

## FUNDAMENTOS DE ARQUITECTURA DE SOFTWARE

### EL PROCESO DE DESARROLLO DE SOFTWARE

#### INTRODUCCIÓN AL CURSO DE FUNDAMENTOS DE ARQUITECTURA DE SOFTWARE

#### ETAPAS DEL PROCESO DE DESARROLLO DE SOFTWARE

El proceso de desarrollo tradicional tiene etapas muy marcadas, que tienen entradas, procesos y salidas que funcionan como entradas de la siguiente etapa.

**Análisis de requerimientos:** Todo nace de un disparador que nos crea la necesidad de crear un artefacto o un sistema. Necesitamos entender cuál es el problema que queremos resolver. Hay requerimientos de negocio, requerimientos funcionales, requerimientos no funcionales.

**Diseño de la solución:** Análisis profundo de los problemas para trabajar en conjunto y plantear posibles soluciones. El resultado de esto debe ser el detalle de la solución, a través de requerimientos, modelado, etc.

**Desarrollo y evolución:** Implementación de la solución, para garantizar que lo que se está construyendo es lo que se espera. Al finalizar esta etapa tendremos un artefacto de software.

**Despliegue:** Aquí vamos a necesitar de infraestructura y de roles de operación para poder poner el artefacto a disponibilidad.

**Mantenimiento y evolución:** Desarrollo + despliegue + mantenimiento, en esta etapa estamos atentos a posible mejoras que se hacen al sistema. En esta etapa el software se mantiene hasta que el software ya deja de ser necesario.

## DIFICULTADES EN EL DESARROLLO DE SOFTWARE

En la etapa de diseño y desarrollo estamos concentrados en encontrar cuáles son los problemas que queremos resolver. Estos problemas los podemos dividir en dos grandes tipos de problemas.

**Esenciales:** Los podemos dividir en 4.

1. La complejidad, cuándo lo que tenemos que resolver es complejo en sí mismo, por ejemplo calcular la mejor ruta entre ciudades.
2. La conformidad.
3. Tolerancia al cambio.
4. Invisibilidad.

**Accidentales:** Está relacionado con la plataforma que vamos a implementar, tecnología, lenguajes, frameworks, integraciones, etc.

### Notas adicionales:

Se dividen en 2 tipos de problemas:

- Esenciales: Especificación, diseño y comprobación del concepto
- Accidentales: Detalles de la implementación y producción actual

### Esenciales

1. **Complejidad:** Cuando el problema a resolver es complejo en sí mismo.
2. **Conformidad:** Entender en qué contexto se usará el producto y como se adapta al contexto imperfecto (Ejemplo: ¿Requiere internet, qué disponibilidad existe de Internet?, ¿Requiere actualizaciones mayores?)
3. **Tolerancia al cambio:** La medida de cuánto podemos adaptarnos a los cambios.
4. **Invisibilidad:** Dificultad de entender su forma, ya que es intangible, solo existen en código o documentación

### Accidentales

1. **Lenguajes de alto nivel:** Dificultad de dominar el stack tecnológico.
2. **Entornos de desarrollo:** Las dificultades expuestas por software de desarrollos, editores de textos, IDE's, herramientas de terceros.

## ROLES EN METODOLOGÍAS TRADICIONALES Y ÁGILES

Es importante que diferenciamos el ROL del puesto de trabajo, hay roles que pueden ser desarrollados por la misma persona.

**Experto del dominio:** En una metodología tradicional, es la persona a la que acudimos para entender las necesidades del negocio.  
En [metodologías ágiles](#) y stakeholders.

**Analista:** funcional/de negocio, la persona responsable de definir los requerimientos que van a llevar al software a su buen puerto. En el caso de Ágiles el dueño del producto es quien arma las historias y que nos acompaña en el proceso de construcción del software.

**Administrador de sistemas / DevOps:** Es el rol de operaciones y desarrollo, son las personas responsables de la infraestructura que alojara nuestra aplicación.

**Equipo de desarrollo:** QA / Testing se encargan de la evaluación de nuestro software, comprobar que lo que se está haciendo es lo que se espera que se haga. Desarrolladores involucrados en la construcción del software. Arquitecto, diseña la solución y análisis de los requerimientos, es un papel más estratégico. La arquitectura emerge del trabajo de un equipo bien gestionado.

**Gestor del proyecto / facilitador:** Llevan al equipo a través del proceso iterativo e incremental, entender lo que pasa con el equipo y motivar el avance en el desarrollo del producto.

## **Notas adicionales:**

Metodología tradicional

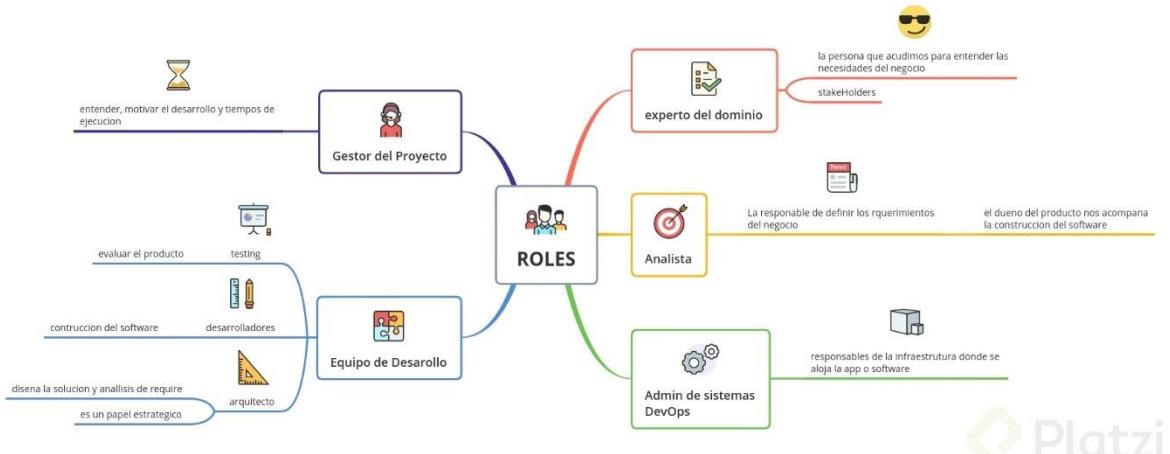
1. **Experto del dominio:** Experto en necesidades de los requerimientos
2. **Analista:** Persona responsable a definir el problema y buscar la solución
3. **Sysadmin:** Encargado de toda la operación del sistema
4. **Equipos de desarrollos:** Desarrolladores encargados de asegurar la calidad del producto, se dividen en muchas áreas, QA, Tester, Desarrolladores, Arquitectos.
5. **Gestor de proyectos:** Administrador general de todo el proceso de desarrollo

Metodología ágil.

1. **Partes interesadas/stakeholders:** Terceros expertos en el área del producto.
2. **Dueño del producto:** El cliente que conoce su producto y sabe sus problemas, el puede realizar las historias y determinar cual sería el mejor camino para priorizar por etapas soluciones.
3. **DevOps:** Conecta el desarrollo y las operaciones de infraestructura
4. **Equipos de desarrollos:** Desarrolladores encargados de asegurar la calidad del producto, los equipos son autogestionados para realizar ellos mismo dichas funciones.
5. **Facilitador:** Administrador general del producto, generalmente se encuentra muy atento a los nuevos cambios del día al día ya que irá cambiando todo el tiempo.



Platzi



Platzi

# INTRODUCCIÓN A LA ARQUITECTURA DE SOFTWARE

## ¿QUÉ ES ARQUITECTURA DE SOFTWARE?

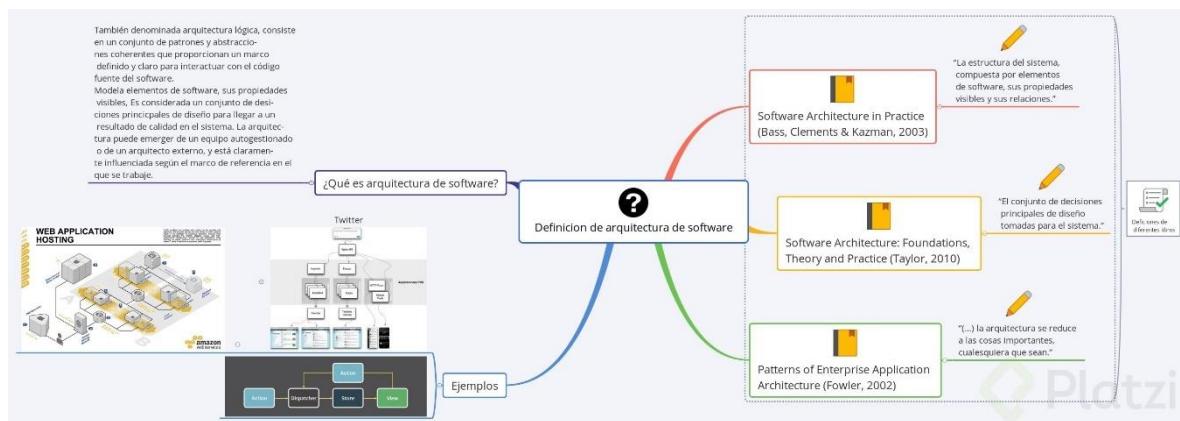
La arquitectura de software modela elementos de software, sus propiedades visibles, etc. Es considerada un conjunto de decisiones principales de diseño para llegar a un resultado de calidad en el sistema. La arquitectura puede emerger de un equipo autogestionado o de un arquitecto externo, y está claramente influenciada según el marco de referencia en el que se trabaje.

Ejemplo de la arquitectura de twitter:

Inicialmente procesa el mensaje que se twitteará a través de una API que se comunica con unos servicios separados que a su vez tienen diferentes funciones y que están en Asynchronous path.

Amazon web services, orienta su arquitectura al despliegue, disponibilidad , carga y crecimiento de una aplicación. Mientras Flux, de React, habla en un alto nivel de su arquitectura según un flujo monodireccional de los datos.

Notas adicionales:

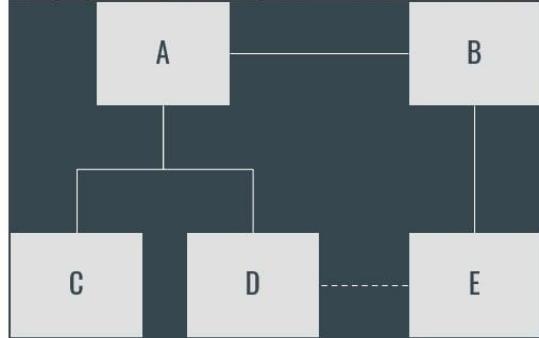


### ¿Qué es arquitectura de software?

En este módulo vamos a entender que es la **arquitectura de software** a través de diferentes definiciones y ejemplos, también vamos a ver que es un **arquitecto de software**, que se espera de un arquitecto, como se desenvuelve en diferentes organizaciones y como estas organizaciones afectan al arquitecto y a la arquitectura resultante del sistema, y por último veremos que es arquitectar, cuáles son los objetivos del arquitecto y como esos objetivos varían dependiendo de la metodología de trabajo.

#### **¿Qué es arquitectura de software?**

Siempre que hablamos de arquitectura de software solemos hablar de algo así:



Un modelo o diagrama en donde tenemos conexiones entre cajas y líneas. La arquitectura de software está muy sesgada por los modelos y por el diseño de una solución a través de un modelado, pero esto no es esencialmente la arquitectura. La arquitectura llega a modelos y plantea modelos haciendo análisis fuertes sobre qué es lo que hay que construir, que son los requerimientos que hay que tomar y como es que ese sistema los va a resolver.

Por ejemplo, tomemos la definición del libro «**Software Architecture in Practice**»

**“La estructura del sistema, compuesta por elementos de software, sus propiedades visibles y sus relaciones.”**

*Software Architecture in Practice (Bass, Clements & Kazman, 2003)*

La arquitectura o la estructura de un sistema está compuesta por elementos y sus propiedades. Es decir, la arquitectura es algo más estructurado, por ejemplo agrupado en módulos o en diferentes objetos, ocultando propiedades y dando un API públicas de esos objetos, etc. El libro de *Software Architecture in Practice* se centra mucho en la estructura del software para definir la arquitectura.

En el libro «**Software Architecture: Foundations, Theory and Practice**»

**“El conjunto de decisiones principales de diseño tomadas para el sistema.”**

*Software Architecture: Foundations, Theory and Practice (Taylor, 2010)*

Se nos habla sobre el conjunto de decisiones. Ya no tanto sobre la estructura en sí mismo, si no sobre que decisiones importantes de diseños hay que tomar para llegar a buen puerto con este sistema. Las decisiones van a tener que ver con los requerimientos, la calidad que se espera tener de este sistema y van a ser llevadas a través del arquitecto a la implementación del software.

En un caso más moderno, en el libro «**Patterns of Enterprise Application Architecture**»  
**“(...) la arquitectura se reduce a las cosas importantes, cualesquiera que sean.”**

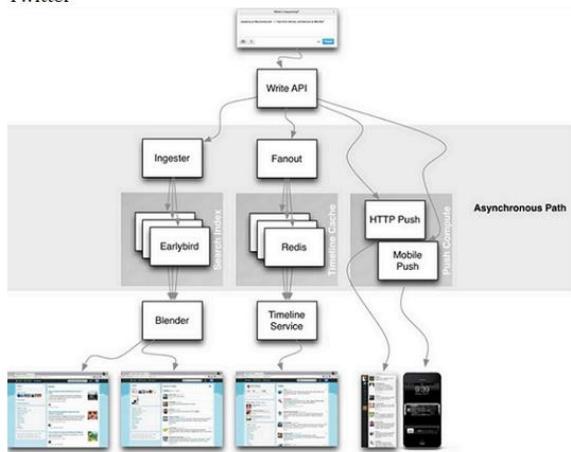
*Patterns of Enterprise Application Architecture (Fowler, 2002)*

Fowler dice que la arquitectura, al fin y al cabo, se reduce a cualquier cosa importante. Es decir, cualquier actividad de diseño o implementación incluso también que sea importante para este sistema va a ser una actividad arquitectura. El planteo de Fowler viene bien arraigado a las metodologías agiles y en la creencia de que la arquitectura emerge de un equipo autogestionado, entonces no hay necesidad de hacer arquitectura por fuera del equipo de desarrollo, si no que sea una actividad que sea constante y que tiene que ver siempre con las decisiones importantes, cualquier que ella sea.

#### Ejemplos sobre arquitectura de software

Para tener una noción de que es lo que solemos ver cuando hablamos de arquitectura de software (diagramas, modelados, etc.), veremos algunos ejemplos:

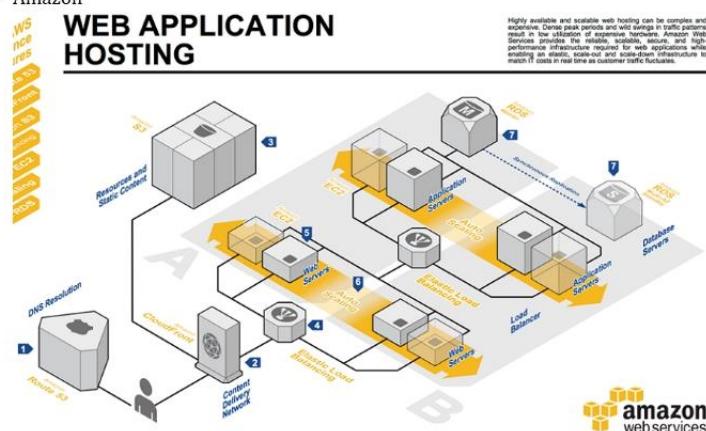
- Twitter



En la arquitectura de Twitter, específicamente la arquitectura de escritura en Twitter o la Write API, tenemos el diagrama que nos explica lo que sucede cuando nosotros damos enviar en su página web.

Lo primero que sucede es que el pedido se envía a una API, diferenciada como un módulo, y ese módulo se comunica con otros para distribuir la información a diferentes clientes o partes interesadas. Una de las partes interesadas se va a encargar de procesar la información para luego armar diferentes consultas que tienen que ver con esta información (por ejemplo, si nosotros buscamos algún hashtag o mención, estos son los servicios que se encargan de conectar el mensaje que acabamos de escribir con ese mismo hashtag o mención), por otro lado tiene un servicio que se encarga de ubicar este mensaje en todas las timeline que sean correspondientes, entonces este servicio procesa este mensaje y al autor de este mensaje y lo conecta con todas las personas que lo siguen para así armar la Timeline específica de cada una de esas personas y la puedan ver en el momento que lo necesiten. Pero si estuviéramos conectado a Twitter ya sea con tweetdeck o con algún cliente móvil también hay un servicio que se encarga de hacer push de este mensaje si nos interesa tenerlo. Entonces, a través de este diagrama Twitter nos explica como resolvió estos problemas que eran importantes para lograr que un mensaje cuando lo escribe un autor le llegue a todas las personas que le interesen ese mensaje.

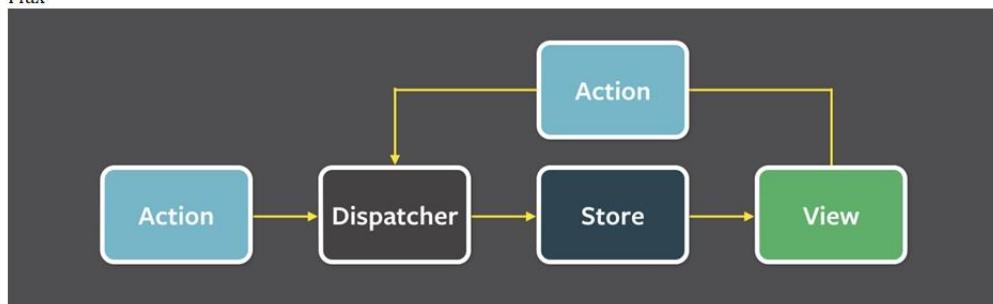
- Amazon



Amazon es otro gran exponente de arquitecturas.

Al ser un proveedor de servicios de infraestructuras y servicios de plataformas provee a sus usuarios de diferentes alternativas de arquitectura para aprovechar al máximo el despliegue de una aplicación. Si nos fijamos en el diagrama ya no habla tanto de los módulos específicos de una aplicación, si no de un sistema de despliegue en donde la aplicación va a vivir en un servidor que quizás es auto escalable y puede estar alojado en un container de Amazon, por ejemplo. Y la base de datos también puede que viva en otro servidor, en otra instancia del container de Amazon o puede que tengamos un servicio de base de datos o un servicio de cache, etc. Amazon se preocupa mucho por la vista arquitectónica que tiene que ver con el despliegue de la aplicación.

- Flux



Nuestro último ejemplo es un diagrama arquitectónico muy pequeño, porque es un diagrama de arquitectura de aplicación, particularmente de la aplicación Flux, la arquitectura de aplicación Frontend de ReactJS.

En este diagrama podemos ver como Flux se concentra en mostrar el flujo de datos, es decir, cuál va a ser la conexión entre diferentes módulos y como los datos van a viajar de un módulo a otro en una sola dirección.

Todos estos diagramas hablan de alguna forma de un sistema de software y lo particular es que cada uno de estos diagramas muestran una vista diferente, arquitectónicamente relevante, pero diferente a la hora de evaluar los criterios o atributos de calidad del software, por ejemplo en Flux vemos que es importante el flujo de datos, mientras que para Amazon es importante la disponibilidad y cómo hacer para distribuir la carga de tu aplicación. Todo eso va a ser super importante a la hora de decidir una arquitectura.

## LA IMPORTANCIA DE LA COMUNICACIÓN - LEY DE CONWAY

Los sistemas monolíticos son disfuncionales, los nuevos modelos consideran la segmentación en equipos/células de trabajo de un organismo mayor, las células se comunican entre sí.

Una empresa genera estructuras que emiten las vías de comunicación de su organización.

### Notas adicionales:

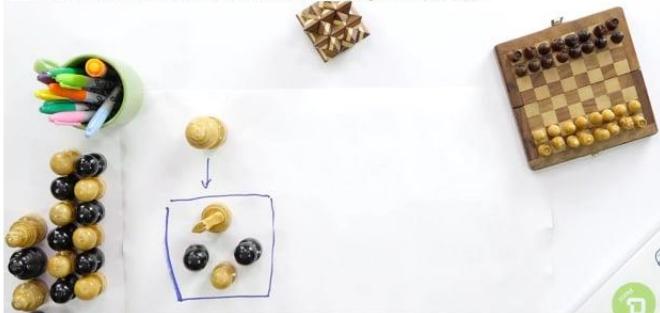
#### La importancia de la comunicación - Ley de Conway

Cuando empezamos un proyecto en general tenemos un primer momento en donde entendemos la arquitectura de lo que vamos a hacer y luego armamos un equipo de trabajo para poder empezar a implementar ese proyecto.

Por ejemplo, tenemos al dueño del proyecto y a nuestro arquitecto.

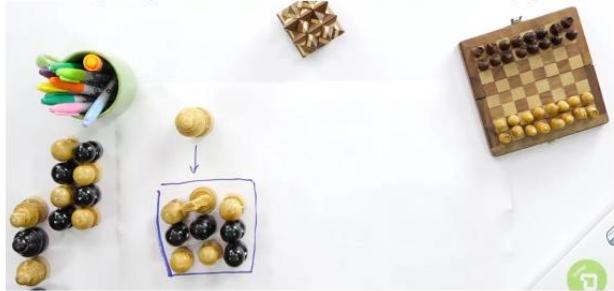


Alrededor de esto iremos armando un equipo de trabajo.



Este equipo de trabajo junto con el arquitecto van a implementar un sistema. Este sistema va a tener una estructura que facilita la comunicación de estas piezas, es importante tener en cuenta que este equipo se va a comunicar entre sí y cada una de estas piezas va a tener comunicación directa con la siguiente, por ende el resultado (la implementación de software) va a tener probablemente esa misma estructura de comunicación. No habrá problemas en hacer una pieza de software y que este se comunique con otro, entonces así también el dueño de producto puede acceder a ese sistema de la misma forma.

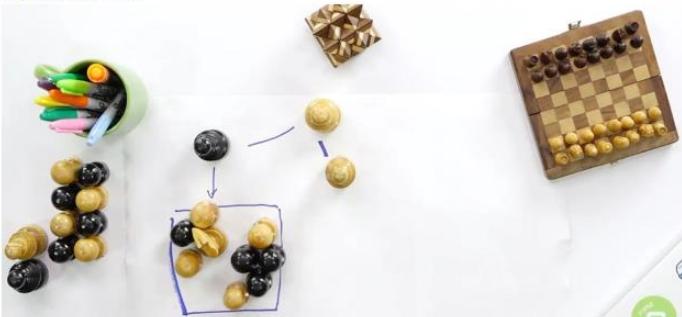
A medida que el producto veremos que necesitamos más equipo de trabajo, así que iremos agregando piezas.



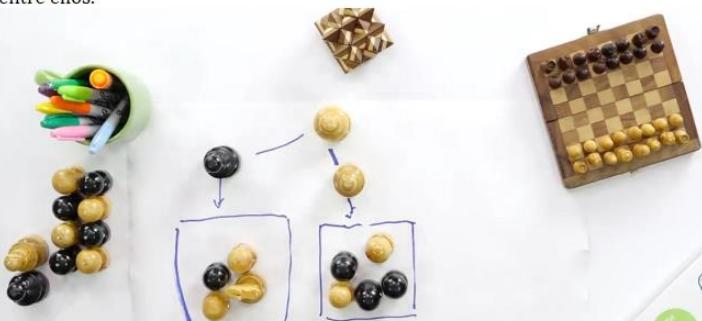
Si nos fijamos, se vuelve muy difícil hacer encajar todas las piezas y la cantidad de comunicación que existe ahora se vuelve mucho más grande. También puede pasar que empiecen a aparecer otros roles de liderazgo, entonces, a medida que estos van apareciendo también lo harán otras necesidades de comunicación.



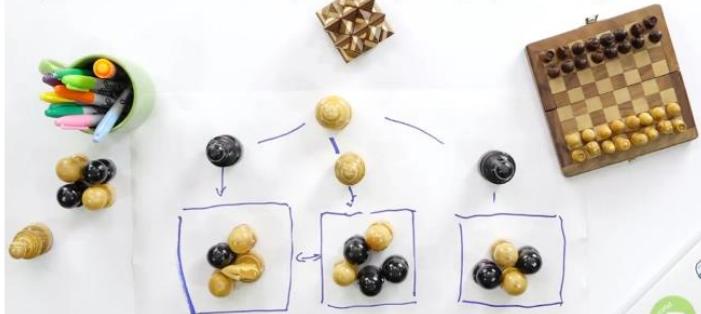
Cuando esto sucede, lo que generalmente se hace es separar en equipos. Entonces, si separamos los equipos y tuviéramos que mantenernos dentro de esa estructura de aplicación, se vuelve difícil encontrar el borde de cuando empieza un equipo y cuando termina el otro.



Cuando esto sucede lo que haremos es mover el equipo entero, que tiene otra responsabilidad, y hacer que se comuniquen solamente entre ellos.



Ahora tenemos una estructura de comunicación un poco más clara en donde tenemos dos equipos, los líderes o gestores de proyecto y al dueño del producto. Estos dos equipos se pueden comunicar cada uno entre ellos y generar una vía formal de comunicación entre sí. Esta estructura de comunicación también va a ser la estructura de la aplicación. La aplicación inicial y luego una nueva aplicación que cumple una nueva funcionalidad, y a mediad que se agregan nuevas aplicaciones se irán formando equipos nuevos.

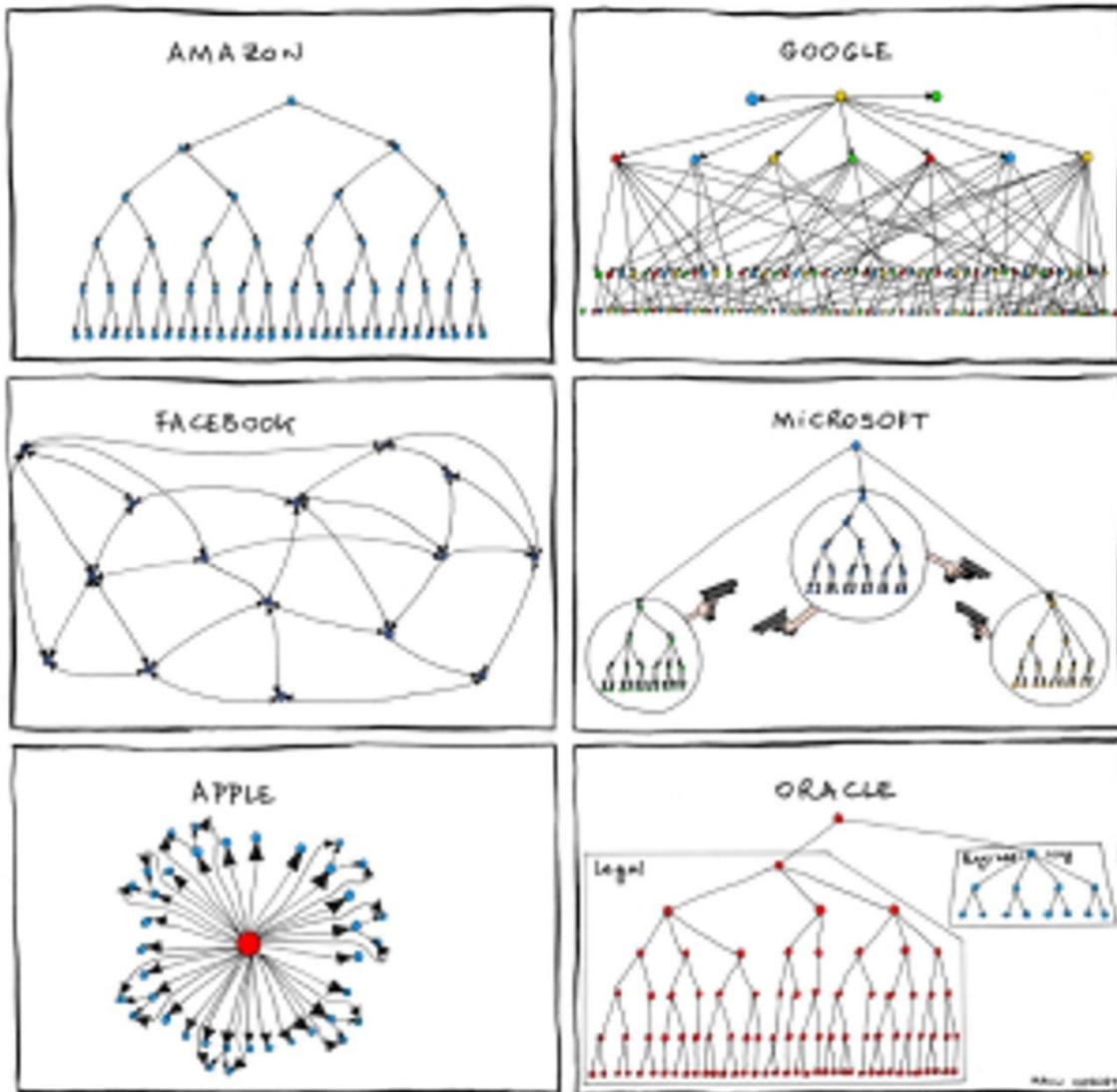


Así iremos pasando de una estructura de comunicación monolítica a una estructura de comunicación distribuida en donde nuestra aplicación puede recibir nuevos productos y generar vías de comunicación entre los equipos que van a imitar las vías de comunicación entre la aplicación.

Esto es lo que se conoce como la **Ley de Conway**, que nos dice que una empresa u organización va a poder generar estructuras que imiten las vías de comunicación de su propia organización. Es decir, podemos tener un equipo que estén todos juntos (como al inicio) y que trabajen en conjunto, comunicándose entre todos generando un sistema monolítico, o podemos también separar los equipos (ya sea artificial u organizacionalmente) y lograr una aplicación distribuida en donde cada equipo sea dueño de su aplicación y luego se comuniquen de una forma más clara.

## La importancia de la comunicación - Ley de Conway

la estructura de las interfaces del sistema será congruente con las estructuras sociales de la organización que produce el sistema



Ejemplos de estructuras de comunicación de algunas organizaciones a simple vista parecen diagramas de arquitectura de software

## OBJETIVOS DEL ARQUITECTO

El arquitecto tiene varias partes interesadas “stakeholders” el cual tiene que conectar esto requerimientos de cada stakeholders con la implementación del sistema.

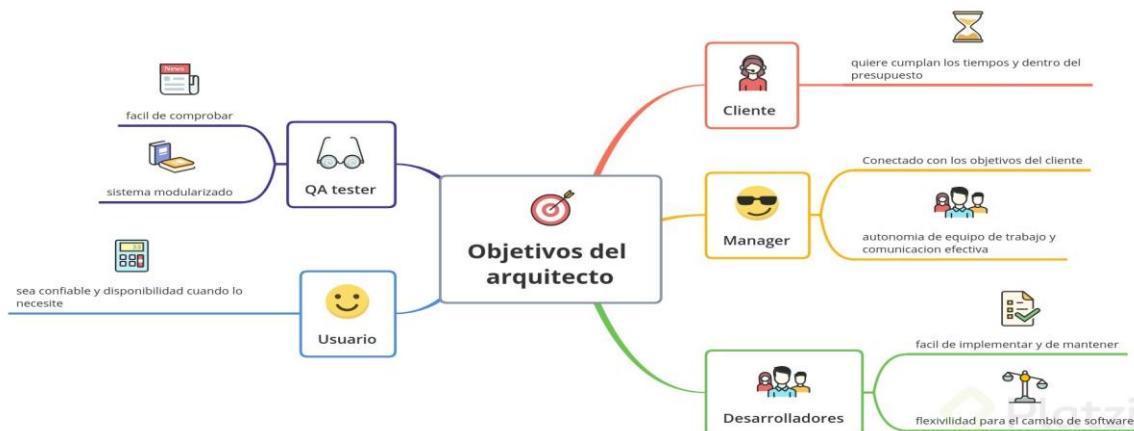
Stakeholders involucrados con diferentes requerimientos:

- Cliente: Entrega a tiempo y que no rebase el presupuesto.
- Manager: Comunicación clara entre los equipos que participan en el desarrollo del sistema
- Dev: Que el desarrollo llevado acabo sea fácil de implementar y mantener
- Usuario: Disponibilidad del producto.
- QA: Fácil de probar.

El arquitecto de software debe gestionar los siguientes puntos para cada Stakeholder:

- Encontrar los riegos más altas que afecten en el desarrollo del sistema (**Cliente**)
- Modularización y flexibilidad del sistema que se está desarrollando (**Manager**)
- Modularidad, mantenibilidad y capacidad de cambio del software (**Dev**)
- Desisdir estrategias para la disponibilidad del sistema (**Usuario**)
- Que el sistema pueda ser modularizado y cada una destas partes pueda ser probado de forma fácil (**QA**).

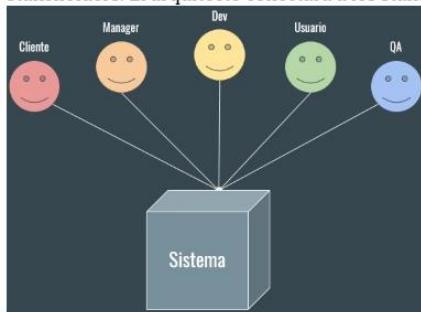
La unión de estos requerimientos (funcionales / no funcionales) va a llevar al arquitecto a tomar decisiones que impactan directamente en el desarrollo del software.



## Notas adicionales:

### Objetivos del arquitecto

Para entender los objetivos que tiene un arquitecto es importante recordar que el arquitecto tendrá diferentes partes interesadas o stakeholders. El arquitecto conectará a los stakeholders y sus requerimientos con la implementación del sistema.



Entre los stakeholders podemos encontrar al **Cliente**, al **Dev o Desarrollador**, al **Gestor o Manager**, al **QA o Tester** y finalmente al **Usuario**. Estos roles tendrán diferentes requerimientos y estos requerimientos afectarán de forma diferente al sistema.

#### Roles y objetivos

- **Cliente:** Tendrá como requerimiento que el sistema esté dentro del presupuesto y sea entregado a tiempo, esto tendrá mucho que ver con encontrar cuales son los riesgos más altos y como evitar que esos riesgos afecten de alguna forma al desarrollo del sistema.
- **Manager:** Mientras que también tendrá en mente el presupuesto y el tiempo, porque está muy asociado a los requerimientos del cliente, también tendrá como requerimiento desarrollar el software de forma independiente, es decir, poder armar equipos que puedan autogestionarse y puedan atacar diferentes partes del sistema. Esto le hablará al arquitecto de la modularización y flexibilidad del sistema que se está desarrollando.
- **Desarrollador:** El requerimiento del Desarrollador es que el sistema sea fácil de implementar y mantener, esto es algo muy importante porque el software no es solamente lo que implementamos la primera vez, sino que es todo el proceso de mantenimiento y evolución. Esto tendrá que ver tanto con la flexibilidad así como con la mantenibilidad y la capacidad de cambio del software.
- **Usuario:** Querrá que el sistema sea confiable y poder usarlo cuando quiera. El usuario requerirá un sistema que, cuando lo busque, este ahí y no falle. Esto tendrá que ver con la disponibilidad del sistema y su confiabilidad, también ayudará a que el arquitecto pueda decir estrategias para atacar esos requerimientos puntuales.
- **QA:** Tendrá como requerimiento un sistema que sea fácil de probar. Es decir, un sistema que cumple ciertos atributos de calidad relacionados a la comprobación; que el sistema pueda estimularse y responder, y que la respuesta sea siempre de forma consistente, que el sistema pueda ser modularizado y que esos módulos puedan ser probados independientemente. Todo esto guiará al arquitecto en decisiones que tengan que ver con la modularización y la capacidad de pruebas.

La unión de todos estos requerimientos, que algunos van a ser requerimientos funcionales y otros van a ser requerimientos no funcionales, llevarán al arquitecto a tomar decisiones de diseño que impacten sobre el sistema a desarrollar.

## ARQUITECTURA Y METODOLOGÍAS

El arquitecto de metodologías tradicionales tiene como objetivo encontrar los problemas y diseñar una solución a gran escala que ataque dichos problemas esenciales del desarrollo.

El arquitecto de metodologías ágiles trabaja con un equipo autogestionado y por ende ven al diseño como un proceso evolutivo y que se va dando sprint a sprint.

### **Etapa del diseño:**

- Definición del problema
- Restricciones
- Requerimientos
- Riesgos  
(El arquitecto no tiene noción de lo que el sistema ya hace.)

Metodologías ágiles: El arquitecto recibe feedback a través de métricas por cada sprint.

### **Notas adicionales:**

#### Arquitectura y metodologías

Ya vimos los objetivos y roles involucrados en el desarrollo del software, y como el arquitecto trabaja con ellos para llegar a cumplir sus objetivos, en esta clase entenderemos el contexto de trabajo del arquitecto y como la arquitectura cambia si la metodología es tradicional o ágil.

Recordemos que la arquitectura como práctica nace en **metodologías tradicionales** en donde el objetivo era principalmente encontrar los problemas y diseñar una solución a gran escala que ataquen esos problemas esenciales o de alto riesgo del desarrollo que se iba hacer. Por otro lado, las **metodologías ágiles** plantean que la arquitectura emerge de un equipo autogestionado y por ende ven al diseño de una solución como algo evolutivo y que se va dando sprint a sprint.

## Diferencias entre la metodología tradicional y la metodología ágil

En las metodologías tradicionales tenemos la etapa de diseño:



En la etapa de diseño están los requerimientos, la definición del problema, las restricciones y los riesgos, todos estos son agentes que ayudarán al arquitecto a tomar decisiones. El arquitecto contará con herramientas de diseño para poder tomar esto como entrada y obtener un modelo de la arquitectura y la documentación para poder implementarlo. Como recordaremos, la etapa de diseño no implementa software todavía si no que le da a la etapa de desarrollo las herramientas para implementar lo analizado en esta etapa.

Algo que falta en el modelo tradicional es el feedback. El arquitecto no tiene todavía nociones de que es lo que sistema ya hace y como el usuario está interactuando con ese sistema. Hasta no hace toda la solución y está sea implementada y desplegada, no tenemos feedback de como esas decisiones son tomadas, si son buenas decisiones o tendríamos que mejorar algunos aspectos en esta decisión, etc. Esto es lo que diferencia fuertemente de las metodologías agiles incluso en el rol de la arquitectura.

Si vemos una arquitectura en la metodología ágil, todo se trata de momentos en donde podemos evaluar o reevaluar nuestras decisiones:



Por ejemplo, cuando hacemos el planeamiento del sprint nosotros planeamos los momentos arquitectónicos importantes, hay una frase en las metodologías agiles que es «el último momento responsable». Este último momento responsable significa que no te adelantes a una decisión que puede ser postergada a menos que este sea el último momento en el que ya tienes que tomar esa decisión. Es decir, una vez que planeado el sprint y definido que es lo que tenemos que decidir arquitectónicamente, ejecutamos, hacemos el sprint en base a las prioridades que tenemos y luego eso se lo llevamos al usuario, hacemos el despliegue y obtenemos el feedback.

### ¿Cómo obtenemos feedback?

A través de métricas y alertas que se disparan con esas métricas.

Por ejemplo, si queremos arquitectónicamente garantizar la disponibilidad de un sistema, tenemos que poder medir, saber cuánto estuvo disponible ese sistema o, más fácilmente, cuánto no estuvo disponible. Y luego, en base a eso, tener una métrica en cuanto si la arquitectura está garantizando o no esa disponibilidad, cuando tengamos esas métricas y hago una retrospectiva de que es lo que está pasando podemos reevaluar esas decisiones de arquitectura y luego volver a poner en el backlog esas decisiones que hay que volver a tomar. Por ejemplo, volver a darle énfasis a la disponibilidad y evaluar que solamente las decisiones que tomamos no fueron suficientes.

Otra cosa importante en el desarrollo ágil es que podemos hacer **esqueletos de solución**, es decir, en vez de plantear ya la solución finalizada y que creemos va a garantizar cierto atributo de calidad podemos plantear una prueba o un esqueleto donde decimos que esta estructura arquitectónica va ayudarnos. Por ejemplo, la flexibilidad de nuestro sistema; implementamos el esqueleto en nuestro sprint, la medimos y la hacemos evolucionar a través del feedback y de nuevas decisiones.

Este esqueleto (en inglés, *tracer bullet*) sirve para hacer arquitectura interactivamente que es algo que se cree no se puede hacer porque la arquitectura es algo muy grande que tiene que ser muy específico, todo esto porque la arquitectura nace en metodologías tradicionales que esperaban que sea así.

### ¿Cómo hacemos para ser arquitectos agiles?

Lo más importantes para ser agiles en general es el feedback, es decir, midamos nuestra arquitectura, volvamos a evaluarla y siempre tengamos en cuenta que es el estado actual de que tenemos y como lo podemos mejorar.

## 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". -> (**análisis del problema**)

El espacio del problema nos ayuda a entender que es lo que vamos a resolver y exactamente como imaginamos como esto va a 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 planificación del software, desde diseño UI, UX hasta diseño de sistemas
- **Desarrollo:** escribir el código, 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 características con un feedback loop (crear, medir, aprender)

### **Notas adicionales:**

Para entender el problema debemos analizar dos partes:

**Espacio del problema:** Esta es la idea principal de lo que se quiere resolver.

- IDEA
- EXITO
- HISTORIA DE USUARIO

**Espacio de la solución:** La forma es que resolveremos el problema que ya hemos detectado,

- Diseño
- Desarrollo
- Evaluación
- Aceptación
- Despliegue

## Entender el problema

A la hora de tomar los requerimientos y procesarlos es importante entender el problema que estamos resolviendo. Para esto tenemos que separar la comprensión del problema de la propuesta de solución. Cuando no separamos esto a veces vemos como parte del problema a resolver ciertas cuestiones tecnológicas como la plataforma, la arquitectura general de la solución o el estilo a implementar cuando en realidad son detalles de implementación.

**RECUERDA:** Para entender bien un problema, tenemos que separar el espacio del problema del espacio de la solución.

### Espacio del Problema vs Espacio de la Solución

Lo primero que tenemos que ver es el espacio del problema:



En el espacio del problema esta principalmente la **idea**, ¿qué es lo que vamos a hacer?, ¿qué es lo que se quiere resolver?. Tomemos como ejemplo a Airbnb, su idea es conectar a los turistas o visitantes con la gente que tiene disponibilidad de una cama, una habitación o incluso un departamento para poder facilitar el hospedaje en una ciudad. Airbnb soluciona el problema complejo de distribuir, donde los hoteles y las cadenas grandes de hoteles tenían el monopolio del mundo del hospedaje, para eso separaron la idea y luego trataron de profundizar el que hacer para implementar esta idea.

El siguiente paso es entender ¿cómo es que yo soy exitoso si esta idea se da?, que es lo que voy a buscar como **criterio de éxito**. En el caso de Airbnb es que más los usuarios elijan cada vez más hospedarse con ellos, que sea más económico, más divertido, más amigable y que conozcan más la cultura del lugar. Todo eso no tiene que ver con el software, tiene que ver con el problema que intentan resolver.

Finalmente, para cumplir con esos criterios de éxito se narran **historias de usuarios**, van a imaginarse como un usuario a través de la solución de este problema puede llegar a tener una experiencia con este sistema. Por ejemplo, una historia en el caso de Airbnb es que como un visitante querría poder encontrar un lugar donde hospedarme que sea más económico que un hotel promedio para poder visitar una ciudad, quizás también poder encontrar un lugar donde hospedarse con referencias culturales o donde pueda conocer gente que vive en esa ciudad para poder profundizar un poco más su experiencia como turista. Todo esto tampoco habla del sistema sino que habla de características del problema que queremos resolver; en los hoteles hay tratos impersonales, el visitante está atado a hacer turismo contratando alguna expedición o tour privado que no te da conexión con la cultura del lugar. Airbnb ve estos problemas y trata de modelarlos donde el usuario consigue un valor agregado con su propuesta.

Después de entender el problema que queremos resolver e incluso entender el problema que NO vamos a resolver (ejemplo, no vamos a hacer hoteles) pasamos al espacio de la solución:



El espacio de la solución consta de un **desarrollo**, de un **diseño**, de la **evaluación** de ese desarrollo y todo esto enmarcado en el concepto de que vamos a solucionar estos problemas con software. Para solucionar estos problemas con software tenemos criterios de aceptación de ese software, es decir, como se va a resolver el problema. Además, vamos a tener que poder garantizar que todas estas historias de usuario, lo que imaginamos que los usuarios pueden hacer, tenemos que garantizar que el sistema esté disponible, que los usuarios tengan acceso a él, y cerrar el ciclo del feedback a través de esa conexión entre el uso de mi aplicación con métricas, con alertas o con login, lo que sea que necesite para entender como los usuarios están usando mi sistema.

Así, el espacio del problema nos ayuda a entender lo que vamos a resolver y exactamente como imaginamos que esto va a agregar un valor a nuestros usuarios sin entrar en detalles de cómo lo va a resolver el sistema. Mientras que el espacio de la solución entra en el detalle va a reflejarse en ese espacio del problema y tratar de resolverlo teniendo en cuenta todos los detalles técnicos que sean necesarios. Y, muy importante, tratando de evitar resolver problemas que no queremos resolver.

## REQUERIMIENTOS

Una vez que entendemos el espacio del problema y el espacio de la solución, vamos a entrar a analizar los requerimientos de nuestro sistema.

**Requerimientos de producto:** Los podemos dividir en 3.

- Capa de requerimientos de negocio, son reglas del negocio que alimentan los requerimientos del negocio.
- Capa de usuario, tienen que ver en cómo el usuario se desenvuelve usando el sistema, qué atributos del sistema se deben poner por encima de otros.
- Capa Funcional, se ven alimentados por requerimientos del sistema, ¿qué cosas tienen que pasar operativamente? Esta capa se ve afectada por las restricciones que pueden afectar operativamente a lo funcional.

**Requerimientos de proyecto:** Tienen que ver más con el rol de gestor de proyectos, se usan para dar prioridad a los requerimientos del producto.

Estos dos mundos de requerimientos hablan de las prioridades del equipo de trabajo del proyecto.

**Requerimientos de producto:**

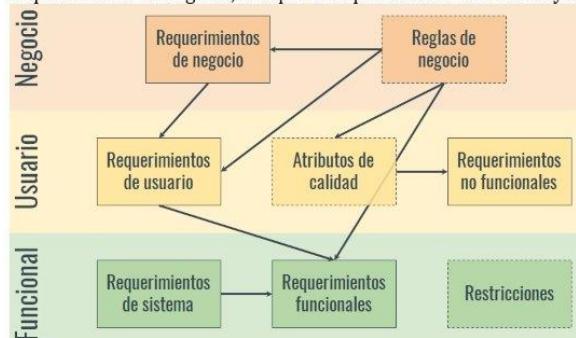
- **Requerimientos funcionales:** Tienen que ver con las historias de usuarios, que hablan sobre específicamente lo que hace el sistema, por ejemplo que usuario ingrese al sistema.
- **Requerimientos no funcionales:** son aquellos que agregan cualidades al sistema, por ejemplo que el ingreso de ese usuario sea de manera segura.

## Notas adicionales:

### Requerimientos

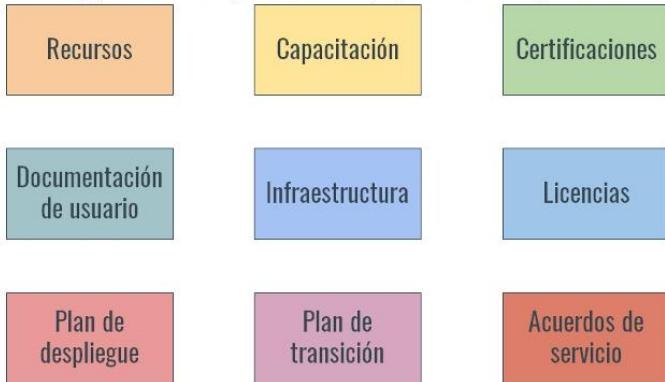
Una vez que entendemos el problema (el espacio del problema y el espacio de la solución) es hora de entrar en detalle con los requerimientos de nuestro sistema. Para eso vamos a separar los requerimientos en dos grandes mundos: los **requerimientos de producto** y los **requerimientos de proyecto**.

En los **requerimientos de producto** entraremos más en detalle sobre que necesita nuestro sistema y que stakeholders será que este diciendo esto. Para entrar en detalle tendremos que separar, nuevamente, a los requerimientos de producto en tres: la capa de requerimiento de negocio, la capa de requerimiento de usuario y la capa de requerimiento funcional.

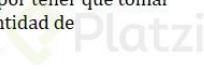


- Capa de requerimiento de negocio: Conta de reglas de negocios que alimentan los requerimientos del negocio. Tomemos a Airbnb como ejemplo, sabemos que Airbnb quiere hospedar y un negocio, o mejor dicho, un problema será conectar a los hospedadores con los visitantes. La regla de negocio tendrá que ver con esa conexión, así que una regla de negocio podría ser que el sistema permita tener diferentes tipos de usuarios; el usuario que brinda hospedaje y el usuario que está visitando la ciudad.
- Capa de usuario: Tienen que ver en cómo el usuario se desenvuelve usando el sistema, estos requerimientos de usuario también tendrán que ver con los atributos de calidad, es decir, que atributos del sistema se deben poner por encima de otros. Por ejemplo, si en Airbnb fuera importante proteger a quien está ofreciendo un hospedaje, entonces, ese usuario va a tener un requerimiento en donde se necesitará cierta seguridad; conocer las identidades de los visitantes, brindar un pago anticipado con tarjeta de crédito y eso también impactará con los requerimientos no funcionales como la seguridad del sistema, es decir, garantizar que el sistema le da al usuario ese atributo de calidad. Se enfatiza esa seguridad de que podrán usar el sistema sabiendo que se cumplirá lo esperado.
- Capa funcional: Se ve alimentado por todo lo anterior para entender que es lo que debemos hacer específicamente, aquí ya no hablamos de una historia genérica ni de un valor de negocios que tiene que ver con la estrategia general de nuestra empresa, si no que hablamos puntualmente de que es lo que debe hacer este sistema para implementar esa funcionalidad.
  - o Los requerimientos funcionales se ven alimentados por los requerimientos del sistema, es decir, ¿qué cosas tienen que pasar operativamente?. Por ejemplo, si el sistema cobra automáticamente a una tarjeta de crédito luego de confirmar un pago o si un administrador puede contactar directamente con un visitante o una persona que ofrece alojamiento.
  - o La capa funcional se verá afectada por las restricciones que tienen que ver con lo funcional, es decir, si desarrollamos una aplicación web tendremos restricciones de comunicación y tendremos restricciones sobre como interactuarán los usuarios con nuestra aplicación que tiene que ver con el mundo de la web. Esto no es así con una aplicación móvil que podría ejecutarse en el dispositivo móvil sin necesidad, en un principio, de conectarse a internet, luego, en algún momento, también podemos hacer una conexión a internet para que pueda conectarse a un servidor remoto. Todo esto tiene que ver con las restricciones contextuales de estos requerimientos, es decir, si el requerimiento una aplicación será que pueda utilizarse remotamente a través de la internet, y afectará a la implementación de los requerimientos funcionales, los requerimientos de usuario y los requerimientos del sistema, todos a la vez.

Por otro lado tenemos a los **requerimientos de proyecto** que no tienen que ver mucho con la arquitectura si no con lo que sería las fechas de entregas, planes, etc. No hurgaremos mucho en estos requerimientos porque tienen que ver más con el gestor de proyectos. El arquitecto, mientras que tendrá que estar atento a estos requerimientos, tiene que poder trabajar junto a un gestor de proyecto para entender y priorizar los requerimientos de proyecto con los requerimientos del producto.



Por ejemplo, si necesitáramos obtener recursos como disponibilidad de servidores, estaciones de trabajo para los programadores o incluso poder almorzar en la oficina y cosas así, el gestor de proyecto será quien se encargue de esto, el arquitecto no tendrá mucho que ver con ello. Hay algunos requerimientos de proyecto que afectarán el desarrollo del sistema más que nada por tener que tomar decisiones que quizás nos gustaría tomarlas en un nivel más detallado, pero que por restricciones de tiempo, cantidad de programadores o incluso de capacitación y entrenamiento, no podremos tomar en el momento que nos gustaría.



Entonces, un arquitecto puede, con el equipo y las prioridades de negocios que tiene actualmente, decidir una arquitectura temporal para luego evolucionarla dado que requerimiento de proyecto sea más prioritario. Por ejemplo, si vamos a una feria o exposición y necesitáramos una beta de nuestro sistema disponible ya que sin eso el negocio entero podría caerse y entonces no podríamos priorizar algunos atributos de calidad en la arquitectura si dejamos de lado la importancia de este requerimiento de proyecto.

Estos dos mundos de requerimientos hablan mucho sobre las prioridades de nuestro proyecto, que el arquitecto junto con el gestor van a trabajar sobre eso, en metodologías ágiles en general van a trabajar sobre el backlog de nuestros sprints.

Volviendo a los requerimientos de producto, separaremos o enfatizaremos los requerimientos **funcionales** de los **no funcionales**:

- **Requerimientos funcionales:** Detallan específicamente como el sistema se va a comportar bajo un estímulo, como va a implementar una historia. Por ejemplo, si nosotros como usuario registrado queremos poder acceder al sistema para poder tener un contenido personalizado, esto es un requerimiento funcional, el sistema tiene que poder permitir a un usuario ingresar con su nombre de usuario y su contraseña.
- **Requerimientos no funcional:** son aquellos que agregan cualidades al sistema. Por ejemplo, cuando digo que como usuario quiero poder ingresar al sistema de forma segura para poder tener un contenido personalizado estoy agregando una calidad que es la seguridad, estoy hablando de ya no querer ingresar solamente diciendo el nombre de usuario porque pueden llegar a quitarme mi identidad, a obtener información sensible y personal. Queremos enfatizar que sea seguro, que lo que estamos usando tiene seguridad. La seguridad es un atributo de calidad y está relacionado con un requerimiento no funcional.

Los requerimientos no funcionales suelen ser parte de los requerimientos funcionales, pero a veces es algo más global. Por ejemplo, en algunos estilos específicos de la arquitectura, como la arquitectura web, se suele dar como algo más general, como que todo contenido personalizado debe ser encriptado. Eso habla de seguridad, pero no habla específicamente de un requerimiento funcional, si no que habla ya de todo el sistema.

Estos requerimientos funcionales y no funcionales siempre van a ser importante para el arquitecto, sin embargo, el arquitecto tiene como responsabilidad el priorizarlos. Siempre va haber requerimientos no funcionales o atributos de calidad más importantes que otros. Por ejemplo, si tuviéramos un sistema médico, como para un personal de enfermería, y quisieramos poder **obtener datos del paciente** para poder tratarlo correctamente, suena como un **requerimiento funcional** coherente. Sin embargo, si quisieramos esa información **en tiempo real**, estamos dando una prioridad a la eficiencia y disponibilidad del sistema, considerando también el marco de lo que de lo que es la medicina y los sistemas de contexto médico probablemente este **requerimiento no funcional** sea muy importante a la hora de desarrollar el sistema y el arquitecto debería tener muy alta prioridad contra otros como pueden ser la modularización del sistema u otros atributos de calidad que quizás no sean tan pertinentes en este contexto.

Estos atributos de calidad, de todas formas, siempre deberían surgir de la toma de requerimientos, del análisis de que es lo que se espera que haga el sistema, tanto funcionalmente como no funcionalmente, es decir, como se espera que el sistema se comporte más aún en situaciones adversas, como por ejemplo mucha carga, atacantes cuando quieran vulnerar nuestro sistema, etc.

Los requerimientos no funcionales siempre estuvieron muy vinculados con la arquitectura, sin embargo, en algún momento de la historia del desarrollo de la arquitectura de software no se tenía tan en cuenta a los requerimientos funcionales. Para unir estos dos conceptos y tener en cuenta funcionales como no funcionales se habla de **requerimientos significativos** para la arquitectura, en donde se agrupan cualquier tipo de requerimiento que afecte a la hora de diseñar la arquitectura correcta.

- Los requerimientos funcionales detallan cómo el sistema se va comportar bajo un estímulo.
- Los requerimientos no funcionales detallan cualidades del sistema; pueden ser características más globales como atributos de calidad,

#### Ejemplos de requerimientos no funcionales:

- Rendimiento.
- Disponibilidad.
- Durabilidad.
- Estabilidad.
- Accesibilidad.
- Adaptabilidad.
- Capacidad.
- Integridad de datos.

## RIESGOS

Los riesgos son importantes para priorizarlos y atacarlos en orden y asegurar que las soluciones arquitectónicas que propongamos resuelvan los problemas más importantes.

Intenta tratar los riesgos con posibles escenarios de fracaso y que pasaría en caso de que ese riesgo se haga real.

Veamos como identificar los riesgos:

**En la toma de requerimientos -->** dificultad / complejidad

**En los atributos de calidad -->** incertidumbre, cuanto mas incertidumbra hay, mas alto es el riesgo.

**Conocimiento del dominio -->** Riesgo prototípico, son aquellos que podemos atacar de forma estándar.

Una vez que tenemos los riesgos identificados, debemos priorizarlos, recuerda que no es necesario mitigarlos todos, debemos siempre tener en cuenta y dar prioridad a aquellos riesgos que ponen en peligro la solución que se está construyendo.

### **Notas adicionales:**

Se deben tener en cuenta los riesgos al momento de implementar el sistema. Estos riesgos son importantes para priorizarlos y atacarlos en orden, esto garantiza que las decisiones arquitectónicas que se están tomando solucionan los problemas más importantes.

Para trabajar orientado a los riesgos los primeros es describir los riesgos, es decir, encontrar los posibles riesgos que el sistema puede tener y como los va a resolver.

Una vez descrito el riesgo se debe entender de que consta este riesgo si es un riesgo del sistema de ingeniería y se debe mitigar a través del diseño y la implementación del sistema o es un riesgo de gestión del proyecto.

## Riesgos

Más allá de los requerimientos que tomemos a la hora de definir un problema y encontrar una solución, también tenemos que tener en cuenta los riesgos que corremos a la hora de implementar el sistema. Los **riesgos** van a ser muy importantes para poder priorizarlos y atacarlos en orden, y que eso nos garantice que las decisiones arquitectónicas que tomamos solucionan los problemas más importantes y no cualquiera que hallamos encontrado.

### ¿Cómo hacemos para trabajar orientados a riesgos?

Lo primero es describir ese riesgo, es decir, encontrar esos riesgos que creemos el sistema va a tener y como los va a resolver. Es importante encontrar los riesgos tratando de describir que pasaría si fracasáramos, que pasaría si no llegáramos a cubrir ese riesgo.

Por ejemplo, si tuviéramos un sistema que trabaje con picos de tráfico, un riesgo sería que el cliente experimente latencias mayores a cinco segundos, eso sería un riesgo enorme que podría hacernos perder dinero e incluso, si contáramos con acuerdos con otras empresas, tendríamos que respetar esto por contrato. Otro ejemplo sería que un atacante pudiera obtener información confidencial del sistema a través de un «**Ataque de intermediario**» (en inglés, *Man in the Middle*). Un ataque de intermediario es cuando se hacen pasar por nosotros para que una persona obtenga información sensible estando en el medio entre el sistema y el cliente. En situaciones web esto es un estándar de seguridad y por eso se utiliza el https.

Una vez descrito el riesgo tenemos que entender de que consta ese riesgo; si es un **riesgo de ingeniería**, que podemos mitigarlo a través del diseño y la implementación del sistema, o es un **riesgo de gestión del proyecto**, que pueden ser que proyecto no llegue a tiempo, no tener suficientes recursos o incluso no haber planificado bien el despliegue de la aplicación. Los riesgos de gestión del proyecto no van a tener que ver tanto con la arquitectura, pero es importante tenerlos en cuenta también porque tanto como los requerimientos van a ser parte del planeamiento.

### ¿Cómo identificamos riesgos?

Hay frameworks para identificar riesgos que se basa en la toma de requerimientos, en los atributos de calidad y en cuanto sabemos o no del dominio del problema que estamos resolviendo.



- En la toma **requerimientos** es importante entender si es un requerimiento complejo, es decir, si la dificultad de resolver este requerimiento es muy alta, esto va directamente relacionado con un riesgo.
- En los **atributos de calidad** es importante entender si sabemos o no sabemos cómo mejorar un atributo específico. Por ejemplo, si detectamos que un requerimiento no funcional es importante la seguridad, ¿sabemos resolver ese problema de seguridad? ¿cuánto sabemos de seguridad?. Cuanto más incertidumbre detectamos en algo que es importante, más alto es el riesgo de esa situación.
- En **conocimiento de dominio** tenemos que saber si lo que estamos implementado ya ha sido implementado o no. ¿Por qué? Porque los dominios conocidos suelen tener riesgos prototípicos, es decir, si vamos a implementar un dominio como, por ejemplo, el de aplicaciones web distribuidas hay riesgos que podemos atacar de forma estándar. El Man in the Middle es uno de ellos.

A través de frameworks podemos detectar todos estos riesgos que quizás al empezar este proyecto o al imaginarse la solución no habíamos visto.

Entonces, ¿qué hacemos con esto? Como dijimos, tenemos que priorizarlos.

### ¿Porque priorizar riesgos?

Priorizar riesgos es importante porque generalmente no podemos resolver todos, si nos concentraremos en resolver riesgos que no eran importantes, es decir, riesgos que si fracasamos no es tan relevante para el éxito o fracaso del sistema, entonces estaremos invirtiendo mucho tiempo en algo que al fin y al cabo no era tan relevante. Sin embargo, debemos tener siempre en cuenta que riesgos ponen en peligro el éxito o fracaso de la solución. Priorizamos estos riesgos y entendemos, tanto nosotros como nuestros stakeholders, que algunos riesgos no van a poder ser cubiertos en el primer momento, si no que vamos a postergar el ataque o la mitigación de esos riesgos para cuando podamos invertir tiempo en ellos.

Así los riesgos y los requerimientos van a ser priorizados, y van a poder ser parte de nuestro plan ordenado en donde entendemos que es lo más importante arquitectónicamente para resolverlo.

## **RESTRICCIONES**

Las restricciones en el contexto de un proceso de desarrollo de software se refiere a las restricciones que limitan las opciones de diseño o implementaciones disponibles al desarrollar.

Los SH nos pueden poner limitaciones relacionadas con su contexto de negocio, limitaciones legales.

También hay limitaciones técnicas relacionadas con integraciones con otros sistemas.

El ciclo de vida del producto va a agregar limitaciones al producto, por ejemplo a medida que avanza el proceso de implementación el modelo de datos va a ser más difícil de modificar.

El arquitecto debe balancear entre los requerimiento y las restricciones.

### **Notas adicionales:**

“Una restricción limita las opciones de diseño o implementación disponibles al desarrollador”.

Software Requirements 3rd Edition (Wiegert, Betty, 2013)

Cuando empezamos a trabajar, no vamos a tener disponibles todas las tecnologías, plataformas, patrones de diseño, infraestructura, porque hay ciertas restricciones que nos da el contexto.

Usualmente, estas restricciones están dadas por las partes interesadas [stakeholders], que nos van a poner limitaciones que tengan que ver con su ecosistema, o su contexto de negocio, o regulaciones o legales.

Limitaciones por integraciones con otros sistemas, por ejemplo, la aplicación necesita conexión a internet, eso nos va a limitar en la capacidad o contexto de despliegue.

El ciclo de vida: A medida que crece la aplicación es más difícil de cambiar.

## Restricciones

Cuando hablamos de requerimientos también mencionamos las restricciones.

### **¿Qué son las restricciones?**

En el contexto de un proceso de desarrollo de software, las restricciones se refieren a las limitaciones en las opciones que vamos tener tanto de diseño como de implementación.

**“Una restricción limita las opciones de diseño o implementación disponibles al desarrollador.”**

*Software Requirements: 3rd Edition (Wiegers, Betty, 2013)*

Es decir, que cuando comenzemos a trabajar, ya sea para diseñar la solución o implementarla, no vamos a tener disponibles todas las tecnologías, plataformas o incluso algunos patrones de diseño o arquitectura porque tenemos ciertas restricciones que nos da el contexto.



En general el contexto está dado por los stakeholders de nuestra aplicación, ellos nos pondrán limitaciones que quizás tengan que ver con su ecosistema, su contexto de negocio, regulaciones o cuestiones legales.

También tendremos limitaciones con respecto a las integraciones con otros sistemas. Por ejemplo, si vamos a comunicar el sistema a través del http necesitaremos poder tener una conexión, poder acceder a la internet, y eso nos limitará la capacidad de despliegue o el contexto de despliegue de nuestra aplicación.

Por último, el ciclo de vida del producto va agregando ciertas limitaciones a las opciones que tenemos. Un ejemplo muy común es que, a medida que evoluciona el sistema, el modelo de datos sea más difícil de cambiar, porque quizás tendremos un modelo de despliegue en el que tenemos que tener varias versiones combinadas o soportadas al mismo tiempo. Entonces, todo recurso común, sea base de datos, cash o motor de búsqueda, tiene que ser compatible con varias versiones de nuestra aplicación en ese contexto. Esas restricciones que tenemos no nos permitirán, por ejemplo, hacer un cambio en una tabla y eliminar una columna que siga siendo utilizada en la versión anterior de nuestra aplicación.

Así, el arquitecto tendrá que balancear entre los requerimientos y las restricciones que van a ir encontrando a medida que se toman decisiones tanto de diseño como de implementación.

## **RETO: CLASIFICACIÓN DE REQUERIMIENTOS Y ANÁLISIS DE RIESGOS**

El reto que tienes es tomar un sistema conocido (del trabajo, algún proyecto propio o un sistema que hayas usado del que conozcas su arquitectura).

Describir qué problemas resuelve y cuáles son sus requerimientos no funcionales.

Si tuvieras que hacer de este sistema un “producto reutilizable” en otros escenarios:

¿Cómo cambiaría su arquitectura?

¿En qué otro escenario debería repensarse completamente?

¿En qué otros escenarios se mantendría?

compártenos en el sistema de discusiones.

## ESTILOS DE ARQUITECTURA

### ARQUITECTURA, PANORAMA Y DEFINICIÓN

#### **¿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. Así también, hay decisiones arquitectónicas tales como si empezar un proyecto con un monolito o iniciar con una estructura de microservicio, que se dan por sentado que cualquier cosa sería de mucho más valor iniciar con 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 más 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 páginas de internet implementan diferentes arquitecturas. Sin embargo, todas esas páginas son una página 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 está resolviendo del dominio del problema, sino de que problema está 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 implementó anteriormente.

## Notas adicionales:

### Arquitectura, panorama y definición

En este módulo hablaremos de **estilos de arquitectura**, para eso debemos entrar en panorama de que es lo que está pasando arquitectónicamente en el software, entenderemos que es un estilo particularmente de un software y entraremos en detalle de varios estilos conocidos que puedan ayudarnos a tomar decisiones arquitectónicas mucho más claras.

#### ¿Cuál es el panorama arquitectónico actual?

Pasa que hay muchas librerías, muchos frameworks y mucho conocimiento arquitectónico implícito en las comunidades.

Por ejemplo, si hablamos de ABC o flags con reats estaremos hablando de arquitectura. Sin embargo, está implícito en el uso de una tecnología, si de repente hablamos flags en Java o C# no tendría sentido porque no es una arquitectura en la que se suele encontrar esas tecnologías, pero arquitectónicamente tienen el mismo valor y podría ser implementados en otras tecnologías. También hay decisiones arquitectónicas, como empezar el proyecto con un monolito o con una estructura de micro servidores, que se dan por sentadas, como si cualquier proyecto fuera mucho más valioso empezarlo por un micro servidore porque es la tendencia, porque se da más fácil para el equipo de desarrollo o porque las herramientas más modernas están más orientadas a un microservicio.

Eso provoca una falta de análisis más profundo sobre qué define ese estilo, ese patrón de arquitectura, y cuáles son los sacrificios que estaremos haciendo al usarlo y cuáles serían los beneficios que esperemos nos traigan.

Ningún patrón de arquitectura solo trae beneficios, recordemos que "no hay balas de platas", es decir, ningún patrón o estilo nos va a solucionar todos los software. Siempre habrá beneficios y consecuencias en las decisiones de diseño que tomamos.

#### ¿Qué es un estilo de arquitectura?

Cuando hablamos de estilo de arquitectura nos referimos a algo genérico. Por ejemplo, podríamos entrar en detalle sobre diferentes páginas web como Facebook, Twitter, WordPress e incluso Wikipedia, todas estas son páginas web que implementan diferentes arquitectura, pero siguen siendo una página web, es decir, tienen una estructura cliente-servidor en donde tenemos un navegador web que a través del sistema DNS y el HTTP logra conseguir un documento en formato HTML que se muestra al cliente. Esta estructura genérica es lo que define el estilo de una arquitectura en donde no hablamos en detalle sobre qué problema se está resolviendo, si no que está hablando de que el problema se resolverá arquitectónicamente a nivel de conectores entre diferentes aplicaciones, como en nuestro ejemplo reciente en donde tenemos un navegador web y un servidor o una red peer-to-peer en donde dos sistemas tratan de comunicarse, también podría ser en una aplicación móvil donde tratan de comunicarse con un API a través del HTTP o HTTPS.

Todo esto define algo genérico que podemos reutilizar a través de diferentes software, son estilos de arquitectura que ayudaran a poder reutilizar el conocimiento y aprender sobre soluciones anteriores que tuvieron éxito implementando esas comunicaciones, implementando esos componentes con esos conectores.

Entonces, si tuviéramos que ponerlo en una definición tendríamos lo siguiente:

**Un estilo de arquitectura es una colección de decisiones de diseño, aplicables en un contexto determinado, que restringen las decisiones arquitectónicas específicas en ese contexto y obtienen beneficios en cada sistema resultante.**

*Software Architecture: Foundations, Theory and Practice* (Taylor, 2010)

En «**Software Architecture: Foundations, Theory and Practice**» nos dice que el estilo de arquitectura es una colección de decisiones arquitectónicas (decisiones de diseño) que, dado un contexto, nos permite ya restringir las decisiones arquitectónicas, es decir, nos da un set de decisiones tomadas y nos restringe el resto de las decisiones arquitectónicas para un beneficio ya estimado.

Resumidamente nos dice que podemos usar estas decisiones ya tomadas en el pasado y que tuvieron éxito, y aplicarlas a nuestro sistema que comparte un contexto similar en espera de tener un éxito similar al que tuvo el que la implementó anteriormente.

## **ESTILOS: 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**

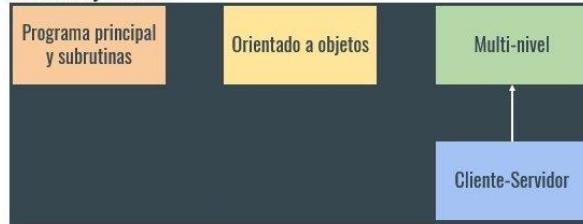
Una aplicación le habla a un servidor mediante una petición y servidor responde. Si ahondamos en este modelo el nivel de detalle aumenta. Aplicación le pregunta a REST/API y luego a server, y server devuelve.

## Notas adicionales:

### Estilos: Llamado y retorno

En los estilos de arquitectura de Llamada y Retorno los componentes van a ser invocados por otros componentes y esperar una respuesta.

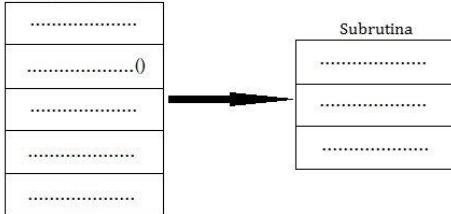
#### Llamada y Retorno



El estilo de arquitectura **Programa Principal y Subrutinas** es el estilo más básico evolucionado de un script. Al principio de la programación lo que teníamos eran instrucciones ordenadas y que el programa secuencialmente ejecutaba una por una, pero a medida que los programas se fueron complejizando era necesario poder reutilizar bloques enteros de programas, entonces se hacían instrucciones de salto donde iban a un punto y luego volvían después a donde estaban antes. La extracción de esto fueron las **subrutinas** o las **funciones**, donde podíamos modelar un bloque de código, ponerle un nombre y luego invocarlo desde algún punto. Esto es lo que solemos aprender en general cuando empezamos a ver un lenguaje de programación como Python, JavaScript, PHP o incluso C o C++ de una forma muy básica.

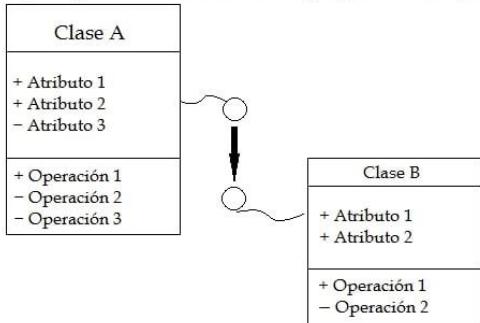
Entonces lo que tenemos es un programa principal en donde en cada línea podemos hacer alguna instrucción que llama a una subrutina, y esta subrutina va tener sus instrucciones y puede llegar a tener un dato de retorno o no dependiendo de lo que necesitemos. El programa continuara hasta que ser terminan las subrutinas ejecutadas.

#### Programa Principal



Por ejemplo podemos encontrar todavía programas de subrutinas en contexto scripting para deployment (*despliegue de software*) o cuando tenemos un programa muy pequeño que debe ejecutarse varias veces. Entonces, supongamos que tenemos un script en Python que nos permite tener ese código de forma rápida y luego a medida que el código se va a complejizando, necesitaremos empezar a configurar el programa o a tener una mejor estructura por lo que empezamos a estructurarlo como un subprograma, un programa principal y sus subrutinas.

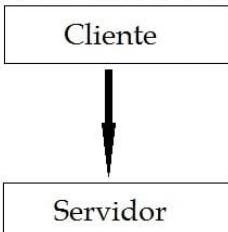
El estilo de arquitectura **Orientado a Objetos** es un estilo muy común que se usa para hacer aplicaciones que serán necesarias mantener por mucho tiempo. En el estilo Orientado a Objetos se trata de juntar el estado de la aplicación con el comportamiento asociado a ese estado en particular, para eso hacemos objetos que tendrán un comportamiento específico, es decir, una interfaz pública y un estado interno que idealmente no lo revelaremos a menos que sea necesario pedir algún detalle. Comparado con programas de subrutinas significa que la extracción ya no va a ser el llamado a una función, sin embargo va a seguir siendo un llamado, tenemos objetos que se van a llamar entre sí y esperar una respuesta entre ellos.



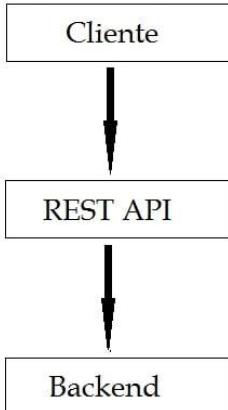
Por ejemplo, el típico escenario Orientado a Objetos en general los objetos son representados por clases que tendrán un estado interno y funcionalidades. A su vez, van a coexistir con otros objetos y cuando las instancias de estos objetos A y B se comuniquen, es decir, el objeto A llama al objeto B, como por ejemplo el método del objeto A inicial llama al primer o segundo método del objeto B, se delega la ejecución y el objeto B es el que tendrá que cumplir todo lo necesario por hacer y dar o no un retorno dependiendo del método, pero devolviendo la ejecución al objeto A con la que se inició.

En el estilo de arquitectura **Multinivel** tendremos distintos componentes que se comunicaran entre sí en un orden específico. El nivel superior será el que inicia la aplicación, tendrá su propia lógica, y en algún momento llamará a un nivel inferior, este nivel inferior a su vez tendrá su propia lógica y podrá llamar a un segundo nivel inferior, así sucesivamente estos componentes pueden ser de un programa o incluso podrían ser múltiples programas que se ejecutan independientemente.

Un caso específico de multinivel son las arquitecturas Cliente-Servidor en donde tendremos cliente que hace un pedido a un servidor y espera un resultado de ese servidor.



Por ejemplo, imaginemos una aplicación móvil que cuando la abrimos en el teléfono se empieza a ejecutar y a ocupar espacio de memoria, puede hacer algunas cosas sin necesidad de llamar a algún servidor. Sin embargo, en algún momento necesitaremos datos y que pueda acceder a internet, para eso llamamos a un servidor, probablemente una API que puede ser una REST o un GraphQL o lo que sea que hallamos implementado, y va a devolver esos datos como respuesta a ese pedido. Esta es una arquitectura Cliente-Servidor básica con la aplicación más común que el internet y el API. Sin embargo, este servidor podría tener también algo más por debajo, por ejemplo podríamos hacer que esto no sea solamente un servidor sino que sea una aplicación que represente nuestra REST API y tengamos otra aplicación por debajo que sea nuestro backend.



Si solo miramos una sección, como el Cliente y al REST API o el REST API y el Backend, como una sola cosa podemos seguir entendiendo esta arquitectura como una arquitectura Cliente-Servidor. Pero si entramos en detalle y vemos todos los componentes que están siendo usados, entonces podemos ver que esta arquitectura es una multinivel de tres niveles. En principio son equivalentes, sin embargo depende del nivel de detalle que estamos analizando podríamos tener más valor en ver los tres niveles juntos.

## ESTILOS: FLUJO DE DATOS

Este estilo se utiliza cuando tenemos un proceso que tiene que tener una salida clara; pero esa misma salida puede segmentarse en partes. Partes que ya se sabe que hay que hacer.

### Lote Secuencial:

- Se trata de la ejecución de una pieza de código, Ya procesa y asegurase de que esta pase a otra etapa o proceso.

### Tubos y Filtros:

- Es un String o un flujo de datos continuo, en donde cada aplicación recibe esos datos. los procesa. y los envía como salida a otra aplicación o quizás. ya hasta al final de la ejecución.

### Notas adicionales:

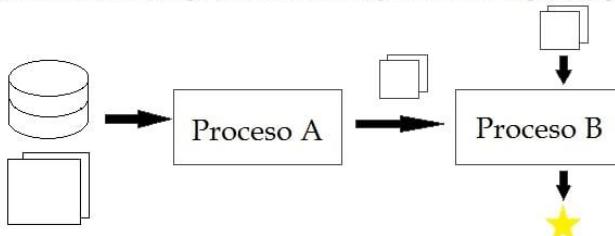
#### Estilos: Flujo de datos

En el estilo de arquitectura **Flujo de datos** no estamos tanto preocupados por cual es la secuencia de ejecución, sino de como los datos van a ir fluyendo de un punto a otro.



En los flujos de datos tenemos la arquitectura de **Lote Secuencial** en donde lo importante es ejecutar una pieza de código y que al final esa, ya toda procesada, pase a una siguiente etapa. Mientras que en la arquitectura de **Tubos y Filtros** tenemos un flujo de datos contiguo en donde cada aplicación recibe continuamente los datos, los procesa y los pasa como salida a otra aplicación o, quizás ya, al final de la ejecución.

En la arquitectura de **Lote Secuencial** solemos tener aplicaciones que tienen algo para procesar y que suelen ser costoso hacerlo. Por ejemplo, en podemos tener como entrada una base de datos de ventas, es decir, la lectura de esa base de datos o algunos archivos en donde hicimos una exportación de archivos y los tenemos disponibles para procesarlos.

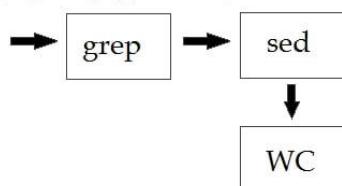


Del archivo procesado por Proceso A dará como salida otro set de archivos. Este nuevo set de archivos, ya sea de forma automática o manual, será procesado por Proceso B. Entonces, podemos decir, por ejemplo, que en el Proceso A tenemos que sumar toda la cantidad de ventas de este mes y tener como salida el resultado, mientras que en el Proceso B podríamos tener como entrada también los procesos de los meses anteriores y hacer todo un cálculo de la totalidad de ventas en el año.

El Proceso A hace un cálculo muy costoso, porque tal vez el volumen de datos es muy grande, y en el Proceso B se usa esa información agregada para hacer un nuevo cálculo y una nueva salida que puede ser consumida por el usuario.

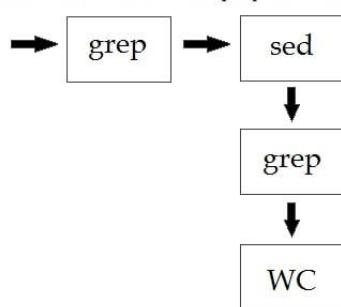
En la arquitectura de **Tubos y Filtros** lo que tendremos son diferentes aplicaciones que van a estar conectadas generalmente en tiempo real, así ya no necesitaremos la interacción del usuario para decidir cuándo empieza un proceso y cuando termina el otro.

Es típico de las aplicaciones Linux el tener una entrada que, por ejemplo, puede ser una cadena de texto que podemos procesarla con el programa grep y esto nos generara otra cadena de texto nueva.



En este caso quizás estamos filtrando líneas que no nos interesan, esta cadena luego puede entrar a otro programa sed que reemplaza el texto de una línea y nos da como salida otra cadena de texto. Finalmente podríamos usar un programa como WC (word count) para contar la cantidad de palabras que contiene la String final.

Como vemos, estamos construyendo a través de pequeñas aplicaciones una aplicación más grande que recibe una cadena de texto y, dado un texto específico y un reemplazo que hacemos, cuenta la cantidad de líneas finales. Filtrar, reemplazar y contar, la aplicación está compuesta por partes separadas, así que podemos modificarla agregando un programa nuevo como por ejemplo otro grep para filtrar nuevamente casos que pudieron haberse generado en el ciclo nuevo.



Incluso sin programar algo nuevo, sino que simplemente reconfigurando la aplicación y reutilizando, podemos construir una nueva funcionalidad. Esto es muy potente y se usa mucho en Script Time cuando accedemos a un servidor y vemos que sucede con login, así como también después de dejar programas separados para que puedan hacer acciones de forma repetitiva y no tengamos que estar pensando en filtrar, leer o reemplazar.

#### ¿Cuándo usamos el estilo de arquitectura de flujos de datos?

Cuando tenemos un proceso que tiene que tener una salida clara, pero puede ser separada en donde tenemos parte en parte lo necesario para hacer. Así como cuando, dado una cadena de texto de entrada, irla procesando parte por parte y teniendo una salida al final de todo ese proceso.

## ESTILOS: CENTRADAS EN DATOS

### Pizarrón:

- El pizarrón es el núcleo de la arquitectura. Donde componentes externos a él se encargarán de procesar un dato y escribirlo en el pizarrón (Este funciona como centralizador). Cuando el pizarrón ya tiene todos los datos necesarios; el mismo podría generar una salida, Ejemplo: Sistema Fiscales

### Centrado en base de datos:

- Es un estilo común; Se trata de que una cantidad de componentes comparte una misma base de datos. de Ejemplo: aplicaciones que poseen comunicación por Internet.

### Sistema experto Basado en reglas:

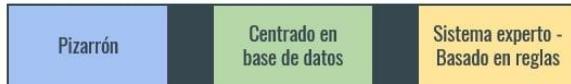
- Este sistema no se ve muy seguido en aplicaciones modernas; un componente A (Tipo Cliente) consulta a uno B, donde este se encargará de tratar de entender si la petición del cliente es una **consulta** o **regla**. Para que el componente B logre resolver la petición se va a comunicar con un tercer componente © este trabajara como KDB: **Knowledge DataBase**.

## Notas adicionales:

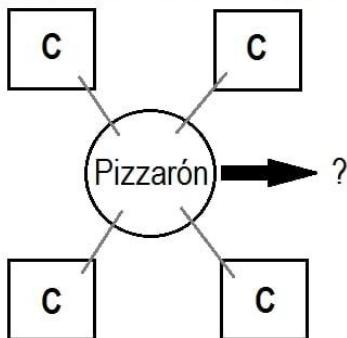
### Estilos: Centradas en datos

En los estilos de arquitectura **Centrados en Datos** lo que podemos observar es que nuestra aplicación tendrá múltiples componentes, pero algunos se concentraran en cómo hacer para almacenar los datos, cómo hacer para tenerlos disponibles y cómo hacer incluso para que sean los datos correctos, es decir, si son o no datos pertinentes a la acción que quiere hacer el usuario.

#### Centrado en Datos

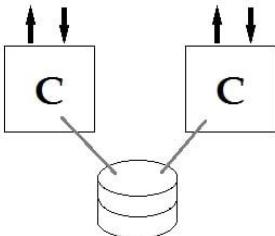


En el estilo de **Pizarrón** tenemos son diferentes componentes que interactuaran con un componente central. El componente central será el pizarrón, ¿Cuál es el uso de este pizarrón? Cada componente tendrá como responsabilidad el procesar, calcular o recibir algún dato y escribirlo en el pizarrón. El pizarrón por un lado funcionara como centralizador de la información y por el otro tendrá la lógica en la cual, cuando ya se sabe que se tiene todos los datos necesarios y se puede obtener una salida, esta salida podría ser un cálculo o algo que necesita que todo esto suceda.



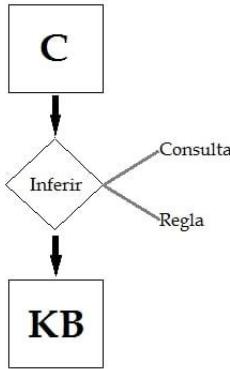
Un ejemplo de Pizarrón sería cuando tenemos un proceso donde hay varias partes involucradas y necesitemos esperar que todas esas partes hagan su trabajo hasta obtener la información necesaria. Por ejemplo, en los sistemas fiscales en lo general tenemos un Cliente que estará haciendo un reporte de sus consumos o ingresos del año y luego esos datos lo ingresar al pizarrón. Incluso podríamos tener un proceso automatizado que ya vaya detectando si hay alguna anomalía, por ejemplo, que compare con los ingresos de años anteriores o que vea que no haya ningún tipo de fraude de forma automatizada. De otro lado, más allá de automatizarlo, en general estos procesos tienen revisión manual; otro componente va a comunicarse con un operador de este sistema fiscal, va a leer lo reportado por nuestro cliente y va a esperar la autorización de este operador para al fin y al cabo dar como resultado que esta persona está reportando correctamente sus tributos al estado.

El estilo de arquitectura **Centrado en Datos** es un estilo mucho más común, más usado que el Pizarrón, y es un estilo familiar para los que tienen aplicaciones que usan base de datos y de repente se tiene una segunda aplicación que usa la misma base de datos. En este caso tendremos componentes y una base de datos compartida, lo importante de este estilo es que cualquiera de estos componentes para comunicarse, en vez de comunicarse entre sí, deciden escribir a la base de datos que es pertinente para el otro componente y que luego el otro componente lo pueda leer.



Un ejemplo típico son las aplicaciones que tienen comunicación por internet; un componente puede implementar la parte REST API, mientras que el otro componente puede ser la aplicación web en sí misma. Esto, mientras se puede ser así, en general lo vemos como componentes separados dentro de un mismo gran componente monolítico que consume la base de datos. Sin embargo, a medida que evoluciona esa aplicación quizás es pertinente separar ese monolito y ver cómo hacer para compartir el recurso de la base de datos sin necesidad de ser una aplicación monolítica y poder ser desplegada de forma independiente.

El estilo de **Sistema Experto o Basado en Reglas** es bastante particular y no se lo ve muy seguido en aplicaciones modernas, sin embargo, tiene sus usos y es importante conocerlos. En este estilo lo que vamos a tener es algún componente de tipo Cliente que se comunicara con un componente que va a tratar de inferir las reglas, es decir, va entender que es lo está queriendo decir el cliente y va a ver si esto es una regla o una consulta. Luego, para poder resolver esto, se comunicara con un tercer componente que es la base de conocimiento (en inglés *Knowledge Base* o *KB*).



Lo que logramos es que este componente pueda ir escribiendo lo que es una regla, los conocimientos que tenemos en nuestra aplicación, y luego, cuando tenemos una consulta, pueda construir esa consultar y hacerla a nuestra base de datos de reglas. De esta forma no tenemos estructurado ya de antemano cual es el esquema de base de datos que necesitamos, sino que iremos rellenando la base de datos de conocimiento a medida que vamos descubriendo que es lo que necesitamos hacer y luego podemos consultarla.

Por ejemplo, en una sistema de inteligencia artificial es común que no sepamos el esquema de datos que vamos a procesar ni sepamos cuales son las reglas a tener en cuenta, en general eso es algo que se espera que el propio sistema pueda hacer. Entonces, a través de interacciones en el sistema, podemos ir generando una base de datos de conocimiento, desestructurado pero conocimiento al fin, y luego poder hacer consultas en base a lo que ya sabemos es la verdad.

En los estilos de arquitectura basados en datos siempre vamos a tener esta concentración de cómo hacer para consultar o guardar algo, en cada uno de estos estilos el dato tiene diferentes formas o diferentes importancias, en el primer caso es simplemente para centralizar la comunicación, mientras que en el segundo tenemos procesos claros en donde sabemos cuántos datos faltas y cuantos datos necesitamos. En el último caso vimos también como los datos pueden no ser conocidos de antemano y es importante ver si ese es nuestro escenario para ver cuál es el estilo que necesitamos usar.

## **ESTILOS: COMPONENTES INDEPENDIENTES**

Existen dos grandes familias que tienen que ver con la forma en que se comunican los componentes. (la invocación implícita y explícita)

### **Invocación implícita:**

Se tienen varios componentes y se cuenta con un BUS de eventos en el cual los componentes van a escribir algunos eventos y luego el BUS los comunica a los componentes que les conciernan dichos eventos. Existen varios tipos de BUS.

#### **Tipos de BUS:**

**Publicar/suscribir:** en donde el componente inicial es el que publica un evento y los componentes que están suscritos reciben la notificación de dicha publicación.

**Orientado a servicios (Enterprise Service BUS):** en este caso el BUS es un componente inteligente, en el que los componentes notifican al BUS a través de eventos y el BUS decide a quien notificar dichos eventos.

### **Invocación explícita:**

Se tienen componentes desarrollados individualmente pero que todos se conocen entre sí, dichos componentes tienen que publicar qué vía de comunicación existirá para comunicarse entre ellos y a qué se dedica cada uno.

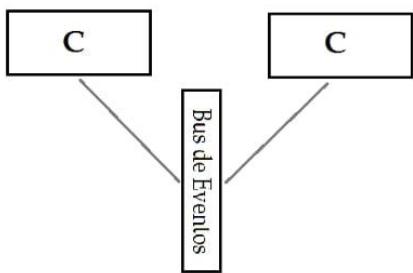
## Notas adicionales:

### Estilos: Componentes independientes

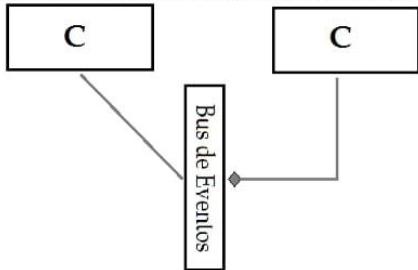
En los estilos de arquitectura **Componentes Independientes** veremos cómo podemos desarrollar aplicaciones en donde cada una de esas aplicaciones se pueda hacer independientemente, es decir, que no tengamos acoplamientos fuerte entre cada uno de estos componentes.



Una de las dos grandes familias dentro de los Componentes Independientes es la **Invocación Implícita**, suele ser basada en evento y habla sobre como suele hacer para que nuestros componentes puedan mandar mensajes entre sí sin que un componente conozca exactamente a quien le está hablando. En general, cuando tenemos eventos lo que tenemos son varios componentes y lo que se conoce como **Bus de Eventos**. En el bus de eventos estos componentes van a publicar o escribir algunos eventos, y luego el bus notificara de estos eventos a otros componentes que estén interesados en saber que sucedió.

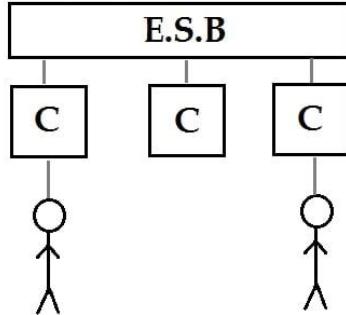


En el caso del bus de eventos tenemos dos familias grandes, una de ellas se llama **Publicar - Suscribir**, en donde el componente inicial es el que publica un evento y luego otros componentes están suscritos y reciben ese evento.



Un ejemplo sería si tuviéramos un componente que publica y otro componente que suscribe. Por ejemplo, nosotros publicaríamos cada vez que se vende un producto en un E-Commerce, entonces el primer componente podría ser la aplicación del E-Commerce, y cada venta se notificara al bus (que producto se vendió, el precio de venta, etc.). Por otro lado, podríamos también contar con una aplicación Reporting que escuchara estos eventos y los guardara en su base de datos independientes, datos de reporte que no son tan importantes para el sistema E-Commerce (cuánto dinero se ganó, que vendedor vendió el producto, etc.). De esta forma, el componente Reporting se registrara en el bus, estará suscripto a un evento, y el E-Commerce va a publicar ese evento en el bus.

La otra familia es **Orientación a Servicios**, lo que hace es tener un bus inteligente en donde se notifica a través de eventos al bus, pero el bus es el que decide a quien le va a decir que hacer. En este caso tenemos un bus inteligente que en general para cuando queremos buscar una implementación buscamos lo que se llama Enterprise Service Bus o E.S.B., que son buses que se pueden programar internamente. Entonces, estos buses van a tener registrado componentes a los que le va hablar o le van hablar a él. El E.S.B. a través de su programación va saber el proceso que hay que llevar a cabo para lograr el objetivo. Sin embargo, los componentes no se conocen entre sí, así que logramos hacer esta invocación implícita entre ellos.



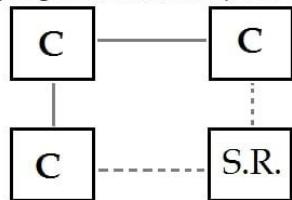
Entonces, si continuamos con el ejemplo anterior de una aplicación fiscal, podemos imaginar que el primer componente es el componente que se conecta con el usuario y que recibe los formularios de este usuario. Este componente procesa ese formulario y publica un evento de "formulario recibido", el E.S.B. recibe ese formulario y lo delega al componente de procesamiento de formularios automático para ver si esta persona es alguien que está queriendo hacer fraude o si es alguien que está contribuyendo más de lo que debería o menos de lo que debería. Por otro lado, y quizás en paralelo, el E.S.B. puede hablar con el componente de revisión de formularios y que este haga que se dispare una revisión de nuestro operador que a su vez va a volver a publicar el evento de "revisión finalizada", "revisión aprobada" o "revisión rechazada". Luego así podríamos volver hacia atrás, al componente anterior, y notificarle al usuario si su publicación correcta o si tiene que volver a publicar.

En este esquema, a diferencia de un esquema Centrado en Datos, el bus es el que tiene la inteligencia y a su vez, a diferencia del Publicar y Suscribir, los componentes no saben mucho de este proceso. Cada uno se dedica solamente a resolver sus problemas y reciben las instrucciones del E.S.B.

#### Invocación Explícita



La **Invocación Explícita** habla sobre cómo hacer para desarrollar componentes que si se conocen entre sí, pero que sean desarrollados independientemente. Esto significa que estos componentes van a tener que de alguna forma publicar cual es la vía de comunicación que tienen disponibles y, a su vez en general en orientación a servicios y en servicios de tipo Enterprise, tenemos algún tipo de registro de servicios, es decir, cada uno de estos componentes siendo un servicio se registran a un punto central y luego dicen como es que se pueden comunicar entre sí. Entonces, un componente le pregunta al registro central "¿hay algún componente que resuelva este problema?", y le dice "sí, es este. Esta en esta dirección, comunícate así", entonces va y se comunica. La invocación es explícita, simplemente no están acoplados, no están dependientes al momento del despliegue, están desplegados independientemente y luego se encuentran en ejecución.



Un ejemplo típico de esto es la **Center Price**, más específicamente las Center Price de los últimos 10 años para atrás, lo que solían hacer era desarrollar productos independientes y en algún momento interconectarlos. Para interconectarlos no tenían la capacidad o potencia en su momento de hacer eventos y guardar toda la información que necesita un bus de eventos modernos, sino que hacían conexión directa entre los componentes. Podríamos tener un componente que se dedique a hacer un E-Commerce y luego se comunique con un componente que se dedique a hacer los pagos; entonces, el E-Commerce se concentra solamente en como publicar los productos y cuáles son los clientes que tenemos en este momento, y los precios de los productos y luego a la hora de cerrar la compra delegan al componente de pago para que pueda procesar esa venta. En aplicaciones modernas incluso hacemos esto, pero lo hacemos delegando a un componente externo que podría ser una plataforma de pago.

Por ultimo podríamos tener componentes que se dediquen al reporte o a la operación que sería entender los productos que están disponibles, publicar nuevos productos o incluso modificar los precios o crear promociones. Todo esto podría ser un componente que maneja ese estado interno sabiendo quienes son los usuarios que están autorizados a hacer esto y cuáles son las acciones que están disponibles, y comunique esto a la plataforma E-Commerce separada de la gestión. Esta arquitectura nos da la capacidad de hacer despliegues independientes, sin embargo, las API's que tiene que exponerse a estos componentes son sensibles, es decir, por más que lo despleguemos independientemente, tenemos que garantizar que la hora de comunicarse no va haber una ruptura de compatibilidad, este componente tiene que poder siempre saber cómo hablarle a este otro componente. Es importante esto, porque de esta forma tenemos que garantizar que cada vez que cambiamos nuestra API's o cada vez que cambiamos el código no estamos dejando de funcionar para ciertos clientes.

## COMPARANDO ESTILOS: ¿CÓMO ELIGO?

### Estilos monolíticos.

Estilos donde se despliegan un artefacto de software

1. **Eficiencia:** Al tener un solo artefacto se puede ser optimizado de manera más personalizada. // En estilos distribuido, es un problema debido a los canales de comunicación, red, internet que comunican los componentes.
2. **Curva de aprendizaje:** El monolítico contiene toda la información allí. Un monolítico bien diseñado permite tener todas las piezas en el mismo lugar, por lo que se facilita la lectura y entendimiento. // En el caso distribuido hay que entender cada componente. Nota: Un componente interno en un distribuido puede ser visto como un monolítico. Es la base de los microservicios.
3. **Capacidad de prueba:** Son más fáciles de probar una funcionalidad de principio a fin. // En distribuidos necesito tener todos los componentes disponibles, incluyendo los BUS de eventos.
4. **Capacidad de modificación:** Un cambio que se despliega todo junto garantiza menos estados intermedios. Las versiones nunca coexisten // En distribuidos diferentes componentes tienen diferentes versiones, por lo que requiere de compatibilidad entre versiones. Una modificación en un distribuido es más difícil hacer llegar.

## **Estilos distribuidos.**

Componentes que luego de ser desplegados se conectan de alguna forma.

1. **Modularidad:** Es separar componentes que prestan servicios
2. **Disponibilidad:** Es mayor que en monolítica, podemos tener multiples copias de un componente, que esté disponible significa que sea más barato, tener una copia entera de un monolítico es mucho más caro que copiar el componente distribuido que necesitamos que escale. Microservicios aprovecha recursos.
3. **Uso de recursos:** Es más fácil gestionar los recursos del sistema
4. **Adaptabilidad:** Al ser distribuido se puede detectar mucho más fácil qué componente necesita ser adaptado del sistema y es más fácil realizar esa actualización // en monolíticos puede ser mucho más complicado, como lanzar una app en un sistema operativo diferente.

## **¿Como elijo qué necesito?**

Tener en cuenta los requisitos, los objetivos de negocio / arquitectura de software, atributos de calidad/ Estrategias de arquitectura, Escenarios/ Desiciones arquitectónicas. Con el fin de analizar qué sacrificios, riesgos y no riesgos cuento y cómo impacta en mi proyecto

## Notas adicionales:

### Comparando estilos: ¿Cómo elijo?

Para comparar los estilos de arquitectura que acabamos de ver, lo separaremos en dos estilos:

- **Monolítico:** Estilos de arquitectura en donde vamos a desplegar un solo artefacto de software.
- **Distribuido:** Estilos de arquitectura en donde cada despliegue es independiente y luego cada componente se interconecta entre sí.

Estilo Monolítico

Eficiencia	Curva de aprendizaje
Capacidad de Prueba	Capacidad de Modificación

En este estilo de arquitectura veremos que es mucho más fácil darle prioridad a la **eficiencia de ejecución**, al ser un solo artefacto de software puede ser optimizado, se puede ser bien detallista en cómo se comunica usando memoria interna o usando llamados a procedimientos. Esto es más difícil en estilos distribuidos, en donde la comunicación entre dos componentes probablemente tenga que ir por red o por algún tipo de protocolo de comunicación.

También, los estilos monolíticos son **mucho más fáciles de probar**. En el caso de que tengamos que probar una funcionalidad de principio a fin, con el estilo distribuido tenemos que tener todos los componentes disponibles y quizás también un bus disponible, y esto a veces se complica en entornos de desarrollo local. Mientras que las pruebas unitarias o las pruebas más pequeñas se pueden hacer en cualquier de estos dos estilos, el estilo monolítico tiene más facilidad de prueba de principio a fin.

En el estilo monolítico, la **curva de aprendizaje suele ser más suave** porque el monolítico tiene toda su información ahí. Obviamente no estamos hablando un monolito mal diseñado o en donde se hallan tomado malas decisiones en los nombres de los métodos o variables, sino que estamos tratando ser agnóstico a las pequeñas decisiones y dando la misma calidad de desarrollo, tener todas las piezas que componen al monolito en un mismo lugar. Mientras que en el caso distribuido, para poder entender la gran escala, es decir, entender toda la aplicación en su conjunto se tendría que entender cada uno de los componentes. Hay que tener en cuenta que cuando hablamos de monolíticos y distribuidos, un componente interno de un estilo distribuido se puede ver como un componente monolítico. Por ejemplo, en los microservicios se suele decir que es más fácil aprender a trabajar en estos porque cada microservicio es muy pequeño y esto hace que aprender de ese servicio sea más fácil, pero si aislamos el servicio podemos decir que en realidad ese servicio en sí es un monolito, es un artefacto que se despliega todo junto. Entonces, esto es lo que queremos dar a entender cuando decimos que la curva de aprendizaje de una aplicación monolítica es más suave que en el estilo distribuido.

Finalmente, en el estilo monolítico, la **capacidad de modificación es más fácil**. Esto puede ser polémico porque acabamos de decir que los microservicios son más fáciles de mantener, sin embargo, estamos hablando de cambios que involucran a varios componentes a la vez. Entonces, si los componentes en el estilo monolítico están todos dentro de una misma base y se despliegan todos juntos, va a ser más fácil garantizar que no hay estados intermedios, es decir, que la versión actual y la versión siguiente nunca coexisten porque los desplegamos todo juntos. En los estilos distribuidos, al ser los componentes desplegados independientemente, van a poder ser versionados independiente y tendremos que mantener compatibilidad para atrás, dependiendo de nuestro esquema de despliegue, quizás a más de una versión a la vez. Entonces, si existe una modificación para toda la aplicación en su conjunto, es más difícil hacerlo en un estilo distribuido que en un estilo monolítico.

Estilo Distribuido	
Modularidad	Disponibilidad
Uso de Recursos	Adaptabilidad

En el estilo distribuido veremos que **es natural modularizar** porque necesitaremos definir cuáles son los componentes que estaremos desplegando independientemente, sin embargo, en un estilo monolítico todos sus módulos estarán internamente y, mientras que podremos modularizar, es mucho más fácil romper esa modularización y de esa forma no garantizamos la separación en el largo plazo.

La **disponibilidad** de un estilo distribuido es mucho mayor al de un estilo monolítico. En el esquema distribuido podremos tener múltiples copias de cada servicio, cada componente, y a su vez cada componente va a ser más pequeño, entonces la disponibilidad será más barato. Esto también lo veremos en el **uso de recursos**, porque en el estilo monolítico, mientras que puede llegar a ser menor a su equivalente distribuido, a medida que tengamos que escalar el tener una copia entera del monolito resulta más caro que poder copiar solamente el componente distribuido que necesitaremos que escale. De esta forma, el microservicio, que es una arquitectura que se está utilizando mucho, puede escalar mucho más fácil y aprovechar mucho mejor sus recursos.

Por último, la **capacidad de adaptar el sistema** para que se despliegue en contextos diferentes va a ser mucho más simple en estilos distribuidos, en esquemas donde tenemos los componentes se pueden desplegar independientemente en otros contexto, que en estilos monolíticos en donde todo nuestro monolito tiene que ser adaptado a ese nuevo contexto. Naturalmente, el nuevo contexto puede ser similar y entonces el monolito no tiene mucho problema para adaptarse, o pueden ser tecnologías totalmente diferentes, por ejemplo tener que desplegar una misma aplicación en Linux, Windows o Mac puede ser un desafío muy grande cuando la aplicación es monolítica, o si la aplicación es distribuida, únicamente debemos detectar cual es el componente que necesita ser adaptado y no todo el sistema, es mucho más fácil adaptar ese componente solo que va a ser mucho más pequeño y más manejable.

### ¿Como elijo que estilo de arquitectura necesito para mi aplicación?

A estas alturas ya deberíamos de entender que requerimientos, cuáles son los requerimientos funcionales y no funcionales, podríamos también saber cuáles son las restricciones que tenemos, y los riesgos asociados a este desarrollo.



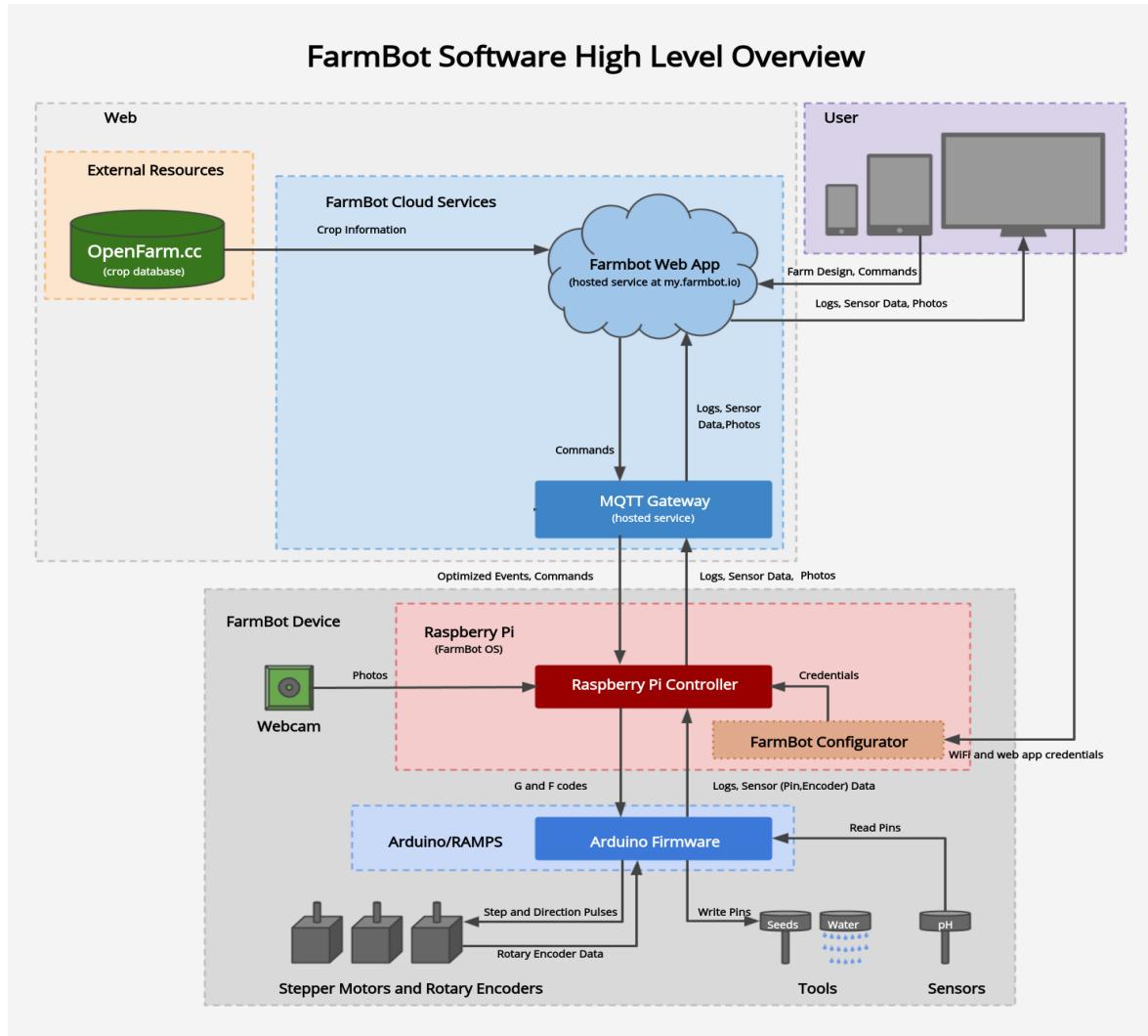
Veremos como conectar esos riesgos, restricciones y requerimientos para llegar a atributos de calidad y escenarios de arquitectura, esas herramientas nos van a permitir diseñar una arquitectura específica para nuestro problema y de esa forma poder analizar si esa arquitectura resuelve o no resuelve todo esto que tenemos analizado. Este tipo de cosas van a permitirnos hacer un loop y hacer de nuevo una arquitectura sabiendo ya que trade-offs, es decir, que sacrificios vamos a estar haciendo para cada uno de esos requerimientos y a su vez ver que cosas que creímos que eran riesgos realmente lo eran o quizás no lo eran.

## RETO: UN PRODUCTO, MUCHOS ESTILOS

Esta vez el reto es que tomes un sistema conocido y haz lo siguiente:

- Describe sus atributos de calidad percibidos.
- ¿Estás de acuerdo con las decisiones tomadas?
- Propón el énfasis en algún atributo de calidad.
- Describe qué se vería favorecido
- ¿Qué se vería afectado negativamente?

Recuerda contarnos en el sistema de discusiones.



## **DESARROLLO DEL PROYECTO**

### **DESARROLLO DEL PROYECTO: PLATZISERVICIOS FASE STARTUP**

**Situación/problema que se nos presenta:** Estamos en la ducha y de repente se rompe nuestra cañería!

**Disparador:** Cómo encuentro un plomero de confianza? A partir de allí, empezamos a trabajar en una idea de software

\*\*Primero \*\*comenzamos con los **requerimientos del sistema:**

#### **Criterios de éxito:**

- (Para el cliente) Conectar rápidamente a un cliente con un profesional de confianza.
- (Para el profesional) Garantizar el aumento del volúmen de trabajo al profesional.

**Idea:** Definición de una forma ideal de como se satisface una necesidad.

Ejemplo: Tener una forma mucho más sencilla de solicitar un servicio de plomería que llegue a mi casa con un plomero que se conozca.

\*\*A partir de los criterios de éxito, vamos a intentar encontrar las Historias de usuario: \*\*

- Como cliente necesito contactar un profesional en el momento para reparar un problema en mi hogar.
- Como cliente necesito conocer la experiencia del profesional para decidir a cuál contacto.
- Como profesional necesito cobrar mi trabajo realizado para continuar prestando el servicio.
- Como profesional necesito ampliar mi cartera de clientes para tener más flujo de trabajo.

#### **Requerimientos (más técnicos):**

Ciclo de prestación de servicio:

- Solicitar, aceptar y finalizar una prestación de servicio de forma segura.

Comunicación:

- Capacidad de búsqueda y comunicación rápida entre clientes y profesionales disponibles.

Evaluación:

- Capacidad de evaluar profesionales y clientes para referencia futura.

Riesgos: Son referentes a historias de los usuarios.

Ejemplos:

- El cliente utiliza un servicio y no completa el pago en un tiempo determinado
- Un profesional llegó a la puerta de mi casa y no puedo confirmar que sea quien dice que es
- El proyecto no está terminado para la feria de profesionales independientes de Mayo 2019.

\*\*Restricciones: \*\*Limites que tiene nuestro proyecto de acuerdo a variables.

Ejemplo:

- Recursos disponibles para el desarrollo: Programadores, equipos, energía, lugar de trabajo, etc.
- Registro de impuestos del profesional.
- Garantía de profesionales sin antecedentes penales.

Teniendo en cuenta todas las restricciones y requerimientos que existe, tratar de entender qué estilo arquitectónico vamos a utilizar. Tratamos de encontrar la forma más simple de empezar a trabajar: Una arquitectura cliente-servidor montada en la web en donde podamos aprovechar toda la infraestructura de la internet.

### **Desarrollo del proyecto: PlatziServicios Fase Startup**

Arrancaremos como arquitectos en un proyecto real llamado PlatziServicios que tratará sobre resolver un problema muy específico que nos pasa en la vida real. Tengamos como ejemplo que estamos en la ducha y de repente la cañería se rompe, ante esta urgencia debemos de conseguir un plomero de confianza. Es esta dificultad de encontrar un plomero que se vuelve el disparador, la idea que nos hace empezar a trabajar en el software que resuelva este dilema, para eso primeramente debemos realizar el análisis de los requerimientos del sistema.

#### **Criterios de Éxito**

Por un lado tenemos el criterio de éxito en cara al cliente, mientras que a su vez también tendremos el criterio de éxito para el profesional.

**Conectar rápidamente a un cliente con un profesional de confianza**

**Garantizar el aumento del volumen de trabajo al profesional**

El cliente tendrá la necesidad de **contactar rápidamente con un profesional de confianza**, para esto se necesitaría que los profesionales estén en el sistema y que ellos tengan una razón para utilizar el sistema, lo que nos da el criterio del profesional que será **aumentar el volumen de trabajo habitual**.

#### **Historias de Usuario**

Con estos criterios intentaremos encontrar las historias de usuario.

**Como cliente necesito contactar un profesional en el momento para reparar un problema en mi hogar**

**Como cliente necesito conocer la experiencia del profesional para decidir a cuál contacto**

**Como profesional necesito cobrar mi trabajo realizado para continuar prestando el servicio**

**Como profesional necesito ampliar mi cartera de clientes para tener más flujo de trabajo**

Con los stakeholders que vienen con la idea encontraremos varias historias de usuarios referidas tanto a clientes como a profesionales. Por ejemplo, un **cliente necesitaría contactar con un profesional** o este **profesional necesitaría poder cobrar por su trabajo** para así continuar brindando servicios. Será con estas ideas que iremos trabajando y definiendo bien claro el problema siempre tratando de evitar entrar en la solución.

**RECUERDA: Es importante que el espacio del problema sea genérico y no específico del software.**



### **Requerimientos**

El siguiente paso es tratar de encontrar los requerimientos técnicos pasando ya al espacio de la solución.

#### **Ciclo de prestación de servicio:**

Solicitar, aceptar y finalizar una prestación de servicio de forma segura

#### **Comunicación:**

Capacidad de búsqueda y comunicación rápida entre clientes y profesionales disponibles

#### **Evaluación:**

Capacidad de evaluar profesionales y clientes para referencia futura

Por ejemplo, para el **ciclo de prestación de servicio** tendremos que poder como clientes solicitar un servicios y como profesional aceptar este servicio, y que en la **evaluación** ambas artes acuerden, una vez sea finalizado este servicio, que hubiera sido efectuado correctamente o no.

En la **comunicación**, el cliente que solicita un servicio tendrá que poder comunicarse rápidamente con el profesional para coordinar una visita a su hogar.



### Riesgos

Una vez visto los requerimientos, también tendremos que encontrar los riesgos relacionados a las historias de usuario.

**Un cliente utiliza un servicio y no completa el pago en un tiempo determinado**

Un profesional llegó a la puerta de mi casa y no puedo confirmar que sea quien dice que es

**El proyecto no está terminado para la feria de Profesionales Independientes de agosto 2018**

Uno de los riesgos será que el **cliente utilice el servicio y después no pueda pagar por él**, de esta forma el profesional no podrá seguir brindando su servicio y no continuaría utilizando la aplicación. También tendremos el riesgo de que un **profesional llegue a la casa de un cliente y no pueda ser identificado**, que el cliente no sepa asociar a la persona física que se encuentra en su puerta con el usuario de nuestro sistema. Son estos tipos de riesgos al que daremos más contexto sobre la solución que tendremos que implementar.

### Restricciones

A medida que hablemos con nuestros stakeholders iremos encontrando algunas restricciones.

Recursos disponibles para el desarrollo

Registro de impuestos del profesional

Garantía de profesionales sin antecedentes penales

Por ejemplo, **tendremos que tener en cuenta los recursos disponibles**; si tendremos un equipo de desarrolladores. También necesitaremos trabajar, obviamente, dentro de la ley, es decir, tener un **registro de impuesto** con el que el profesional pueda trabajar integrado dentro del sistema y así garantizar que estemos correctos fiscalmente. Por último, debemos **garantizar que los profesionales no serán un problema** a la hora de ir a trabajar en la casa de un cliente, entonces deberíamos integrarnos a una base de datos policial o verificar antecedentes de los profesionales registrados.

### Proyecto

Todo esto nos llevará a pensar en cómo solucionar el problema. En primer momento, al ser nosotros un startup, tendríamos que tener un contexto muy particular del inicio del proyecto, junto a ello entenderemos que estilo arquitectónico podemos utilizar.

Al tratar de conectar los requerimientos y las restricciones más que nada nuestras restricciones en cuanto equipo e inversión que podemos llegar a tener, se encontró que la forma más simple de trabajar será mediante una arquitectura Cliente-Servidor la cual será montada a la web en donde podremos aprovechar toda la infraestructura de la internet y desarrollar una aplicación pequeña que pueda ser utilizada rápidamente por nuestros stakeholders.

## **DESARROLLO DEL PROYECTO: PLATZISERVICIOS FASE PRODUCTO EN CRECIMIENTO**

**Nuestro sistema está creciendo,** con eso llegan nuevos requerimientos, riesgos, stakeholders y una visión más amplia de lo que podemos solucionar.

### **Primero: Reevaluamos nuestros criterios de éxito**

- Brindar a las empresas cliente estabilidad y control de costos de las prestaciones de servicios que necesiten
- Brindar a las empresas prestadoras una visión de crecimiento de sus servicios.

### **Luego, las historias de Usuario que salen de esta nueva visión:**

- Como empresa cliente, necesito reportes de gastos en servicios para controlar y entender mis finanzas.
- Como empresa cliente, necesito generar listas de profesionales preferidos para nunca perder la disponibilidad del servicio
- Como empresa prestadora necesito medir el rendimiento de mis profesionales para comprender mi propio crecimiento.
- Como empresa prestadora necesito posicionarme como la mejor empresa del mercado para obtener más clientes.

### **Requerimientos:**

#### Reportes

- De gastos por período y por tipo de servicio contratado
- De ingresos y horas trabajadas por profesional por periodo y tipo de servicio prestado

#### Autorización

- Gestión de Usuarios, roles y permisos asociados a acciones del sistema.

#### Posicionamiento y comunicación

- Ranking de prestadores por evaluación
- Lista priorizada de prestadores por tipo de prestación

### **Riesgos:**

- Las empresas cliente no pueden extraer la información del sistema para integrar a sus aplicaciones existentes (normalmente ya existe un ecosistema de aplicaciones)
- Los indicadores de la empresa prestadora no son indicativos del trabajo realizado
- El proyecto podría recibir juicios de fraude por cobros injustificados.

**Restricciones:**

- Conformar estándares de auditoria profesional
- Garantizar la privacidad de los datos de consumo

\*\*Estilo arquitectónico: \*\*

El requisito más fuerte arquitectónico que debemos tener en cuenta pasa por los reportes.

Ahora nuestra base de datos se separa. Por un lado, dejamos lo transaccional en una base de datos y en otra, la que se utilizará para los reportes, a fin de evitar el costo de la lectura de los reportes sobre una misma base de datos (y poner en el peligro toda la estructura de servicios que tenemos en este momento)

**Notas adicionales:**

## Historias de usuario

Como **empresa cliente** necesito **reportes de gastos en servicios** para controlar y entender mis finanzas

Como **empresa cliente** necesito **generar listas de profesionales preferidos** para nunca perder la disponibilidad del servicio

Como **empresa prestadora** necesito **medir el rendimiento de mis profesionales** para comprender mi propio crecimiento

Como **empresa prestadora** necesito **posicionarme como la mejor empresa del mercado** para obtener más clientes

## **DESARROLLO DEL PROYECTO: PLATZISERVICIOS FASE ESCALA GLOBAL**

Ahora somos una empresa a gran escala y por ende vamos a necesitar de nuevos requerimientos.

### **Análisis de requerimientos**

#### **Criterios de éxito:**

- Conectar a empresas locales y globales con los mejores prestadores de servicios
- Facilitar el crecimiento y la globalización de las empresas prestadoras.

#### **Historias de usuario:**

- Como cliente necesito entender el sistema en mi idioma para poder garantizar el buen uso del mismo
- Como cliente necesito acceder a servicios locales y globales para estandarizar los prestadores en mis diferentes localidades.
- Como usuario necesito acceder a los servicios en cualquier momento para no tener problemas dependientes del uso horario
- Como empresa prestadora necesito brindar mis servicios de forma global para ampliar mi alcance al mercado internacional.

#### **Requerimientos:**

- Internacionalización:
  - Traducciones de contenido
  - Registro de prestadores globales y su capacidad de búsqueda local o global.
- Disponibilidad de datos:
  - Cálculo de reportes en tiempo real

#### **Riesgos:**

- El crecimiento de la compañía hace difícil la transmisión de conocimiento y la productividad de nuestros equipos de desarrollo.
- Pérdida parcial o total de datos por fallas no previstas
- Un mercado específico no es accesible por diferencias de idioma.

#### **Restricciones:**

- Evitar procesos acoplados a un huso horario específico
- Empresas que no permiten que sus datos salgan del país de origen.

## Notas adicionales:

### Historias de usuario

Como cliente necesito entender el sistema en mi idioma para poder garantizar el buen uso del mismo

Como cliente necesito acceder a servicios locales y globales para estandarizar los prestadores en mis diferentes localidades

Como usuario necesito acceder a los servicios en cualquier momento para no tener problemas dependientes del huso horario

Como empresa prestadora necesito brindar mis servicios de forma global para ampliar mi alcance al mercado internacional

### Estilo arquitectónico

