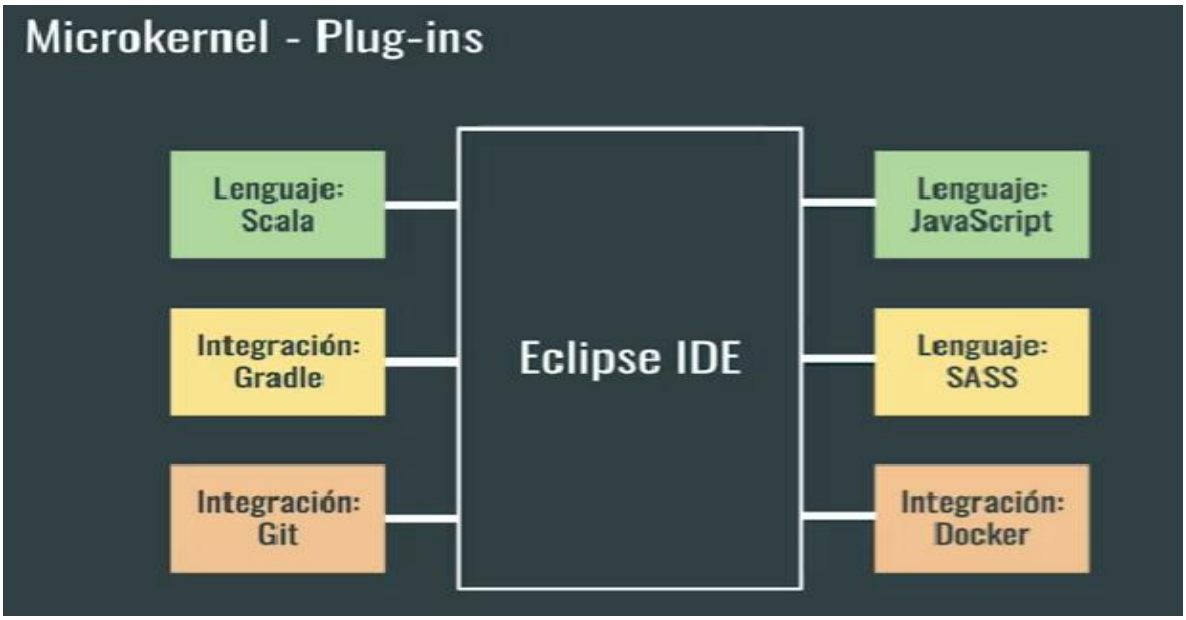
En el caso del patrón orientado a eventos:

Pro: permite desligar un sistema distribuido mediante el bus, logrando así mayor capacidad a la hora de escalar mi sistema distribuido.
Contra: Mayor dificultad a la hora de hacer pruebas end to end.
Contra: El mismo bus de eventos podría ser un punto de falla. Se debe contar con buenas capacidades para soportar el nivel de transaccionalidad del bus de eventos.

PATRONES: MICROKERNEL - PLUG-INS

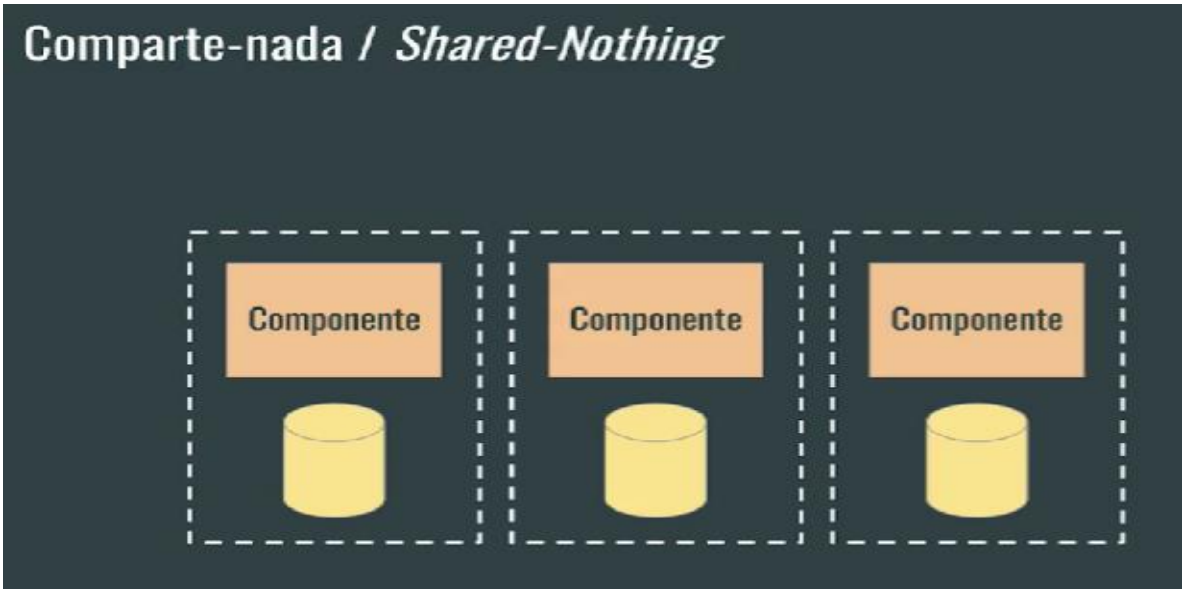
Como hacer para tener un corazón (Kernel / Core) y diferentes puntos de conexión a la aplicación que puedan ser incorporados o quitados dinámicamente.
Puede verse como un patrón monolítico si va con sus plug ins ya desplegados también. O como distribuida si sus componentes fueron cambiados en tiempo de ejecución.

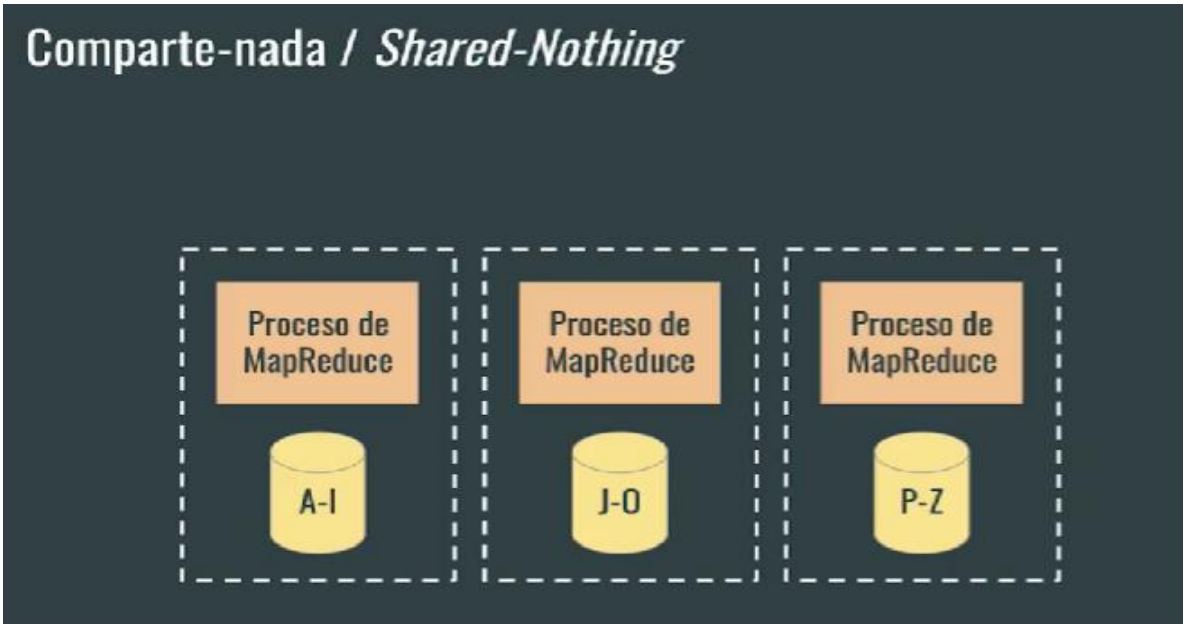




PATRONES: COMPARTE-NADA

Patrón de arquitectura **Comparte-nada**
Como hacer para compartir recursos?
El compartir recursos entre componentes agrega complejidad a la hora de decidir acciones. Por lo tanto, esta arquitectura plantea como **no necesitar un punto de unión entre componentes.**



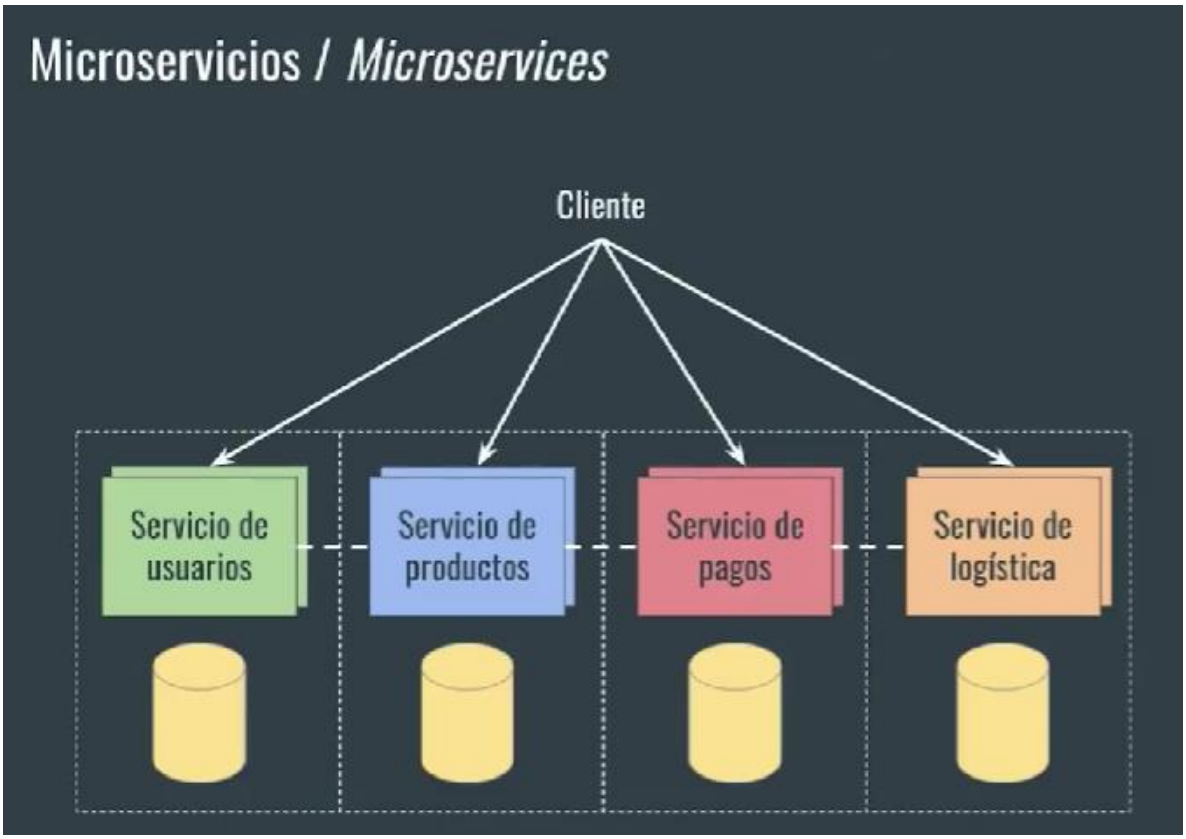


PATRONES: MICROSERVICIOS

Son componentes distribuidos en nuestro sistema en donde cada componente va a exponer una funcionalidad al resto del sistema. de esta manera se modulariza el sistema a través de servicios independientes. Los clientes externos o inclusive los mismos servicios van a consumir las responsabilidades entre ellos.

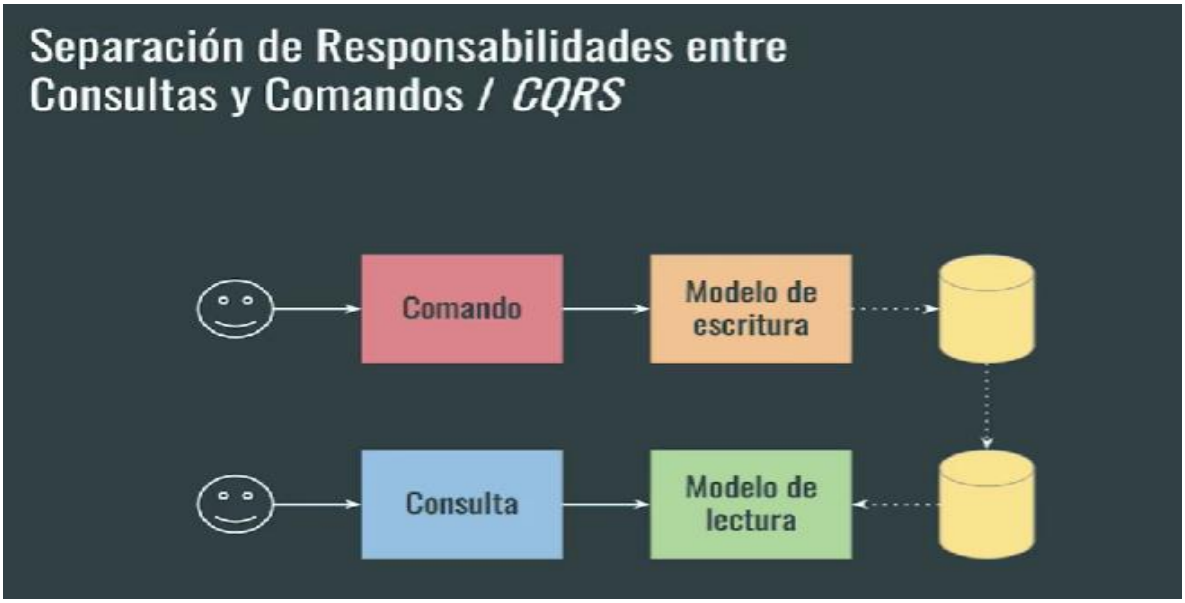
Cada componente debe tener su base de datos independiente.
y este es uno de los desafíos más grande de esta arquitectura; ya que tiene que analizar como interconectar estos servicios.

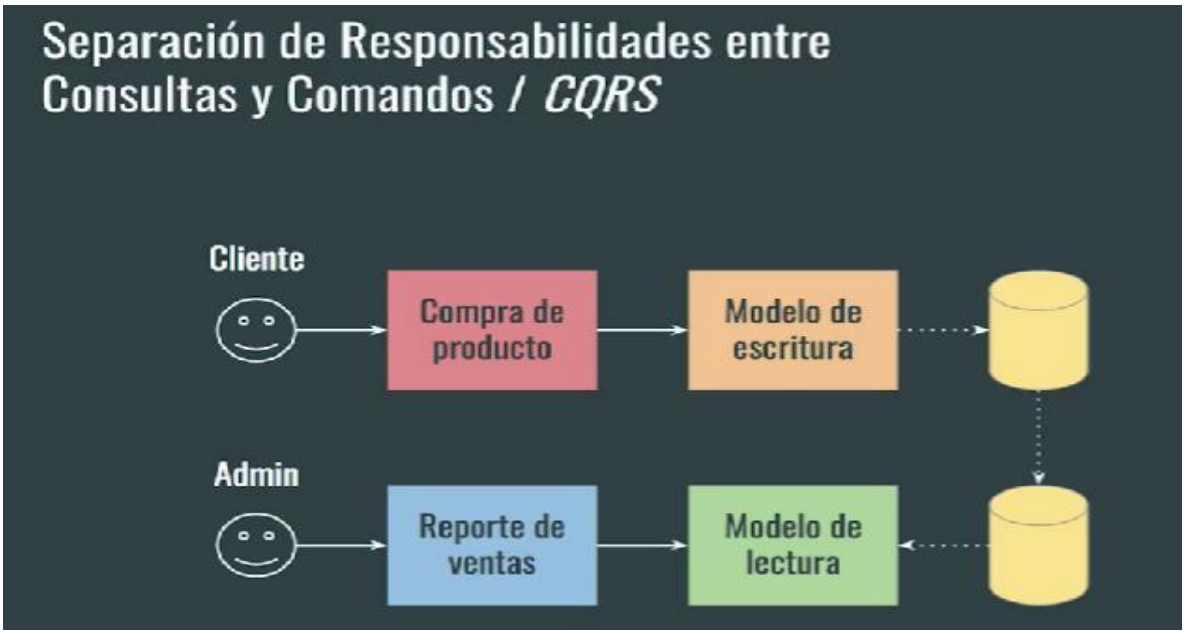
La conexión entre los servicios puede ser **directa:** Es decir, un servicio depende de otro. **Indirecta:** que los servicios se comunican a través de un Bus de Datos (Eventos).



PATRONES: CQRS

Nos dice que cuándo es muy difícil hacer óptima la lectura y escritura con un modelo compartido podemos aprovechar eso para separar ese modelo e incluso separar las bases de datos de esos modelos, de esta manera cuando queremos escribir tenemos un modelo optimizado para la escritura y luego cuando queremos leer tenemos un modelo optimizado para la lectura. Nos sirve para poder modelar el dominio de escritura y a su vez tener preparados los datos para poder leerlos de la mejor forma posible.



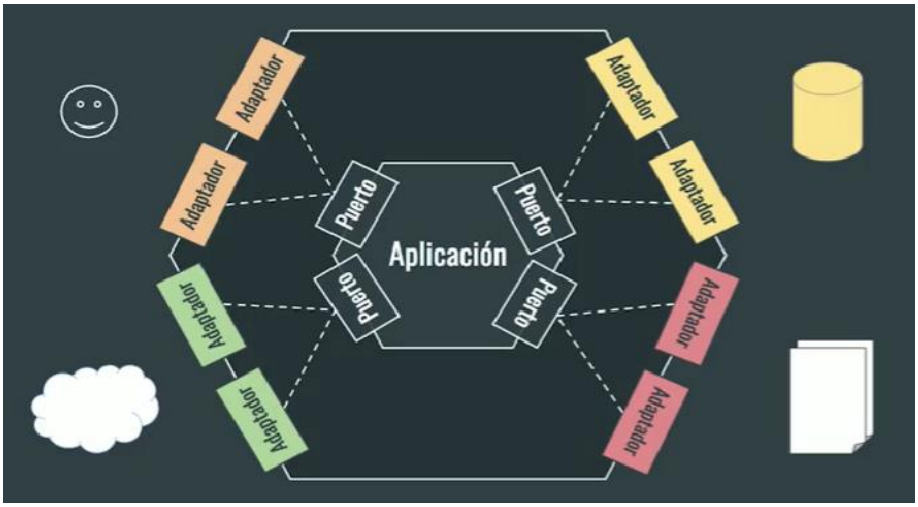


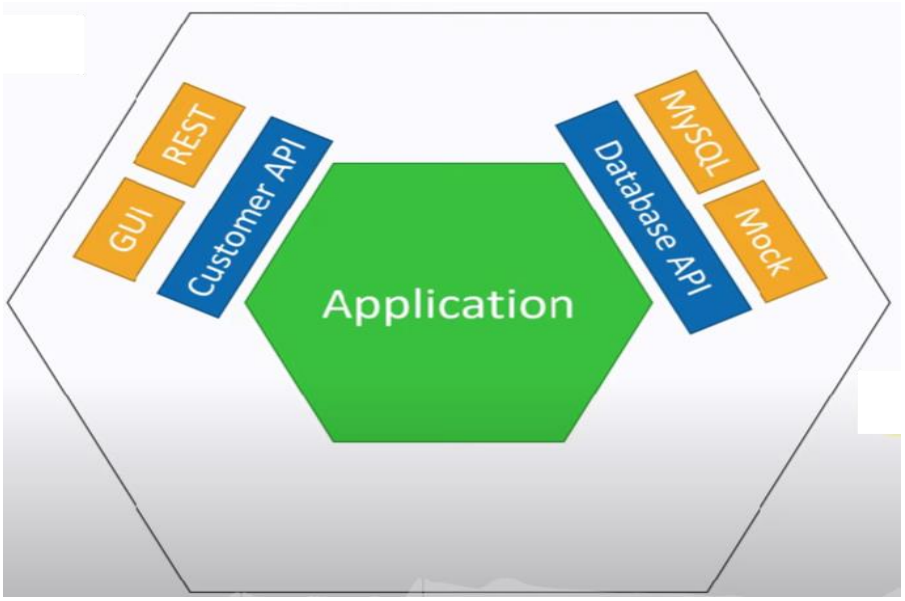
PATRONES: HEXAGONAL - PUERTOS Y ADAPTADORES

Nos ayuda a construir un patron que tiene bien claras sus dependencias externas. Esto lo hace definiendo puertos que son su capa fina con el exterior, donde la aplicación sabe su funcionalidad, pero no el cómo. Y existen adaptares que se encargarán del "Como" especificamente.

Si la App necesita acceso al file system o a la BD va a modelar un puerto que represente a sus datos y luego va a modelar un adaptador. Esta aplicación permite hacer tests de unidades muy fácilmente.

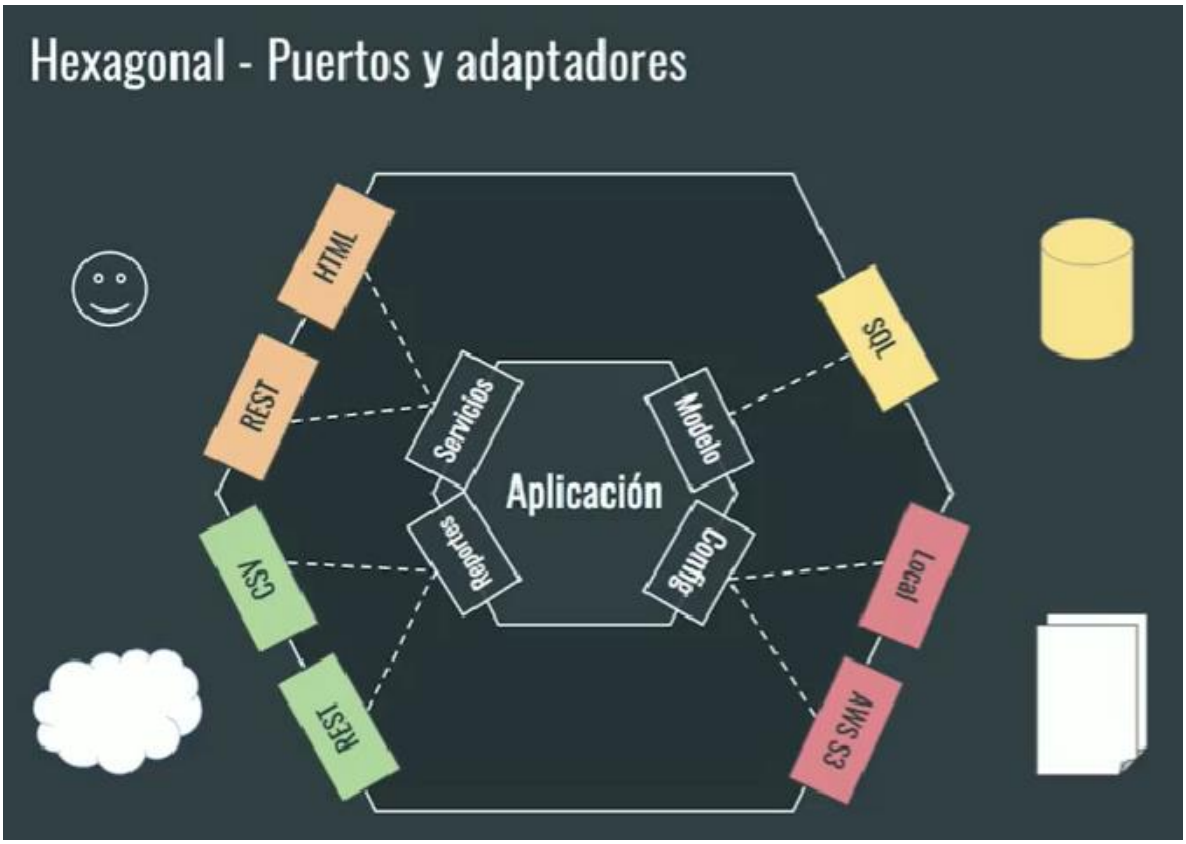
Ejemplos: Aplicaciones WEB.





https://www.youtube.com/watch?v=VLhdDYaW-uI

¿Qué es la arquitectura hexagonal? Aplicación, puerto y adaptador (platzi.com)



Ejemplo en código de implementación:

PATRONES: DISEÑO ORIENTADO AL DOMINIO (DDD)

- Domain Driven Design (DDD) es un nuevo enfoque de desarrollo de software.
- Representa distintas claves, **terminología**y **patrones utilizados** para desarrollar software donde** el dominio es lo más central e importante de una determinada organización**.
- Sus principios se basan en:
- Colocar los modelos y** reglas de negocio de la organización, en el core de la aplicación**
- **Basar nuestro dominio complejo, en un modelo de software**.
- Se utiliza para tener una mejor perspectiva a nivel de colaboración entre expertos del dominio y los desarrolladores, para concebir un software con los objetivos bien claros.

Pros

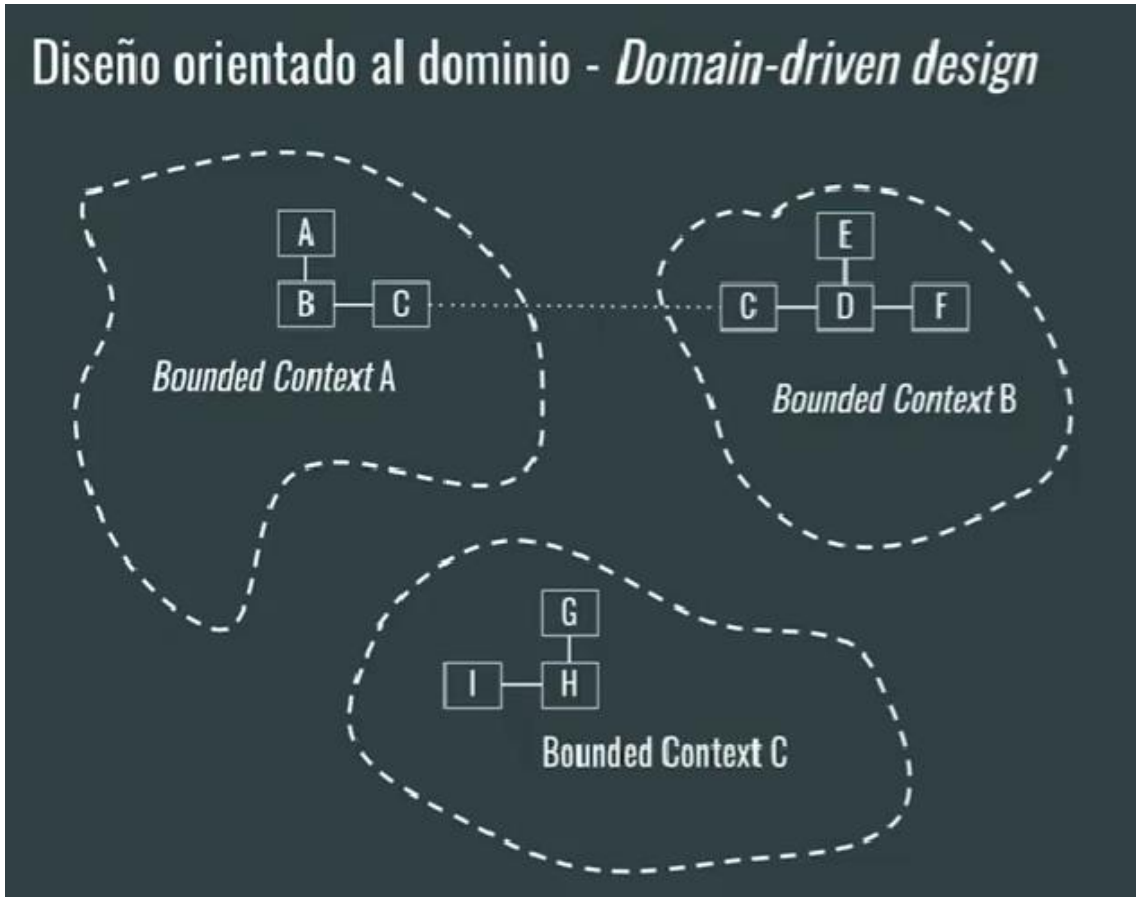
- **Comunicación efectiva entre expertos del dominio y expertos técnicos a través de Ubiquitous Languge**.
- **Foco en el desarrollo de un área dividida del dominio** (subdominio) a través de Bounded Context's.
- El software es más cercano al dominio, y por lo tanto es más cercano al cliente.
- **Código bien organizado**, permitiendo el testing de las distintas partes del dominio de manera aisladas.
- **Lógica de negocio reside en un solo lugar**, y dividida por contextos.
  Mantenibilidad a largo plazo.

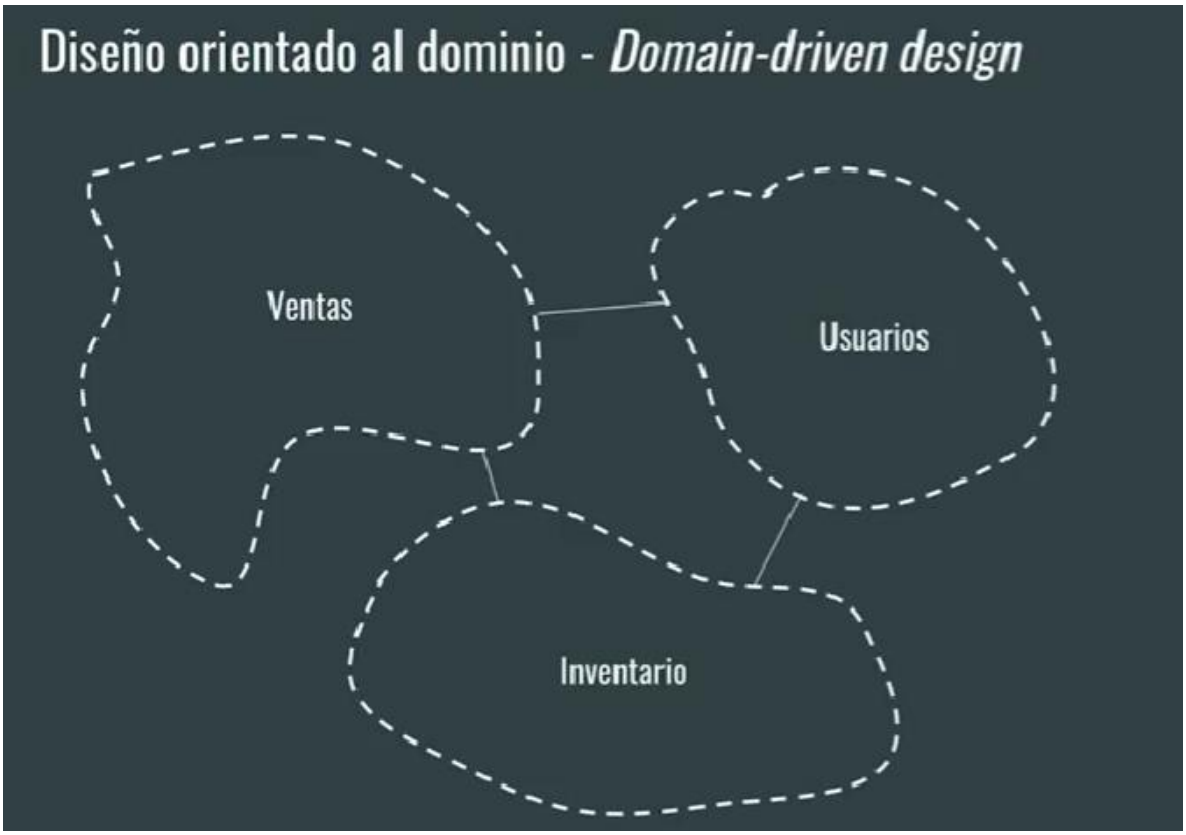
Contras:

- Aislar la lógica de negocio con un experto de dominio y el equipo de desarrollo suele llevar mucho esfuerzo a nivel tiempo.
- **Necesitamos un experto de dominio**
- **Una curva de aprendizaje alta, con patrones, procedimientos**…
- Este enfoque solo es sugerido para aplicaciones donde el dominio sea complejo, **no es recomendado para simples CRUD's**.

Fuente:

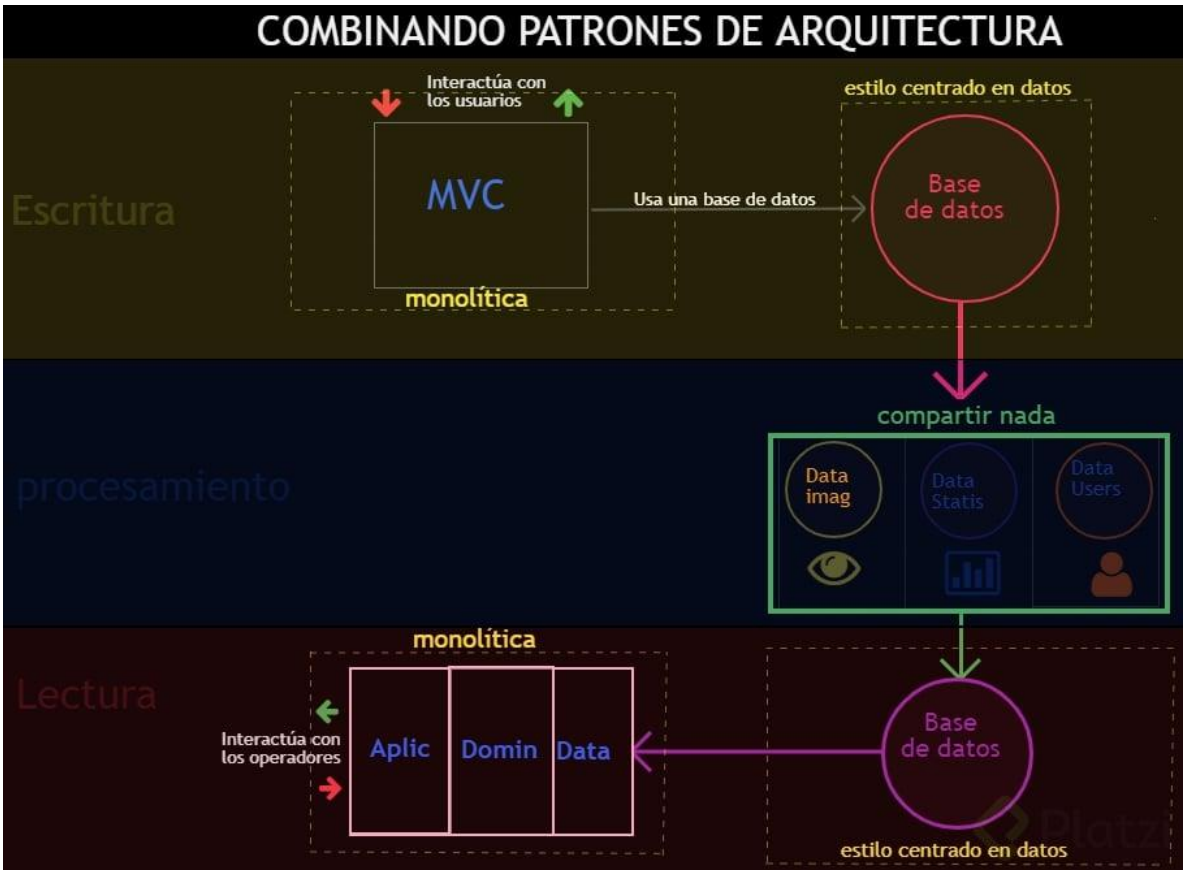
https://medium.com/@jonathanloscalzo/domain-driven-design-principios-beneficios-y-elementos-primera-parte-aad90f30aa35



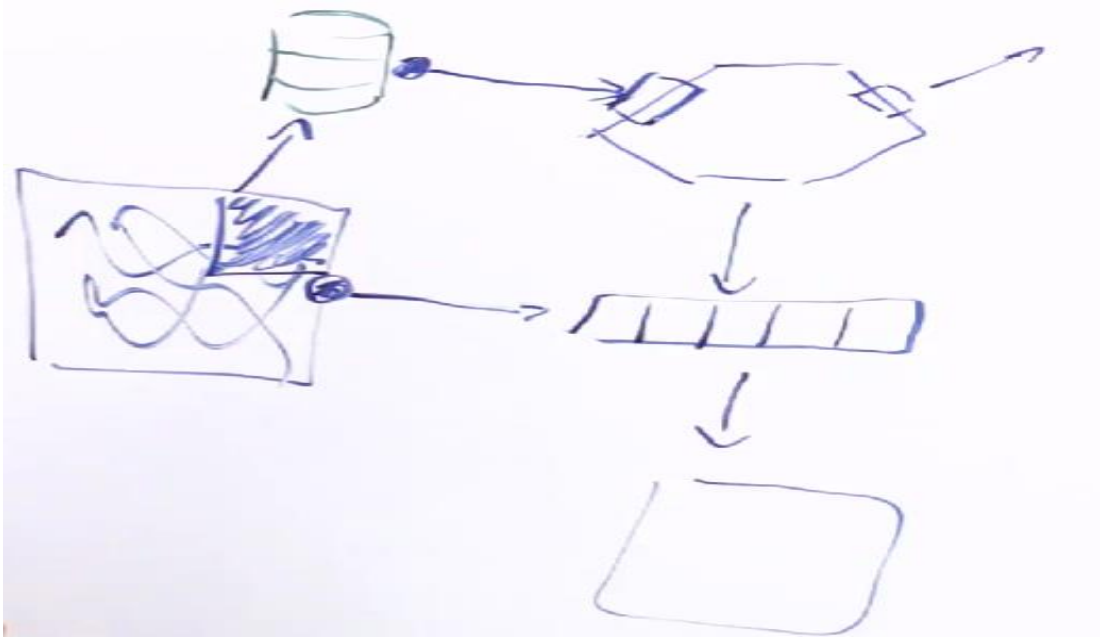

Diseño orientado al dominio - *Domain-driven design*
Ventas
Usuarios
Inventario

COMBINANDO PATRONES DE ARQUITECTURA

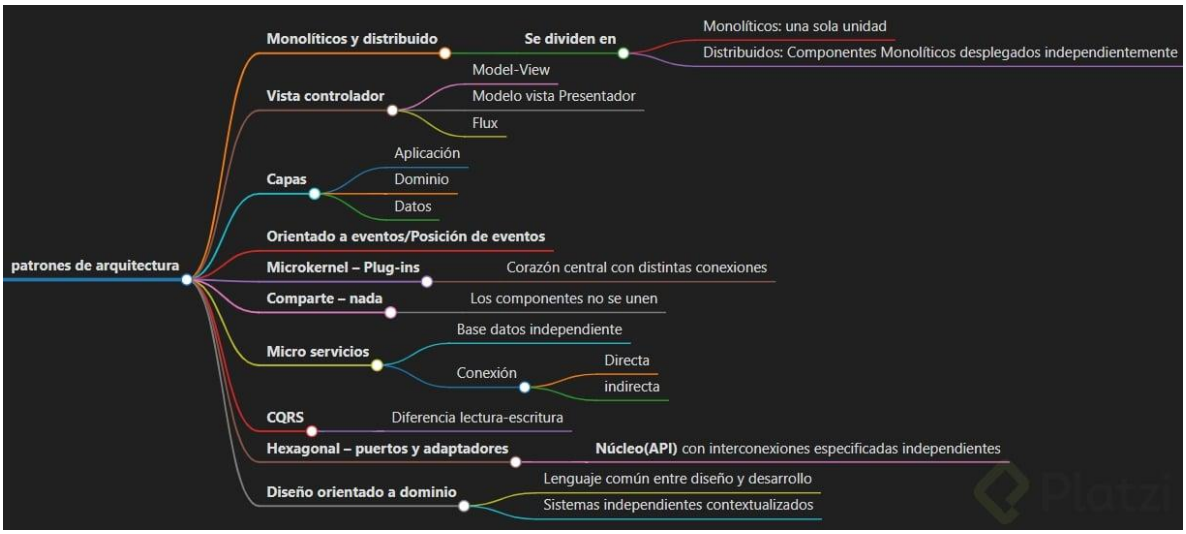
MVC + CQRS



Hexagonal:

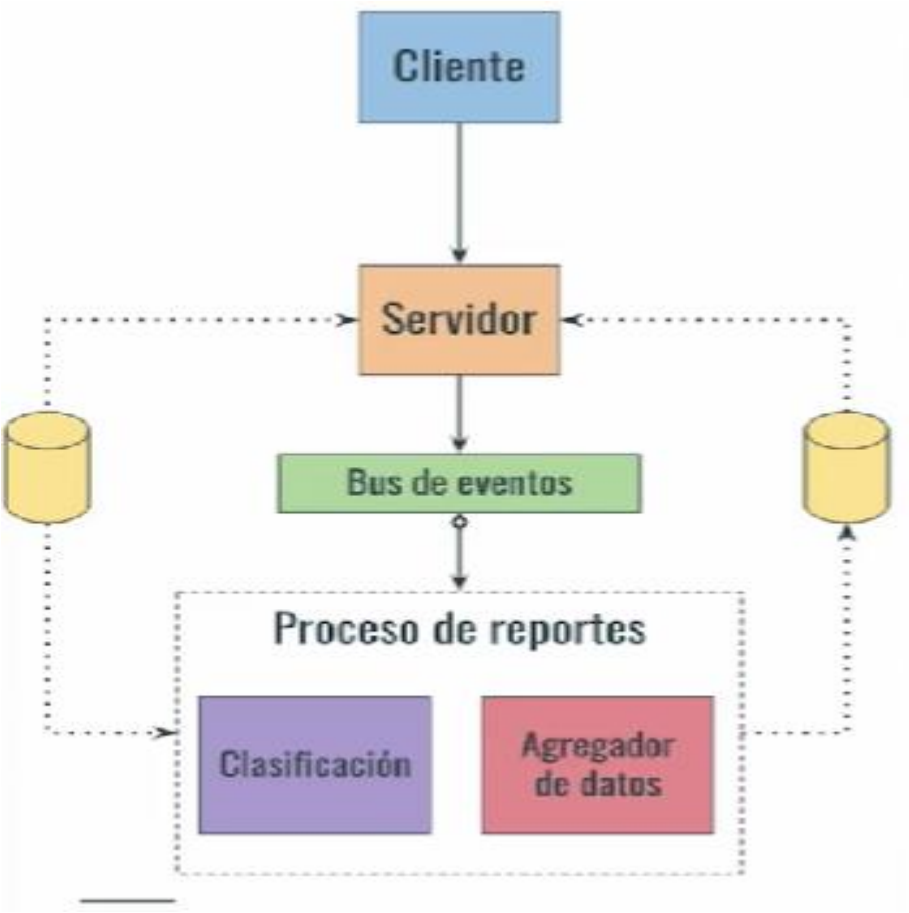


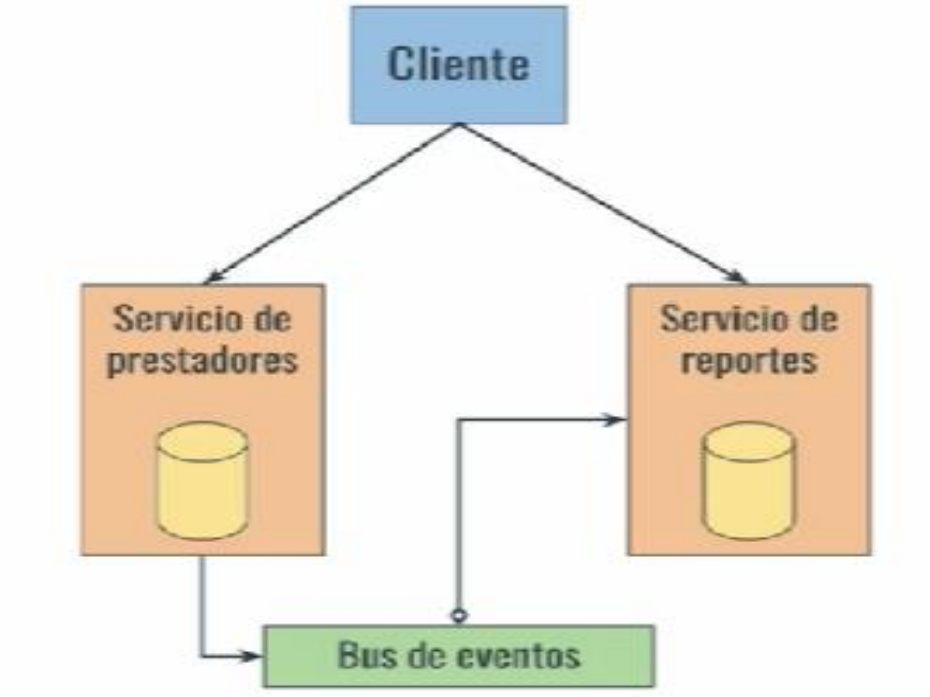
ANALIZANDO NUEVAMENTE PLATZISERVICIOS

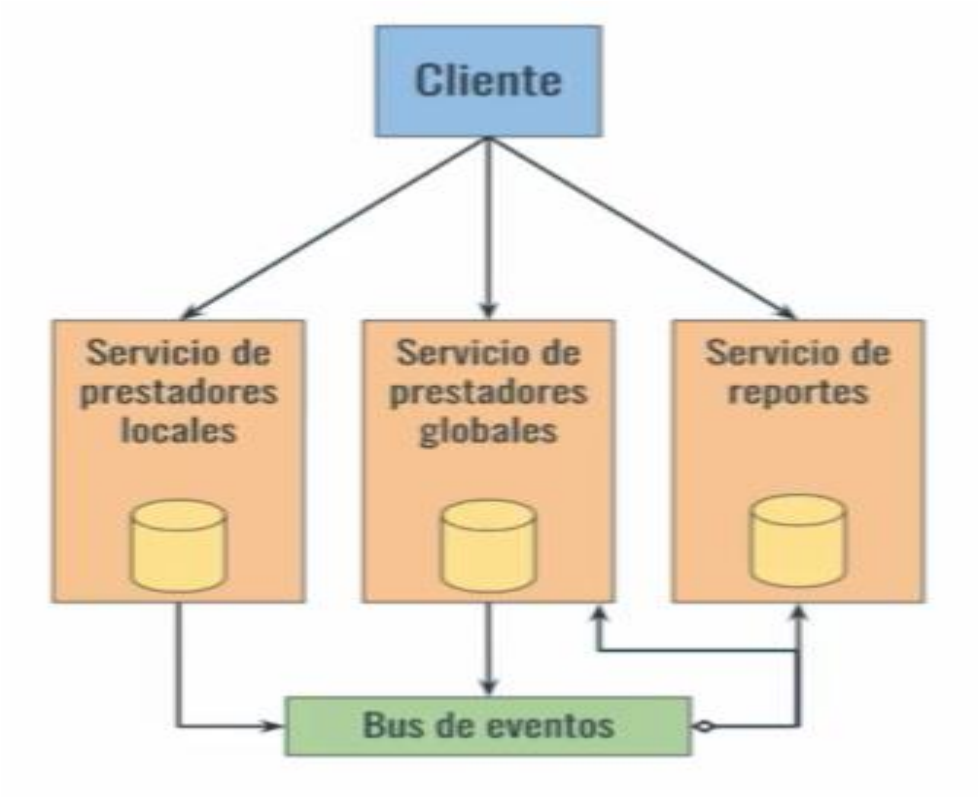


En los últimos años la Arquitectura de Software se ha consolidado como una disciplina que intenta contrarrestar los efectos negativos que pueden surgir durante el desarrollo de un software, ocupando un rol significativo en la estrategia de negocio de una organización que basa sus operaciones en el software.

Actualmente no existe una definición única para el concepto de arquitectura de software, el término ha sido abordado por un gran número de autores,1 no obstante, se reconoce como la definición más completa la dada por la IEEE Std 1471-2000: “La arquitectura de software es la organización fundamental de un sistema enmarcada en sus componentes, las relaciones entre ellos, y el ambiente, y los principios que orientan su diseño y evolución”


Cliente
Servidor
Bus de eventos
Proceso de reportes
Clasificación
Agregador de datos


Cliente
Servicio de prestadores
Servicio de reportes
Bus de eventos


Cliente
Servicio de prestadores locales
Servicio de prestadores globales
Servicio de reportes
Bus de eventos

DISEÑO DE UNA ARQUITECTURA

MODELADO Y DOCUMENTACIÓN DE ARQUITECTURA