**Introducción a la arquitectura de software**

**Etapas del proceso de desarrollo de software – Ciclo de vida del software**





**Dificultades en el desarrollo de software**

**Existen dos principales problemas en el desarrollo de software:**

**1) Esenciales**

Tienen que ver con el problema que vamos a solucionar, diseño y comprobación del concepto. Se dividen en 4 partes:

**• Complejidad:** La complejidad en si misma del problema.  
**• Conformidad**: El software en que contexto se va a usar y como se va adecuar a ese contexto que quizás no va ser perfecto.   
**• Tolerancia al cambio:** Va poder cambiarse o va ser algo que va ser muy difícil de cambiar (facturación y cosas de leyes, por ejemplo).   
      • **Invisibilidad:** El software es difícil al ser tangible y ser abstracto se dificulta ver y entender su forma.

**2) Accidentales:**

Detalles de implementación y producción, lenguajes, integraciones, servers, Frameworks etc. Se dividen en tres partes:  
      • Lenguajes de programación

      • Multiprocesamiento  
      • Entornos de programación

Los problemas accidentales nos traen ganancias cuando los resolvemos, incluso algunos son solucionables con a la alguna librería, api, etc. Lo verdaderamente importante es resolver los problemas esenciales.

**Como solucionar dificultades esenciales:**

Lo complejo de un desarrollo es lo esencial y no lo accidental. **No hay ninguna bala de plata que solucione el problema esencial del desarrollo de software**. Para ello nos dan 4 formas de resolver las dificultades esenciales:

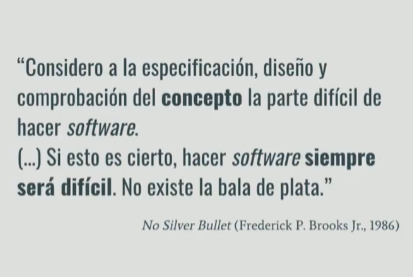
• **No desarrollar:** siempre que podamos primero ver si el problema se puede solucionar con un software ya existente, con algo de open source, servicios, integraciones y soluciones pequeñas que solucionen parte del probela etc.

• **Prototipado rápido**: son la evolución de las metodologías agiles, la idea es obtener e fedback lo más rápido posible de si estamos resolviendo el problema correcto. Para eso hay que ir evolucionando en pasos muy pequeños y siempre obteniendo feedback. EL FEEBACK ES LA HERRAMIENTA DE DESARROLLO MAS IMPORTANTE DENTRO DEL DESARROLLO DE SOFTWARE MODERNO.

• **Desarrollo evolutivo:** está relacionado al tipiado rápido, consta de ir desarrollando en pasos

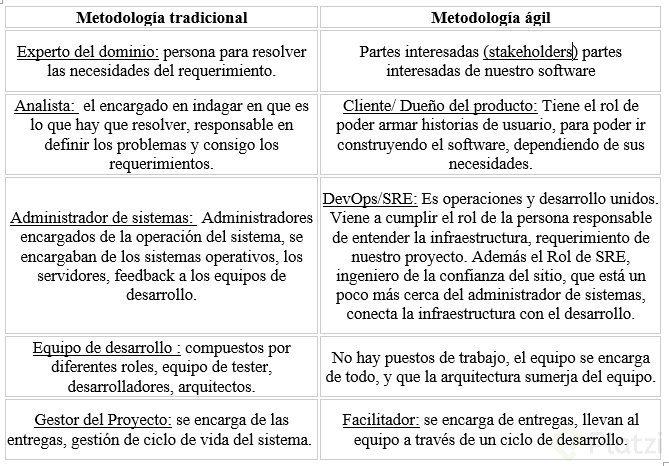
pequeños e ir evolucionando en ese sentido.

• **Grandes diseñadores:** personas que tengan la capacidad de diseñar una solución simple y que resuelva el problema de la mejor forma y con la mejor calidad.





**Roles en metodología tradicionales y ágiles**



**La ley de Conway:**

“Las organizaciones que diseñan software están limitadas a producir diseños que son copias de las estructuras de comunicación de estas organizaciones”

Ósea que, una organización compleja y burocrática solo puede producir sistemas complejos y burocráticos.



**¿Qué es arquitectura de software?**

No hay una definición formal entendida por toda la comunidad

**3 definiciones posibles:**

1. La arquitectura de software de un sistema es el conjunto de estructuras necesarias para razonar acerca del sistema.

2. Es el conjunto de decisiones de diseño importantes para organizar el software y promover los atributos de calidad deseados.

3. La arquitectura es acerca de las cosas importantes sean las que sean(diseño a mas alto nivel y depende de quien la vea)}

**Conceptos claves:** Estructuras, Razonar

Estructrutas: Como se va a organizar cada una de las partes del sistema y como se van a conectar cada una de estas partes, para eso se usan los patrones de arquitectura.

Razonar: La idea es que sirva para que el equipo de desarrollo pueda tener discusiones del diseño del sistema al mas alto nivel

Decisiones de diseño: Se tiene que definir para la arquitectura cuales son las decisiones de diseño que van a guiar el trabajo del equipo. Deben ser tomadas y documentadas para ser comunicadas apropiadamente al equipo.

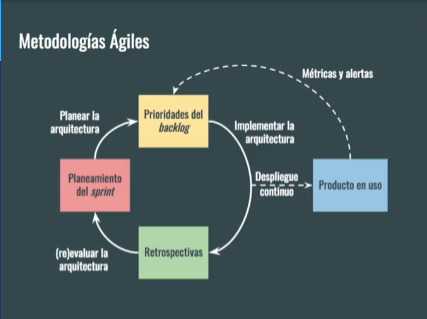
Objetivos del arquitecto de software: tienen diferentes partes interesadas y este arquitecto conecta estas diferentes partes o requerimientos con la implementación del sistema.

Stakeholders: Clientes, gestor, desarrollador, usuario y QA

Tener en cuenta las necesidades de cada uno de estos stake-holders le va ayudar al arquitecto a definir y tratar de establecer en su arquitectura ciertas decisiones, la unión de los requerimientos de todas estas personas van a llevar al arquitecto a tomar las decisiones que impacten y cumplan con el sistema a desarrollar.

**Arquitectura y metodologías:**





### Análisis de requerimientos

**Entender el problema**

La parte más importante de entender el problema es: **separar la comprensión del problema de la propuesta de solución**, si no se entiende la diferencia entre estos dos puntos se tiende a solucionar problemas inexistentes y a hacer sobreingeniería.  
.

**Problema:**

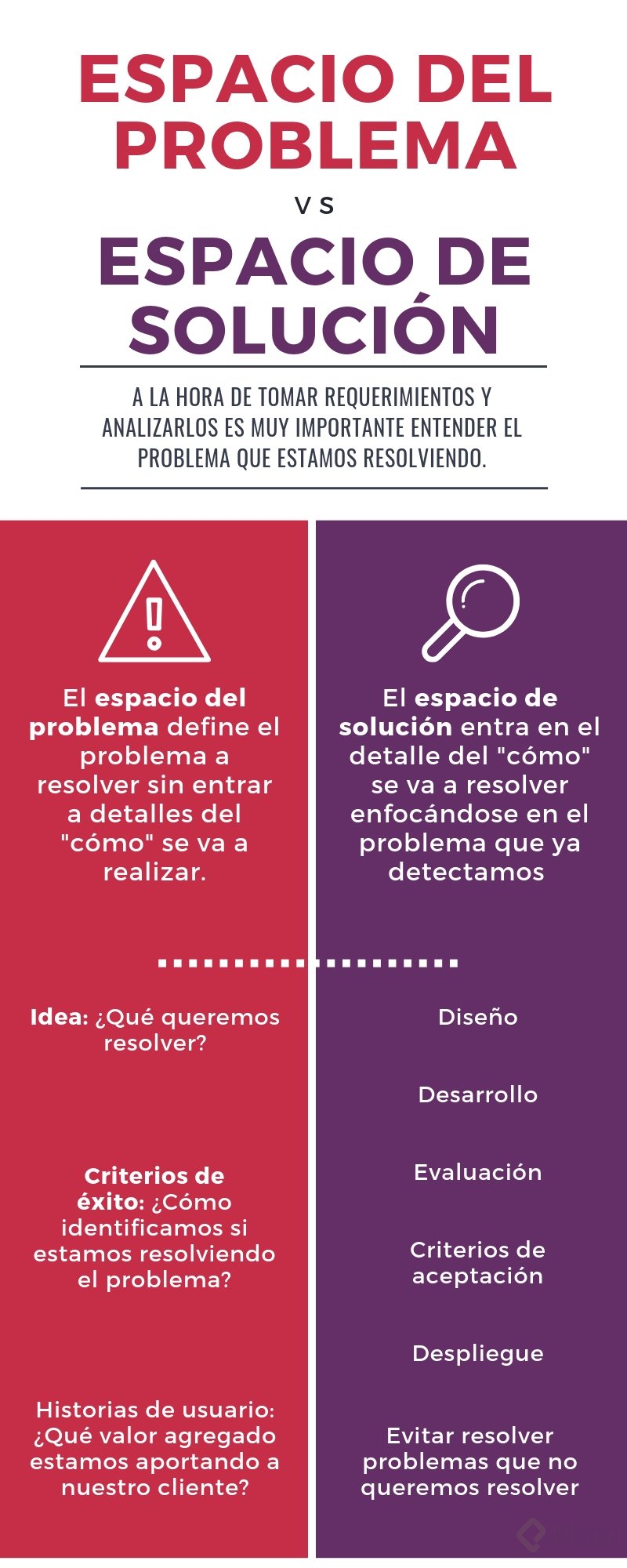
.  
Detalla **¿que es lo que se va a resolver?** (**y qué no** se va a resolver) sin entrar en detalles del “cómo”. -> (**analisis del problema**)  
.  
El espacio del problema nos ayuda a entender que es lo que vamos a resolver y exactamente como imaginamos como esto va agregar un valor a nuestros usuarios sin entrar en detalle de cómo lo va a resolver el sistema.  
.

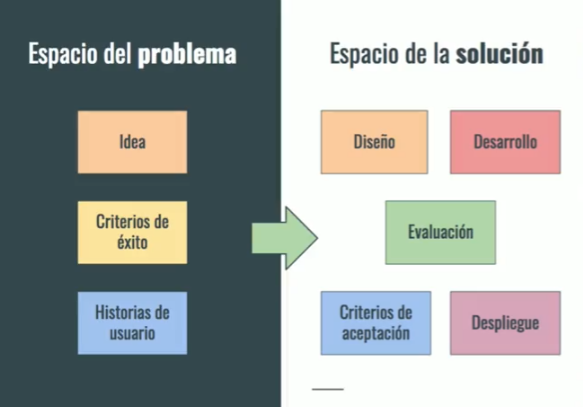
* **Idea**: ¿Qué queremos **resolver**?
* **Criterios de éxito**: ¿Cómo identificamos si **estamos resolviendo el problema**?
* **Historias de usuario**: Supuestos de historias de lo que **va a ganar** el usuario al utilizar la solución usando las **características del problema** a resolver.  
  .

**Solución:**

.  
Brinda el detalle del **¿“cómo” se va a resolver?**, reflejando los detalles del problema detectado y evitando resolver problemas que no se quiere o necesita resolver. --> **(detalles técnicos)**  
.  
Se refleja en el espacio del problema y trata de resolverlo teniendo en cuenta todos los detalles técnicos necesarios.  
.  
Consta de:

* **Diseño**: todo lo referente a la planificacion del software, desde diseño UI, UX hasta diseño de sistemas
* **Desarrollo**: escribir el codigo, configuraciones y contrataciones de servicios
* **Evaluación**: medir la eficiencia y eficacia del software frente al problema
* **Criterios de aceptación**: medir el **impacto** del software, no importa lo bueno que sea el problema si los usuarios no lo usan o no le ven uso
* **Despliegue (deploy)**: lanzar el software en ambientes productivos (mercado) y empezar a mejorar las caracteristicas con un feedback loop (crear, medir, aprender)





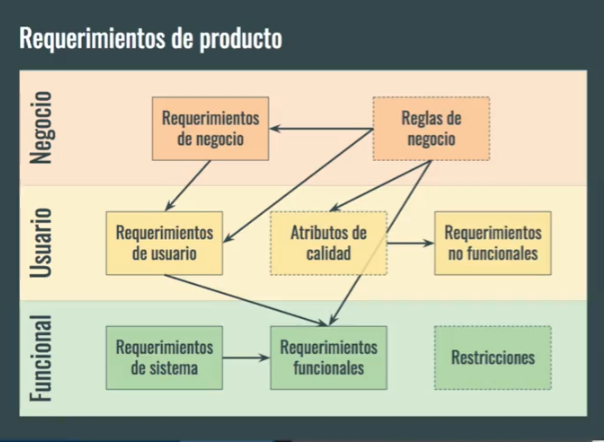
Requerimientos

Los requerimientos de producto se pueden separar en **funcionales** y **no funcionales**.

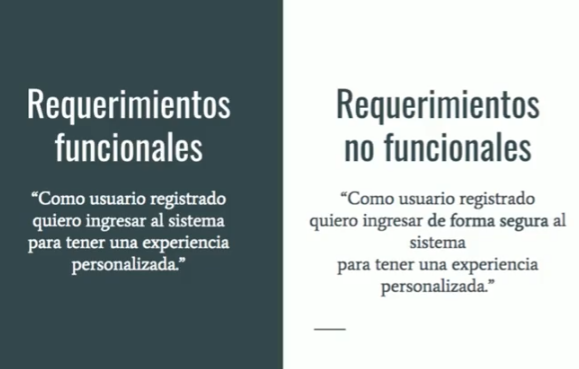
Los **funcionales** detallada específicamente como el sistema se va a comportar. Los requisitos funcionales establecen los comportamientos del software.

Los **No funcionales** representan características generales y restricciones de la aplicación o sistema que se esté desarrollando.  
Ejemplos de No funcionales:

* El sistema debe ser capaz de procesar N transacciones por segundo. Esto se medirá por medio de la herramienta Soap UI aplicada al Software Testing de servicios web.
* Toda funcionalidad del sistema y transacción de negocio debe responder al usuario en menos de 5 segundos.
* El sistema debe ser capaz de operar adecuadamente con hasta 100.000 usuarios con sesiones concurrentes.
* Los datos modificados en la base de datos deben ser actualizados para todos los usuarios que acceden en menos de 2 segundos.







Riesgos

**Describir el riesgo**  
Usar escenarios de fracaso que sean medibles y accionables.

“En situaciones de carga pico, los clientes experimentan latencias mayores a cinco segundos.”

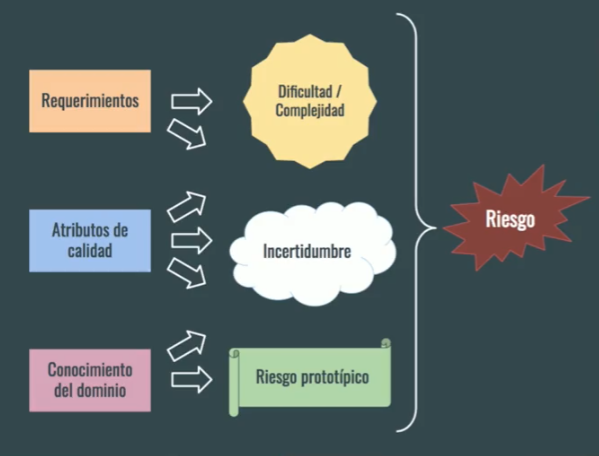
“Un atacante podría obtener información confidencial a través de un Ataque de intermediario (Man in the Middle).”

**¿Cómo identificamos riesgos?**  
**Requerimientos (Dificultad / Complejidad)**  
Es importante conocer si el requerimiento es complejo, es decir si la dificultad de resolver este requerimiento es muy alta.

**Atributos de calidad (Incertidumbre)**  
Es importante entender si sabemos o no sabemos cómo mejorar un atributo específico. Cuanta más incertidumbre hay en algo que detectamos que es importante, más alto es el riesgo de esa situación.

**Conocimiento del dominio (Riesgo prototípico)**  
Es importante saber si lo que hemos implementado ya ha sido implementado o no, porque los dominios conocidos suelen tener riesgos prototípicos.

**Priorizar riesgos**  
Es importante porque generalmente no podemos resolver todos, entonces si nos concentramos en resolver riesgos que no eran importantes, entonces estaremos invirtiendo mucho tiempo en algo que no era tan relevante. Debemos siempre tener en cuenta qué riesgos ponen en peligro el éxito o fracaso de la solución. Priorizamos nuestros riesgos y entendemos tanto nosotros como nuestros stakeholders que algunos riesgos no vamos a poder cubrirlos en el primer momento, sino que vamos a postergar el ataque o la mitigación de dichos riesgos para cuándo podamos invertir tiempo en ellos. Así los riesgos y los requerimientos van a ser priorizados y van a poder ser parte de nuestro plan organizado en dónde entendemos qué es lo más importante arquitectónicamente para resolver.



Restricciones

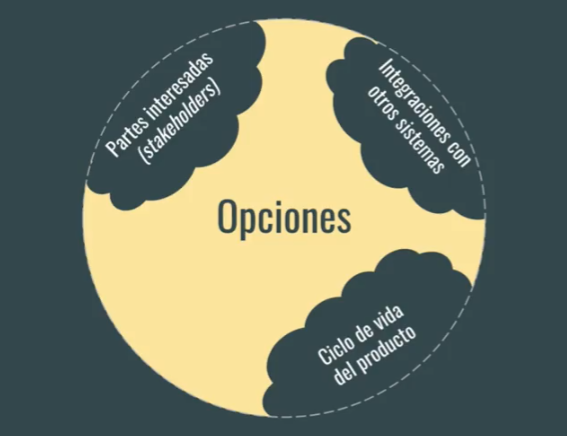
**Restricciones**

“Una restricción limita las opciones de diseño o de implementación disponibles al desarrollador.”  
Software Requirements: 3rd Edition (Wiegers, Betty 2003)

**Partes interesadas (Stakeholders)**  
Limitaciones quizás que tengan que ver con su ecosistema o contexto de negocio, limitaciones de tipo de regulaciones o cuestiones legales.

**Integraciones con otros sistemas**  
Si vamos a comunicarnos con un sistema por HTTP necesitamos tener internet, eso nos va a limitar en el ciclo de despliegue, como ejemplo.

**Ciclo de vida del producto**  
Un ejemplo común es que a medida que evoluciona la aplicación el modelo de datos es más difícil de cambiar



ESTILOS DE ARQUITECTURA DE SOFTWARE

**¿Qué es lo que está pasando arquitectónicamente en el software?**  
Hay muchas librerías, muchos frameworks y conocimiento arquitectónico implícito en las comunidades. Por ejemplo, si hablamos de palabras como MVC o FLUX (con React) estamos hablando de arquitectura, sin embargo, esta implícito dentro del uso de una tecnología específica, y de repente si hablamos de FLUX estando en Java o C# no tiene ningún sentido, ya que no es una arquitectura que se suele encontrar en esa tecnología, sin embargo arquitectónicamente tiene el mismo valor y podría ser implementado en otra tecnología. Asi también, hay decisiones arquitectónicas tales como si empezar un proyecto con un monolito o iniciarlo con una estructura de microservicio, que se dan por sentado que cualquier cosa seria de mucho mas valor iniciarlo como un microservicio ¿Por qué? Puede ser porque es la tendencia, porque es lo que se da más fácil para el equipo de desarrollo o porque las herramientas mas modernas de hoy están orientados a microservicios, sin embargo, falta un análisis más profundo de que es lo que define ese estilo o patrón de arquitectura y cuáles son los payloads o sacrificios que estamos pagando por usarlos y cuáles son los beneficios que esperamos que traigan.  
Ningún patrón tiene solo beneficios, cuando hablábamos de que no hay balas de plata, recordemos que ninguno de estos patrones ni estilos nos va a solucionar todos los softwares, siempre hay beneficios y consecuencias de las decisiones de diseño que tomamos.

**¿Qué es un estilo de arquitectura de software?**  
Al hablar de un estilo de arquitectura hablamos de algo genérico. Por ejemplo, podríamos entrar en detalles sobre diferentes páginas de internet: facebook, twitter, wordpress, Wikipedia, etc. todas esas paginas de internet implementan diferentes arquitecturas. Sin embargo, todas esas paginas son una pagina web, por lo tanto tienen una arquitectura cliente-servidor donde hay un navegador web que a través de un sistema de DNS y TCP/IP logra conseguir un documento en formato HTML que se lo muestra a través de un navegador al cliente. Esa estructura genérica define el estilo de una arquitectura, en donde, el estilo no nos va a hablar en detalles que problema esta resolviendo del dominio del problema, sino de que problema esta resolviendo arquitectónicamente a nivel de los conectores entre diferentes aplicaciones. Como dijimos recién podrían ser por ejemplo un navegador web y servidor, o podría ser una red de peer-to-peer, dos sistemas que están intentando intercomunicarse, o también una aplicación móvil que trata de comunicarse a una IP a través también de TCP/IP y HTTPS. Todo esto define algo genérico que si nos permite reutilizarlo a través de diferentes softwares, nos va a ayudar a poder reutilizar este conocimiento y aprender de soluciones anteriores que tuvieron éxito implementando esas comunicaciones o esos componentes con esos conectores. Si tuviéramos que bajarlo a una definición podemos decir que un estilo de arquitectura es una colección de decisiones arquitectónicas o decisiones de diseño que dado un contexto nos permite ya restringir las decisiones arquitectónicas, es decir, nos da un set de decisiones ya tomadas y nos restringe el resto de las decisiones arquitectónicas para un beneficio ya estimado, podemos usar estas decisiones ya tomadas en el pasado y que tuvieron éxito y aplicarlas en nuestro sistema que comparte un sistema general similar y esperar tener un éxito parecido al que tuvo quien lo implemento anteriormente.

Llamado y retorno

En este módulo se explica el estilo de arquitectura de llamada y retorno, consiste en que una parte A quiere comunicarse con una parte B, a veces con la necesidad de que b devuelva algo o no. Existen diferentes abstracciones de este suceso:

**1. Programa y subrutinas.**

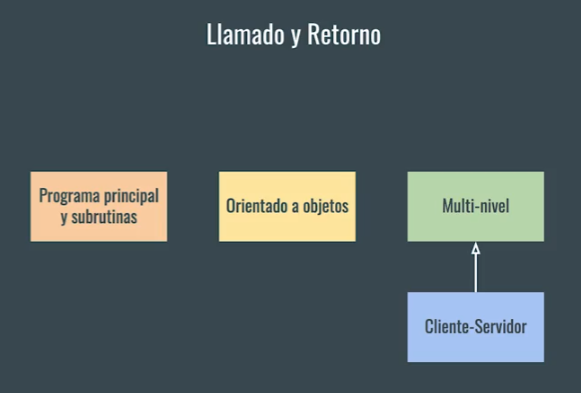
Programa es el conjunto de instrucciones, sub rutina tiene la cualidad de ser fragmentos de código que utilizaremos muy a menudo, al que necesitamos hacer un llamado.

**2. Orientado a objetos.**

Objetos tienen diferentes propiedades y métodos que se comunican entre sí. Objetos pertenecen a una clase que instancia objetos que se llaman y otros responden.

**3. Cliente Servidor**

Una aplicación le habla a un servidor mediante una petición y servidor responde. Si ahondamos en este modelo el nivel de detalle aumenta. Aplicación le pregunta a REST/API y luego a server, y server devuelve.



Atributos de calidad:

Permiten definir que es lo que va mas alla de lo funcional que va garantizar que el sistema funciones bien.

-Facilidad de mantenimiento

-Desplegabilidad

-Elasticidad

-Escalabilidad

2) Algunas actividades que hace un arquitecto de software

1. Define y balancea los atributos de calidad, entre mas atributos de calidad tenga, se debe tener unos y sacrificar otros si es necesario.

2. Define el problema desde un punto de vista de ingenieria. Toma todo el contexto plantea el problema y lo diseña a un alto nivel para resolverlo

3. Establece los principios de diseño que guían las decisiones de tecnología de una empresa. Debe ser capaz de definir linieamnientos que el equipo pueda seguir y poderlos guiar para que tomen decisiones

4. Garantizar el cumplimiento de la arquitectura, aunque define la arquitectura, debe involucrarse para que a medida que se implemente el diseño el este ahí garantizando que se cumpla.

5. Mantenerse al día con las ultimas tendencias tecnologicas. Saber que desarrollos se estan dando en la actualidad y sobre todo en la industria del negocio en la que esta ese sistema.

6. Conocer así sea de forma superficial distintas tecnologias como en front y back y movil, las entiende y puede tomar decisioens apartir de ese conocimiento.

7. Analiza el entorno y propone mejores para que la arquitectura de un sistema se mantenga relevante. Propone mejoras a la arquitectura.

8. Adiministrar la deuda tecnica: Es la diferencia que hay entre el estado actual del sistema y el estado ideal, es decir el que debe tener. Debe saber como disminuir esa diferencia

9. Entiende y sabe moverse en el entorno político de la empresa.

Sus decisiones pueden afectar no solo al area de Ti sino muchos departamentos de la empresa, entiende la empresa y el contexto en el que esta para dar el impacto correcto en la organización

10. Le da mentoría a su equipo.

Es capaz de trasmitir sus conocimientos para que sus personas a cargo mejoren sus capacidades de diseño.

11. El arquitecto de software desarrolla

Así no sea mucho se involucra con el código para entender las consecuencias de lo que esta creando.

Cuales el el patrón correcto para un sistema

Para saberlo se debe entender lo que son:

Los atributos de calidad o requisitos no funcionales

Se hace una comparación de cada patron contra cada atributo de calidad

Escalabilidad= Es la capacidad del sistema para que con el tiempo pueda soportar cargas de trabajo mas grandes. Como por ejemplo mas usuarios

Microservicios

Arquitectura orientada a eventos

Desplegabilidad= Es la facilidad de desplegar un sistema en producción

Microkernel

Microservicios - Masomenos

Eventos - Masomenos

Capas - La peor

Rendimiento

Es cuando la aplicación tiene tiempos de respuesta apropiados a las operaciones que está haciendo

Eventos

Microkernel - regular

Capas - regular

Microservicios- Malo

Agilidad

Es que tan rapido se puede llevar a producción cambios para que la empresa se mantenga comptetitiva.

Microkernel

Eventos

Microservicios

Capas - Malas

Testeabiidad = Es que tan facil se pueden hacer pruebas en el sistema.

Capas

Microkernel

Microservicios

Eventos-Mala

Facilidad de desarollo= Es qué tan fácil es empezar a implementar el patrón de arquitectura.

Capas

Microservicios-Depende

Microkernel - Mala

Eventos – Mala

Arquitectura por capas

Es un patrón que contiene grupos de subtareas

Analogía



Si tiene que pasar SI O SI atraves de una capa para llegar a la capa de abajo, las capas no se pueden saltar. La app se separa en grupos o capas que da la estructura general

Carateristicas

-Los componentes estan organizados en capaz horizontales

-Cada capa tiene un rol y una responsabilidad unica dentro de la app

-Capas de alto y bajo nivel, alto nivel significa que esta mucho mas cerca del cliente, bajo nivel significa que es una capa mucho mas cerca de manejar cosas como hardware.

-El patrón en ningun momento define cuantas capas son ni de que tipo deben ser, se puede tener una, dos tres, seis o 10 capas en función del contexto y del sentido que tenga tenga n cantidad de capas.

-La capa de alto nivel deberia poder usar servicios de una capa de bajo nivel, pero al reves no se puede, la capa inferior no debe tener ningun conocimiento de la capa superior

-Las capas deberian tener una abstracción o encapsulamiento de la funcionalidad que tiene, algunos detalles probablemente o funciones quizas no sea conveniente que se puede acceder o interactuar con ellos. Dado que si esto pasa se pueden generar dependencias entre capas mas alla de lo necesario

-Las capaz deben estar bien definidas de tal manera que se puedan remplazar por otras capas si es necesario por algun motivo. Si se remplaza una capa a por b c deberia ser indiferente a esto.

-Los componente de cada capa solo deben hacer tareas relacionadas con la capa.

PATRÓN DE ARQUITECTURA DE TRES CAPAS

Se compone de

PRESENTACIÓN

Esta capa se encarga de manjar la interacción general entre un cliente y una aplicación, es ese punto por donde todo inicia.

El cliente puede ser:

Un ser humano

Un programa externo

Responsabilidades

-Entrega y solicita información al usuario, todo eso con el fin de alimentar las capas posteriores

-Interpreta solicitudes del usuario en acciones para las capas de negocio y persistencia

Ejemplos de componentes en la capa de presentación

-Una aplicación web

-Una interfaz por consola

-Una aplicación de escrtiorio

-Una aplicación móvil

-Un servicio web

-Un API

CAPA DE DOMINIO

-Es la que contiene la lógica de negocio del sistema, es una funcionalidad que es central para que el negocio opera y esta existe haya o no sistema de información.

-Contiene el trabajao o las tareas para las cuales el sistema esta hecho.

-Normalmente, la capa de dominio deberia esconder el acceso a datos.

Ejemplos

-Calculos basados en datos de entrada y datos almacenados, como calcular un impuesto, un salario, una venta, es decir toma los datos de la capa d epresentación o de acceso a datos y hace calculos.

-Validaciones de datos que vienen de la capa de presentación

-Definir que algoritmo o logica de negocio utilizar

Otros nombres para la capa de dominio

-Capa de negocio

-Capa de lógica de negocio

-Capa de lógica de dominio

CAPA DEACCESO A DATOS

-Se comunica con bases de datos u otros sistema de perseistencia para obetener y guarar información, es decir, es la que tiene el codigo para obtener los datos de las fuentes d einformación como la base de datos y las diversas opciones que existen

-De forma mas general, se comunica con sustema que realizan tareas en n nombre de la aplicación

-Monitores de transacciones

-Sistemas de mensajeria

-Entre otros

Otros nombres para la capa de acceso a datos.

-Capa de persistencia

-Capa de datos

PATRÓN DE ARQUITECTURA MICROSERVICIOS

Es una patrón de arquitctura considerado mas de moda o mas moderno.

Generalidad basicas

Es muy importante aprender que son las aplicaciones monoliticas, lo mas común al crear una aplicación es tener una aplicación que tenga un cliente y un servidor. Eso esta junto en una sola app ejecutable, esto se ve como si fuera como si fuera un gran bloque junto, por eso se le llama un monolito, escalarlas es dificl y caro otro reto que hay es que para desplegar un cambio hay que desplegar la aplicación entera

De aquí es que surge la idea o lo que viene a romper los microservicios, Esta habla de desarrollar una aplicación como un conjunto de servicios pequeños, en lugar de un gran monolito donde esta toda la app, ese monolito se separa en servicios mucho mas pequeños con las siguientes caracteristicas

-Cada microservicio corre su propio proceso, es decir son procesos totalmente independitenes y no hay como un solapamiento del uno con el otro.

-Si los microservicios necesitan comunicarse, lo hacen con mecanismos muy livianos

-Se desarrollan alrededor con las necesidades del negocios, hay una relación muy fuerte entre la forma en como opera el negocio y la forma en como se desarrollan los microservicios.

-Tienen que estar soportados por un proceso de despligue automatico.

Tpologías -Distintas formas en las que los servicios pueden estar oganizados

1. Topología basada en APIS Rest, Se tiene un conjunto de servicios y estos al ser consumidos no son accedidos directamente si no que se usa una capa de API y esta capa de API y es la que expone las URLS publicas que los usuarios van a acceder, entonces la idea es que si un usuario invoca una de estas URLS, la capa de API lo que hace redireccionar la URL hacia el microservicio o hacia el componente de servicio apropiado y ese servicio la genera y procesa y envia de regreso para que la API le responda al cliente.
2. Topología basa en aplicación, la aplicación que accede a los microservicios normalmente es una aplicación hecha por le desarrollador y esta APP lo que hace es conectarse directamente a los microservicios, los cliente interactuan con esa app y esta es la que encapasula o esconde los llamados a los microservicios. Se usa generalmente en apps pequeñas o medianas.
3. Topologías basadas en mensajeria, en este caso no se interactuan con los microservicios atraves de REST si no que se hace atraves d eun broker de mensajeria, normlamente en estos casos,hay una aplicación que es la qu e se encarga de acceder la cola y cada vez que los usuarios hacen solicitud a la app, esta pone un mensaje en la cola, y la cola lo que hace es redigiri ese mensaje al mensaje servicio correspondiente y luego devolver la respuesta, este es un enfoque mas usado en aplicaicón grandes donde se suele necesitar mas ocntrol en la capa de transporte.

ORIENTADA A EVENTOS

Componentes

-Generadores o productores.

Se encargan de generar los distintos eventos del sistema.

Mensajes, son los que contienen la información del evento, es como la pieza que contiene la información relacionada con el evento.

Componente de mensajeria, es la platafroma que recibe el mensaje hace un procesamiento con el mensaje si es necesario e informar a las parte interesadas es decir los distintos consumidores.

Canales, son como los canales entre el componente de mensajeria y los componentes interesados en ese notificados acerca del evento. Ya sea con colas de mensajes – Patron publicador - subscritor

Procesadores o consumidores, son los componentes interesados en los eventos, tienen una unica tarea relaiconada con el negocio, que sea lo mas espesficia posible, principios SOLID.

Dben estar lo mas desacoplados posibles de otras partes de la APP, no deberian tener nigun conocimiento de quienes generan los eventos, ni de otros procesadores que puedan existir en el sistema, lo ideal es que sean lo mas independientes posible de las otras partes de la arquitetcura del sistema.

Topologías, son basicamente la forma de organziarel componente de mensajeria

El mediador y el broker

En el mediadior la idea basica es que dentro de el componente de mensajaria el mediador se encarga de sincronizae to-dos los procesadores, se encrga de que los procesadores se ejecuten en la secuencia correcta para producir el resultado deseado.

MVC –Modelo vista controlado

Vista

-Muestra el modelo

-Múltiples vistas para un modelo

-Actualiza con notificaciones del modelo

Controlador

-Lógica de aplicación

-Recibe solicitudes de la vista

-Ejecuta acciones en el modelo

Modelo

-Lógica de negocio

-Tiene datos, comportamiento y estado

-Independiente de controlador y vista

-Activo o pasivo

¿MVC es lo mismo que una arquitectura de tres capas?

Es similar, si se hiciese una relación podría ser así

Vista - Presentación

Controlador – Dominio = Aquí si hay una gran diferencia

Modelo – Acceso a datos = Hay similitud pero hay diferencis muy importantes