

Arquitectura del software

Jueves 6PM - 10PM
Virtual

- Atributos de calidad y técnicas
 - ↳ Performance, estabilidad, disponibilidad, confiabilidad, visibilidad, modificabilidad, portabilidad, seguridad, interoperabilidad, testeabilidad, usabilidad, etc.
 - ↳ Técnicas
- Estilos de arquitectura
 - ↳ Replicated repository, cache, client server, remote session, stateless, layers, code on Demand, remote evaluation, uniform pipes, & filters, events based integration, etc.
 - ↳ Rest
 - ↳ Web
- Cloud computer architecture
- Patterns of Enterprise Application Architecture
- SDGT y SOA
- No SQL

PP 1.

¿Qué es arquitectura del software?

- ↳ La descomposición de los alto nivel de un sistema en sus partes fundamentales
- ↳ Conjunto de decisiones de diseño que son difíciles de cambiar

Conceptos o propiedades fundamentales de un sistema para su entorno, comprendido por sus elementos, relaciones, principios, de su diseño y evolución.

(... definición Roy Fielding)

(... definición Robert Martin - Uncle Bob)

CONJUNTO DE ESTRUCTURAS NECESARIAS para razonar sobre un sistema, que constan de elementos de software

relaciones entre ellos.

y propiedades de ambos

Estructura: Conjunto de elementos unidos por una relación

Elementos: módulos (estáticos) componentes (dinámicos)

Tipos de estructuras

- Module: Unidades estáticas de implementación a las que se les asigna responsabilidades. Razones sobre modificabilidad?
- Component & connector: Dinámicos. Elementos en runtime. Performance, disponibilidad, seguridad?
- Allocation: Mapa entre software y su entorno.

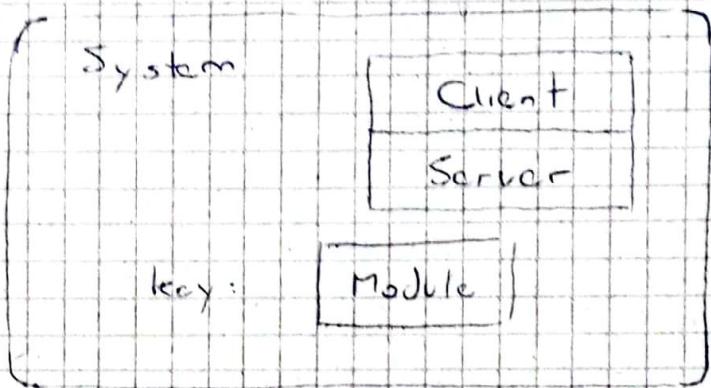
Vistas de la arquitectura

• Abstracción

→ Representación visual de una estructura

Documentamos estructuras, y se documentan las vistas de esas estructuras.

Ejemplo:



Arquitectura de software de un sistema

Arquitectura y
componentes

→ Estructuras

Elementos arquitectónicos

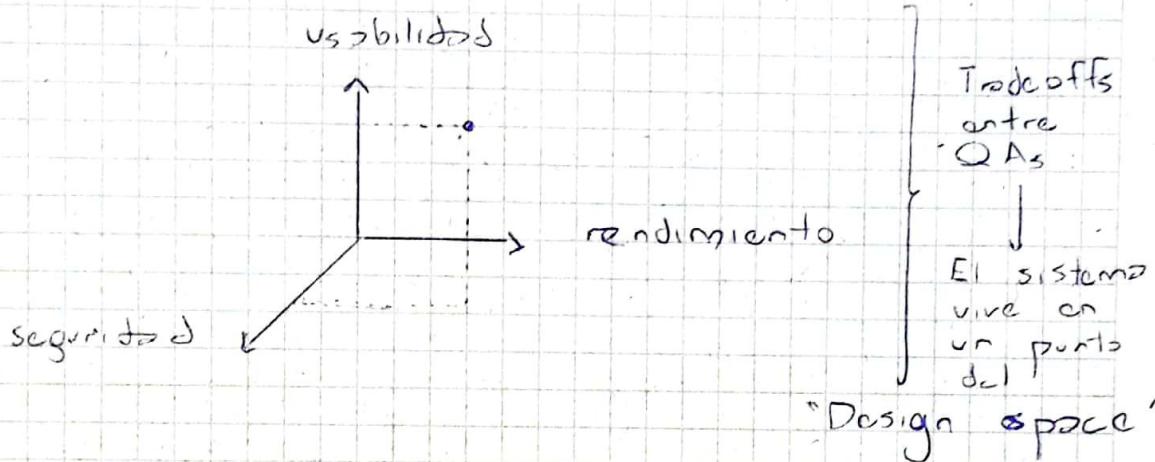
Vistas arquitecturales

Obligatorio
de negocio → Requerimiento
del sistema

"Realizar el 100% de los
trámites online"

X Atributos de calidad - QA → Quality attributes

Propiedades medibles del sistema.
Permiten evaluar en qué medida se satisfacen las necesidades de los stakeholders.



X Tareas → Decisiones de diseño, restricciones

↳ Las tareas promueven o inhiben atributos de calidad.

Allocation of responsibilities, coordination model, data model, management of resources, mapping among architectural elements, binding time decisions, choice of technology.

Objetivos de negocio ≠ Requerimientos

Arquitectura vs. diseño

↳ Afecta a más de un ACC
(Atributo de Calidad Clave).

Arquitectura es diseño, pero no todo diseño es arquitectura.

Q5 Who needs an architect

Software architecture → Consensus between expert lead developers in a project.

↳ about that systems design.

How the system is divided in components + how are interconnected through interfaces.

Architecture excludes subcomponents of the main components. It only considers main components of the system.

↳ "Understood by all developers"

"Architecture is about the important stuff, whatever that is".

Architectus
Relaudus

vs

Architectus / Guide
Oryzis

Just designs,
the layout
of the system.

Makes important
decisions.

(may be a
decision
bottleneck).

Is aware of
everything in the
project, ~~but~~ tries
to mostly catch
problems early.

Talks with
everyone.

Mentors the development
team. Teaches them
to make decisions
(should make few
themselves).

Architecture — "Decisions hard to change"

You could design a system that allows changes, making those decisions non-architectural.

↳ You should eliminate irreversible decisions. ☺

[Irreversibility is a prime driver for complexity.]

⊗ but making everything changeable makes the whole system more complex.

⊗ Change in software architecture is much easier in that in building architecture.

↳ We are only limited by imagination, design, and organization.

↳ People tend to limit change.

23/03/23

Atributos de calidad → Definan cualidades de los requerimientos X

Requerimientos del sistema Funcionales → Qui debe poder hacer el sistema.

Restricciones → Decisiones de diseño ya tomadas, imposibles de cambiar

Mensurable → Los atributos deben poder medirse.

Stakeholder → "Participante" "parte interesada"

Mejorar un atributo implica empeorar otro. ↗ Design Space

↳ Imposible hacer algo más seguro sin robustecerlo.

Hablar de negocios y dinero para facilitar la comprensión

↳ "Si se cae el sistema perdemos un millón por minuto" → "Necesitamos programar los datos técnicos para evitar problemas".

Non-operational

"funcional"

Availability → Capacidad de arreglar el sistema lo más rápido posible. Queremos más availability.

Performance

Solvability

↓
Qui es un **Fallo**. → El sistema dejó de hacer lo que necesitamos

No es binario (sí o no), también que si funciona lento o mal es un **fallo**

↓
Depende si alguien lo percibe.

Que es una
Fall = \rightarrow Un error que se
logró detectar

Fallos < Fallas

Si no lo
tomas so vuolve
una falla.

Si el usuario
no sigue
que algo esté
mal, es una
falla.

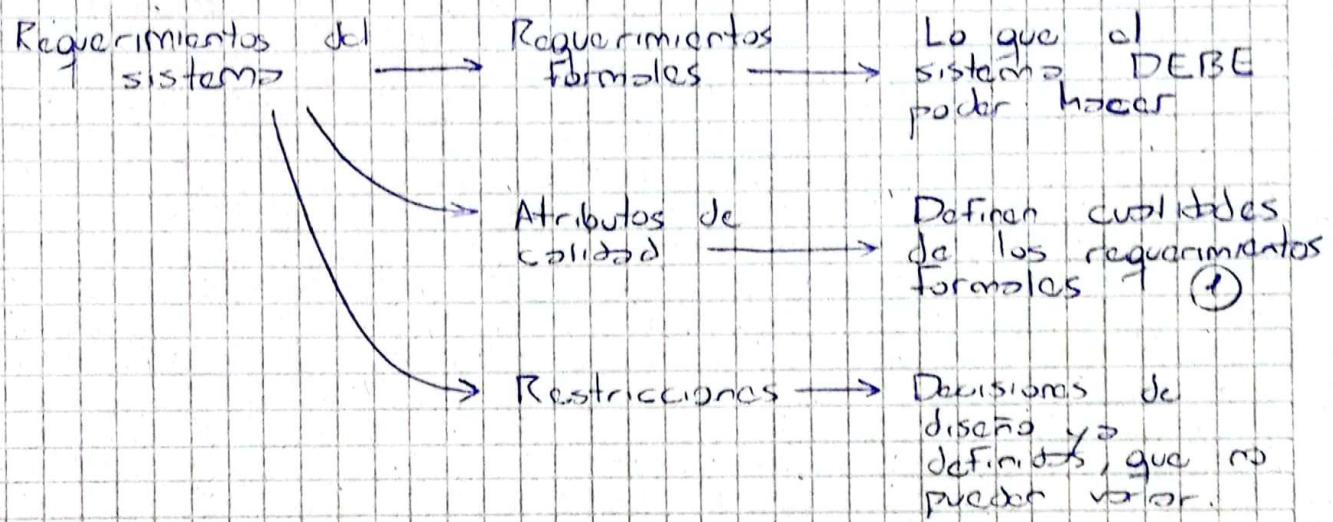
Performance \rightarrow Capacidad de l sistema a
reaccionar ante un evento
en cierto tiempo.

Eventos Periodicos

Estos

Clase 2

Atributos de calidad



(1) Calidad measurable.
Mejorar una calidad tiende
a empeorar otra.

Operativo

Interno & Externo (S. lo
que los stakeholders
o no)

No - operativo Modificabilidad Usabilidad

↓
(si es
funcionalidad
o no)

(Availability)

- Falla → Cuando el sistema (Failure) ~~falla~~ de brindar lo esperado. → y el usuario lo nota.
 - Falta → Fallo que no es corregida

) AVAILABILITY

1) Estabilidad del sistema
para reparar. Ellas antes
de un lapso prefijo 200.

- | | | |
|------------------|-------------------------|---|
| Tipos de fallos: | → Omission : | El sistema no responde
⇒ un pedido. |
| | → Crash : | El sistema omite en
repetidas ocasiones. |
| | → Trimming
Threading | El sistema responde,
pero fuera de tiempo. |
| | → Response Failure : | El sistema da una
respuesta incorrecta. |

Possible scenarios
Frontal > uniaxial
fallas.

- Formar un log de Ella
 - Notificar → (ciertos) usuarios
 - Pesar → modo de degradación (menos operaciones y funciones)
 - Posar → modo de no disponibilidad (hasta que sea reparada Ella).

"Disponibilidade"

→ Probabilidad que el sistema este operativo cuando se lo precise.

99 % 99,9 % ... 99,9999 % ...

↳ Parte del Service level agreement.

Performance

- 1)
- 2) } →
- 3)

Habilidad del sistema para reaccionar ante ciertos eventos en un determinado tiempo.

- ↳ Interrupciones
- Mensajes
- Pedidos de usuario
- Pedidos de otro sistema
- Paso del tiempo
- Periódicos
- Estocásticos (hasta cierto número por unidad de tiempo)
- Esporádicos

la interacción entre elementos define

- ↓
- ↳ Velocidad y eficiencia de la red
- ↳ Performance que recibe el usuario

Cómo es la interacción → costos determinados por la arquitectura

Qué determina la performance

- Qué requiere el sistema
(ej: jobs en A usados en B)
- Número de pasos hasta el objetivo
- Implementación de la arquitectura
- Implementación de cada componente

Cuando mejorar la performance

- Cambiar y/o costos andando la aplicación
- Temporaneamente con cambios de diseño de arquitectura

1) Network performance:

- Throughput → Velocidad de transmisión (cuanto más ~~costo~~ tiempo)
- Capacity → Máxima carga soportada
- Bandwidth → Máximo throughput disponible en un canal
- Usable bandwidth → Ancho de banda disponible para cada aplicación.

Cuanto menos se depende de la red, más rápido podría ser la aplicación.

2) User perceived performance:

Completion time → Desde que se hace una request.

Latency → Tiempo mínimo ^{para} ~~desde~~ esperar a ejecutar una request

Responsiveness → Tiempo hasta aceptar una request

3) ~~Efficiency~~ Efficiency → Performance por recurso consumido

|
└ Load (carga) → Stress según exigencia

Load sensitivity → Variación de response time en función de la carga.

(Scalability)

Habilidad del sistema para recibir nuevos componentes de manera efectiva.

Mejor medida de calidad

No requiere un gran esfuerzo

No interrumpir el sistema

Escalabilidad horizontal



Sumar más unidades
(de la misma capacidad)

Complejo, pero con más
potencia.

Escalabilidad vertical



Mejorar las capacidades
físicas. Crear en hardware

Fácil pero con potencial límitado.

Elasticidad → Capacidad del sistema adaptarse
según las ~~recomendaciones~~
~~del sistema~~, demandas

X Técnicas de disponibilidad → Detección → Ping / Echo → Una parte manda un ping y la otra responde un echo.

Heartbeat → Un componente manda latidos periódicos

Watchdog → Si algo no cambia, actúa.

Votación → El votante compara los resultados de todos los procesadores (que recibieron el mismo dato).

Costoso

Evita errores en un punto crítico

Da redundancia.

X Técnicas de recuperación

Redundancia

SPOF

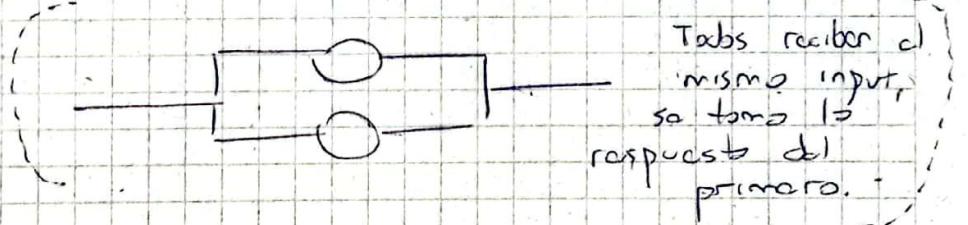
Single Point of Failure

- de información
- de tiempo (tiempo extra de procesamiento)
- físico (hardware y software extra para resistir algún mal funcionamiento)

Redundancia → Componentes redundantes

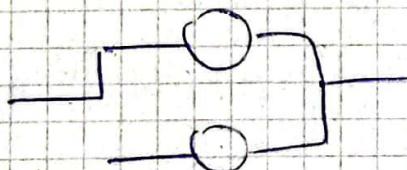
activa

→ Si hay un fail, permite que el downtime sea muy corto



Redundancia

→ Existe redundancia, pero solo se utiliza cuando se desconecta el todo principal.



Spare → Una plataforma extra lista para activarse si falla otra.

• Requiere que se le imprese un checkpoint para seguir con la configuración anterior.

X Técnicas de prevención

→ Removal from service → Quitar preventivo y periódicamente para reparar o reubicar

→ Transactions

→ Process monitor → Cuando se detecta un error, el monitor reemplaza, detiene o reubica el componente

X Técnicas de performance

(Objetivo: Reducir el tiempo de respuesta)

Demandas de recursos

↳ Reducir el nº de eventos y su consumo.

[Razones de bloques]

- Disputa de recursos (esperar a que otro componente libere los recursos).
- Disponibilidad de recursos
- Dependencia de resultados (esperar a que otro proceso da su resultado).

Gestión de recursos

↳ Aprovechar la concurrencia.
↳ Tener más recursos

Arbitraje de recursos

↳ Cómo priorizar los recursos.

~~20/04/23~~

~~Estilos arquitectónicos~~

Clase 3.

Recordando:

Arquitectura del software de un sistema es

Conjunto de estructuras neckline para

razonar sobre el sistema

Las estructuras constan de elementos de software,

relaciones entre ellos → y sus propiedades

Atributo de calidad

Propiedad mensurable para saber si el sistema satisface una necesidad

Tácticas
Arquitectura

Decisión de diseño que afecta a un atributo de calidad

Objetivos de negocio

≠

Requerimientos

Objetivos para la empresa en la realidad

"Ganar más parte del mercado"

Necesidades sobre ciertos atributos de calidad (modificabilidad, usabilidad, performance).

Atributos de calidad

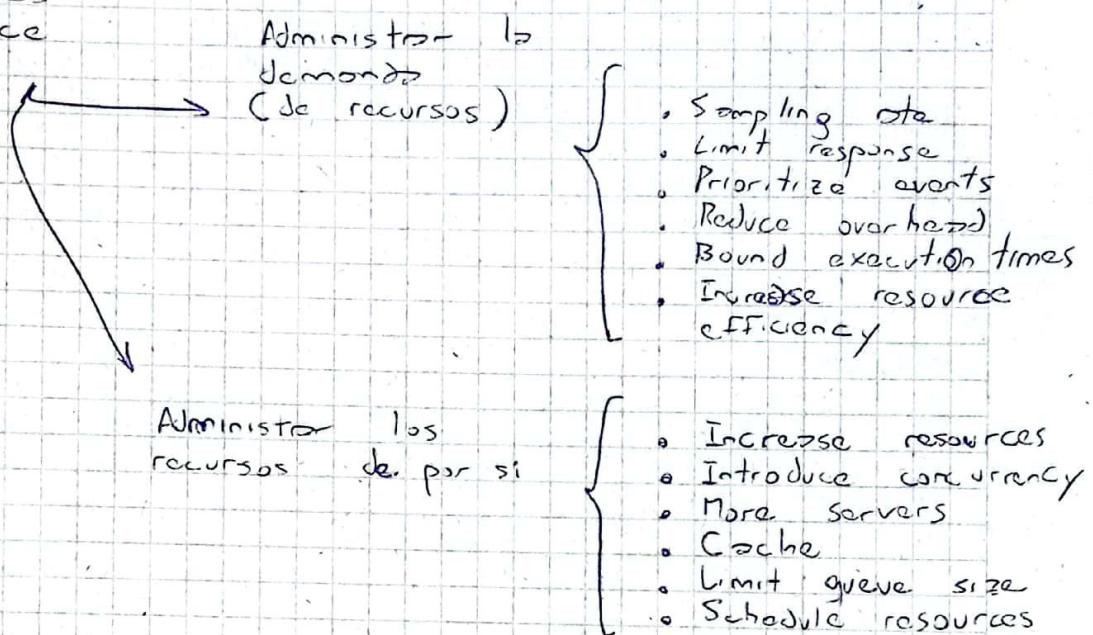
Durante el runtime

Performance, scalability, elasticity, reliability, availability, visibility, interoperability, security, usability

Durante el desarrollo

Modifiability, portability, simplicity, testability

Algunas tácticas de performance



Tácticas de scalability

→ Stateless (store state in client)

→ Evitar presuposiciones de hardware

→ Pro Monitorear

→ Diseño con elementos sincrónicos

→ Sharding

→ Design for multiple live sites

?

?

Tácticas de availability

(prevenir y corregir fallos rápidamente)

→ Detector fallos → Ping, monitoring, sanity check, voting ...

→ Recuperarse de fallos → Redundancy, retry, rollback ...

→ Prevenir fallos → Remove from service, exception prevention ...

Ver todos los tácticos en

P.P. de Close 3.

Close 4. 20/04/23

~~Alojamiento~~ ~~funciones~~ ~~mas~~ ATRIBUTOS DE CALIDAD

● Modifiability → Extensibility

Costo de realizar cambios en el sistema

→ Configurability

→ Evolvability

→ Customizability

→ Reusability

● Security

Confidentiality

Integrity

Availability

Authentication

Authorization

Non-repudiation

Threat modeling → Armar árboles de ataque.
Como podrían atacarnos y
que requiere.

• Testability

Facilidad para
testear y
detectar
problemas

• Usability

Facilidad que tiene
el usuario para
usar el sistema

• Interoperability

Intercomunicación
entre sistemas
a través de interfaces.

Sintáctico → Intercomunicar
datos

Semántico → Interpretar
datos

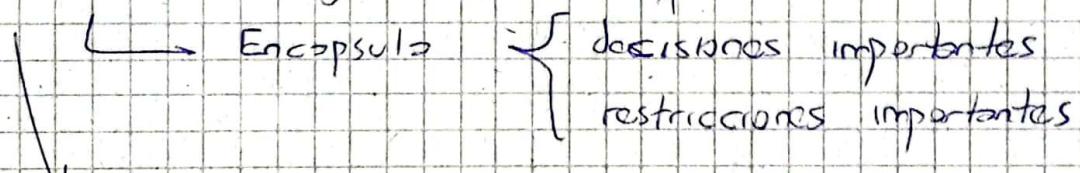
Ej: 21, 22

Close 5. 27/04/23

Estilos arquitectónicos 1

- ↳ Conjunto coordinado de restricciones arquitectónicas
- ↳ Limitan las características de los elementos arquitectónicos y sus relaciones
- ↳ dentro de cualquier arquitectura que se ajuste a ese estilo.

Estilo → Categorización de arquitecturas



↓
Es importante elegir el
estilo correcto en
cada caso

→ No existen
↳ Silver
Bullets

Categorizaciones de estilos según Fielding:

- Data Flow styles
- Replication styles
- Hierarchical styles
- Mobile code styles
- Peer to peer styles.

Replicated repositories → Más de un proceso provee el mismo servicio, son copias de entre ellos.

Cache → Preparar un dato o prepararlo, para que ya esté listo cuando sea solicitado.

Client server → Dos protagonistas, servidores y usuarios. → Uno pide cosas → El otro provee, según que le pidan.

Layered system → Sistemas en capas, partes divididas. Los capas inferiores brindan servicios a las superiores.

[Dibujar polígonos
Dibujar vértices
Contener puntos]

Layered client-server → Igual que CS, pero el servidor es sólo un capa.

Client → App server ← Data storage server/source

Load balancer → Conociendo varios servidores, maneja qué carga se envía a cada uno.

Es la cara de los servidores, esconde sus IPs. El reverse proxy es la única cara visible a los servidores.

(Si un reverse proxy ~~hace~~ impide que los servidores vean internet, un forward proxy impide que los clientes vean = interfect).

Client stateless
server.

Igual que un CS, pero el servidor no retiene el contexto del cliente.



Cada request del cliente contiene TODA su info de estado, y es que el servidor no tiene nada.

Client cache
stateless server

CS con cache.

Otro Layered client → CCS con proxys
cache stateless server y load balancers...

Remote session → La sesión la mantiene el servidor

Remote data access → El servidor crea un espacio de trabajo desde donde el cliente pide cosas

Mobile code
styles

Virtual machine

Ejecuta código en un nuevo ambiente controlado.

Remote evolution

El cliente tiene la info y los pasos para solucionar una request, pero NO tiene los recursos

Code - on - demand

Igual que remote evolution, pero el cliente NO tiene el código temporal. Este lo provee el servidor.

Layered Code-on-demand
Client Cache Stateless Server

Mobile Agent → Mover un componente externo a un sitio remoto. Para estar más cerca del próximo destino, por ejemplo.

Estilos arquitectónicos 3.

• Peer to peer styles:

• Event based integration: Un elemento en mi sistema implementa ~~este~~ ~~estos~~ eventos → la publicación de ciertos eventos. (no importa mucho quién publica). Otros componentes simplemente publican.

↳ Estos eventos se publican en un bus de eventos.

• Distributed objects: Los objetos se comunican entre ellos consumiendo los servicios que ofrece el otro. Una operación invoca a otra. Es → un tercero... (una orden de invocaciones es una secuencia).

Todos los objetos se conocen entre ellos. El estado del conjunto depende del estado de cada uno de los partes.

• Brokered distributed objects: BO + Layered Client-Server
Hay un mediador en la comunicación, yo me registro ~~me pongo~~ con el broker para tener un servicio. Luego otro le pide al broker quien da un servicio. Ahí el último le habla → mi componente.

(como un broker real, busca alguien que compre y alguien que venda).

• Data centered architectures

El almacenamiento de datos es central

→ Y todos los componentes giran alrededor de eso.

Blackboard: El estado actual es quien dirige quién va a hacer.

Base de datos: Los nodos tienen la iniciativa, y solo le mandan info → la base de datos.

• Data Flow styles

× Pipe & Filter → fluxo em um sentido

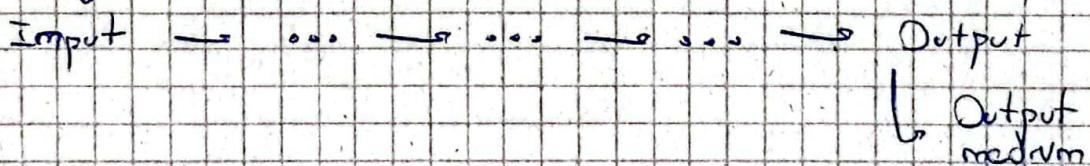
Los datos se consumen → processar "Procesamiento incremental"
antes de que lleguen completamente.

Pipe: Un tipo de cosa que sale de una parte va a la otra

Los filtros
son
independientes

Byz instances
"Primer indicio de respuesta"
(f de cuando termina).

Input
medium



✗ Uniform Pipe & Filter → "Todos los filtros
deben tener la
misma interfaz (formato)".

Facilita usar los filtros de otros módulos,
modificar cuando están.

(empieza la performance
porque tardo que
cambiar de
formats).

✗ Batch sequencial

Requiere que el proceso
anterior finalice antes
de enviarle la información.

Son una serie de pasos, pero no es tanto
un pipe (porque no me envía
"mientras tanto").

11/05. Close 6.

Web architecture & Rest.

⇒ Un estilo de arquitectura en particular → Rest

Representational
State
Transfer

Internet

→ Redes interconectadas
se comunican
con protocolos
↓
Infraestructura

Web → Una enorme
red, global

↓
"Construido sobre
Internet"
Documentos hipertexto.

Web → Objetivo:
↳ Un espacio donde compartir información,
donde las personas y máquinas se
comuniquen.

Requerimientos:
→ Almacenamiento de información
↳ Referenciar info que tiene otra
persona.
↳ Hardware muy variado → lo largo
de todo el planeta.

Atributos de
calidad:
→ Baja barrera de entrada
→ Extensibilidad
→ Disponibilidad
→ Escalabilidad horizontal
→ Deployment individual - independiente.

Estilo de arquitectura
de la web?

→ Client-server

Subclases → (el server no
retiene el
contexto
del cliente).

+ Uniform interface + Layers + Code-on demand
(ejemplo:
Optativo) → jacket

A este conjunto de técnicas
formamos el estilo arquitectónico

REST.

Representación → State Transfer.

Interface
uniforme, 5
técnicas

1. Identificación de recursos. Cada uno tiene una ID.
2. Métodos de acceso tienen sentido definido.
3. Los recursos se representan.
4. Los mensajes son auto-descriptivos
(conexión stateless).

5. Hypermedia As The Engine Of Application State. (HATEOAS) →

Aplicación

Application state

→ Dónde se están (o en home, o en inscripciones, o en contactos...)

En el cliente.

Resource state

→ Lo que se almacena en el servidor.
(ejemplo: una inscripción o la una materia).

Hypermedia → Al ver uno cierto tipo HTML,
puedo que puedo hacer cierto accion.
Ejemplo: Puedo ir a cierto lugar o
puedo hacer cierto post.

La información viene con más info. que / La presencia
nos explica como acceder o como seguir. / sobre como puedo
seguir o
que puedo hacer con la
info que tengo.

⑧ HATEOAS

→ El cliente tiene el

client state ↓
Se modifica con requests y responses

[Hypermedia es el medio para modificar el
cliente de la aplicación.]

! Rest API → Roy Fielding: "Si el engine de application
state no tiene hipermedio, no es Rest".

Debiera poder accederse solo sabiendo su dirección
y un set de standard media types

7

Popover

1/06/23 Clase I.

Gaso Dulzón S.A.

Software que gestiona tu actividad comercial con todos tus clientes

CRM → Customer Relationship Management

BI → Business Intelligence → Información para tomar decisiones.

SAP → Empresa alemana de software para gestión empresarial.

CIO → Chief Information Officer → Investigación de nuevas tecnologías y su posible implementación

CTO → Chief Technology Officer → Lider en gestión de sistemas instalados.

FALTA DE INTEROPERABILIDAD

↳ Solución: Arquitectura basada en servicios montados en un bus común.

Problemas: En qué fallamos? → Si se pone todo que dejar de operar por una causa de varios sistemas críticos

Fitx redundancia.

Se depende de un único sistema → Bus común do datos } Es necesario tener TODO en un mismo bus?

• B Descripción de nuestro sistema:

"Bus común" → Estilo arquitectónico tipo "data flow."

El bus común es una pipe, aparentemente uniforme, ya que

"se reducen drásticamente las cantidad de interfaces"

Arquitectura basada en servicios → Cada parte de la empresa se procesa independientemente, en layers. Pero si la anterior falla, afecta a la siguiente. (2)

• Cómo llegamos a 2007? → No falló el CTO?

Al implementarse sistemas sobre sistemas, nació ideas ni diseño el software de la empresa de forma integral.

Nadie estudió cómo convivirían los sistemas.

Más allá de este problema en 2010, la solución de arquitectura de software solucionó los problemas de 2007.

Fue quien separó las partes de la empresa. (1)

Si ~~se~~ solucionamos el problema de bus 'único' (superando que es al mismo punto de falla) Dulces S.A. quedaría muy bien puesta con un excelente sistema interoperable y escalable.

• Fue un error seguir los consejos?

No, solucionó los problemas de interfaces. (1).

• ¿Qué deberíamos hacer ahora? → Identificar el punto de fail (el vez más) y solucionarlo, hacerlo muy controlable. (con redundancia por ejemplo).

→ Cuando lo solucionamos, todo el sistema será controlable, interoperable y escalable.

①

② Implementar redundancias para que, si falla mi copia anterior, yo pueda seguir trabajando.

L

Estrategia empresarial de Porter.

3 estrategias para tener éxito.



Diferenciación



Costos



Nicho

Homologar ISO.

1/06/23

Sistemas distribuidos de Open

Término

- Aquellos que:
 - son muy grandes y no pueden implementarse desde cero
 - Algunas complejas
 - Sistemas legacy (viejos y sin mantenimiento, pero funcionales).
- Sus partes tienen distintos estados
 - La necesidad de funcionalidad crece, pero el sistema no.
 - Dudas técnicas
 - Diferentes owners
 - Cuellos de botella

Esfuerzos de IT → Implementar costos nuevos o
→ Solucionar dudas técnicas

08/06

~~Cloud computing~~

08/06 (A)

Cloud computing

" Acceso ubicuo, conveniente y bajo demanda
• es un conjunto compartido de recursos computacionales configurables a través de la red, rápidamente provistos y desplegados con un mínimo esfuerzo de administración interacción con el proveedor del servicio".

- Acceso ubicuo (en todos partes, en todo momento)

- Bajo demanda
- A través de la red
- Rápidamente provistos
- Mínimo esfuerzo de administración

SaaS (software as a service)

PaaS (platform as a service)

IaaS (Infrastructure as a Service)

{ On demand self service
Broad network access
Resource pooling
Measured service.
Rapid elasticity.

Características de Cloud Computing

- On-demand usage
- Ubiquitous Access
- Multitenancy (una misma instancia de software para varios consumidores) (usuarios distintos).
- Elasticity
- Measured usage
- Resiliency

↑ Cuando agregar capacidad → Lead strategy (anticiparse a la demanda)

Log strategy (seguir a la demanda).

Over provisioning vs Under provisioning } (prover cuando se esté al máximo).

↓ Match strategy (Seguir directamente la demanda)

Reducción de (o al menos en) costos d corto plazo → Se evita una gran inversión inicial
 • " " muchos empleados de mantenimiento.
 • Electricidad
 • Licencias
 • Responder a fluctuaciones de demanda

(Más complejidad)
 para el uso.

Infrastructure
 as = Service

(Más abstracción)

Platform as = Service

Software as = Service.

No. Gang y/o uso G.t.
 No se usan funciones, pero puede consumirlas.

Trade-offs

Menor inversión inicial.

Costos proporcionales

Escalabilidad

Escalabilidad

Disponibilidad.

Security.

Falta de control

Baja portabilidad

Cuestiones legales (security).

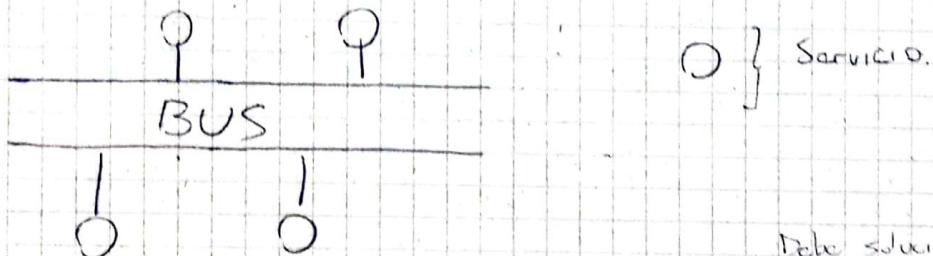
Service Oriented Architecture

08/06

SOA aparece para solucionar los problemas de sistemas distribuidos de gran tamaño.

• SOA es un paradigma para el mantenimiento de grandes procesos de negocio en grandes sistemas distribuidos.

• No es tangible, no es un software en particular



[Servicio] → Cumple una funcionalidad de negocio.

• Esto autocontenido.

Debe solucionar un problema en particular del negocio.

• Su interfaz expone valor, acortando IT y Business.

Debe ser claro entender su valor.

• Puede estar compuesto por varios procesos.

Con un SLA (Service Level Agreement) para pactar las condiciones del servicio.

(no informáticos)

Alguno de negocios podría proponer un nuevo servicio
pasando servicios menores.

[SLA es para un SERVICIO.]

[Enterprise Service Bus (ESB)]

- Ofrece y consume servicios - Lista todos los servicios disponibles
- Tolerar todas las heterogeneidades de los servicios, y no se mete en el funcionamiento interno de los servicios.

[Se deben poner reglas para la convivencia entre service owners. SOA no define qué reglas, solo los pide.]

Gestión

La união de servicios, bus y políticas debe ser gestionado.

No equipos que define las reglas, para valo por la descentralización. Este equipo requiere Seniors.

Convencer → los service owners de conectarse al bus requiere apoyo político (del CEO).

(No tiene sentido agregar todo esto complicación)
y burocracia para sistemas chicos.

El bus y SOA no son perfectos.

• Hay sacrificios en performance y seguridad.

• Según SOA, los servicios son cobros negros, solo m^a importan sus interfaces.

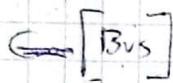
• Se prefieren interfaces BUSINESS DRIVEN (no technically driven)

• Servicios autocontenidos (encapsulados)

• Visible (sobre que arista)

• Stateless

• Reusable e interoperable (que el servicio pueda ser llamado desde otros sistemas).



Conexión indirecta (para lidar si una punta está caida).

Bus mede entre partes, gestiona la conexión.
Los servicios NO tienen conocimiento de los demás.

Beneficios → Cada servicio puede modificarse fácilmente.
→ Multiples proveedores del mismo servicio.

El bus puede ser un name server (DNS) o un load balancer (proxy).

Los ESB se clasifican según Protocol Driven
y API Driven.

Valor agregado del ESB (Enterprise Service Bus) → Qué hace además de mediar entre servicios.

- Data mapping
- Rutas inteligentes. El WS Rutas, en load balancer, token filos. Prioriza mensajes según su contenido.
- Seguridad. Límite acceso.
- Gestión de fiabilidad. Maneja los errores de ciertos protocolos o servicios.
- Monitoring & Logging.
- Facilita actividades de negocio. Informes al estado del negocio.
Permite encontrar oportunidades de negocio.

XML → formato de archivos para intercambio de información entre dos plataformas distintas.

SOAP → Protocolo que define cómo los procesos / objetos se comunican (utilizando XML).

Bases de datos / No SQL

- 1) BD relacionales
- 2) BD orientadas a objetos
- 3) BD NoSQL

SQL,
Structured
Query
Language.

Query: Consulta

Lenguaje de
programación para
bases de datos
relacionales.

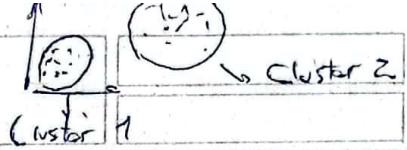
1) Bases de datos relacionales

- ↳ Bon contenido de información
- ↳ Se pueden manejar fragmentos de información
- SQL → lenguaje estandar.
- Conurrencia. Acceso simultáneo.
- Integración. Multiples apps pueden acceder a los datos

— Frustraciones:

- ↳ • Formato de tablas y valores (como tuplas)
- ↳ • Impedimenta mismatch: Diferencia entre el funcionamiento de los computadores y la base de datos.
} El modelo en memoria VS
} El modelo relacional.
- ↳ • Shared DB integration: Que la información tenga varios dueños, y que varias apps puedan acceder compliciza la base de datos.
- ↳ • Difícil de escalar (en los casos de los enormes).
- ↳ • A las BD relacionales les cuesta trabajar en clusters. La información que antes estaba en una BD ahora debe repartirse a varias.

Clustering → formar grupos.



3) Frente a todos estos fracasos se buscó desarrollar una BD no relacional. (NoSQL).
(No hay un estandar ni una organizacion que lo controle, es solo un concepto básico).

NoSQL → Open source, post 2000, no usan SQL, funcionan con clusters
↓
Se pierden algunas ventajas de ACID. ↓ Transacciones / ACID entre clusters.

Aplicación: → Se usa cuando el volumen de datos es muy grande, y clustering se vuelve atractivo.
→ Fácil de trabajar entre la programación y la base de datos, no sufre de Impedimente Mismatch.

ACID y SQL. } Las bases relacionales cumplen con ACID.

- Atomicity → Si hago varios cambios a la vez, se aplican todos o ninguno.
- Consistency → Todas las transacciones serán válidas.
- Isolation → Se manejan las operaciones concurrentes como si fueran secuenciales.
- Durability → Las operaciones exitosas persistirán incluso aunque se caiga el sistema.

No SQL.

Modelo de datos ≠ Modelo de almacenamiento.

↓
Como percibimos
los datos

↓
Como los
manipulamos

Modelo de datos agregados de NoSQL.

key - value
Document
Column - Family
Graph

Estos 3
son
agregados. → "Orientación
a la
agregación"

Permite → Evitar la estructura limitada
de las BD relacionalles

→ Manipular la info en unidades,
de manera más compleja.

→ Anidar tuplas

→ Listas de valores y tuplas

Clustering → El gran efectivo
de NoSQL
Para SUMA COMPLEJIDAD
Cuidado.

Métodos de distribución
de datos

Ya que hacemos
clustering, la
info. estará
partida en varios
lugares.

Definido

La habilidad para manejar
mayores volúmenes de datos

" " " "
mayores volúmenes de
lectura y escritura

La habilidad para afrontar
enfrentamientos y colas

• Sharding → Colocar diferentes partes en diferentes servers.

Hay que buscar las partes de la info
en distintos lugares

• Master-slave replication

UH! cuando
hay muchos
lecturas
pero pocos
escrituras

Un maestro recibe
las updates de datos
y le avisa a los esclavos

Caso se cuando se necesita
leer información, se puede
leer de tanto el maestro
como los esclavos.

Mayor resistencia
a fallos

Si se da algo se puede
seguir leyendo de
otro lado.

Possible problema: Que
el esclavo tiene
info vieja.

Si se quiere escribir solo
se rompe si es el master.

• Peer-to-Peer Replication.

Similar a Master-Slave, pero sin Master.
En Peer-to-Peer, todos lean y escriben.

Aparecen conflictos write - ~~not~~ write.

- ↳ Coordinar antes de escribir (pérdida de performance)
- ↳ Aceptar posibles inconsistencias.

Al no tener un master, los inconsistencias podrían ser permanentes.

• Sharding + Replication.

Consistencia

BD relaciones → Consistencia fuerte!

BD NoSQL → Consistencias distintas.

- ↳ Teorema CAP
- ↳ Consistencia Eventual.

Update consistency

Conditional Update → Checkear el estado antes de escribir (por si hubo cambios en los últimos segundos).

Write-lock → Bloquear el acceso a la información mientras se está escribiendo.

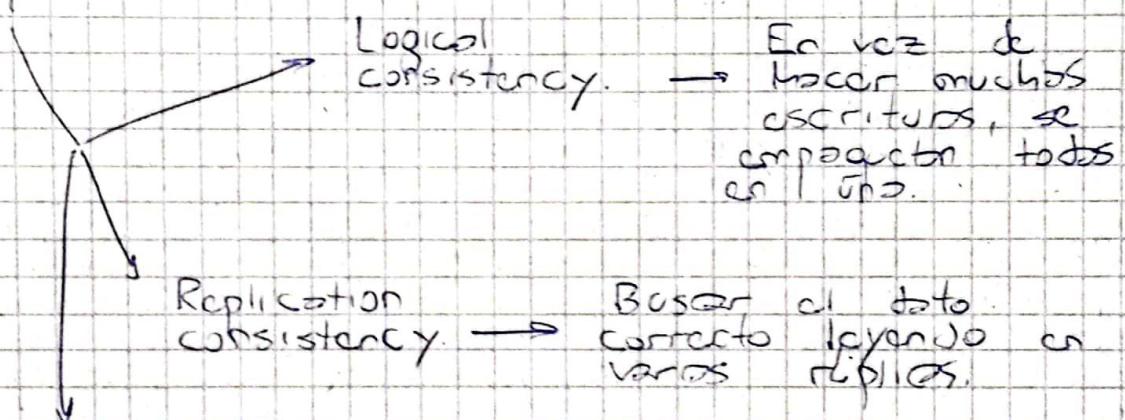
Solicitar al usuario soluciones a los conflictos.

VCS
↳ Version Control System

Read consistency.

Le Conflicto: Leer mientras hay muchas escrituras.

Para solucionar problemas
de lectura mientras hay
muchas escrituras
se puede:



Si se espera suficiente,
todos los replicas estan
actualizados.

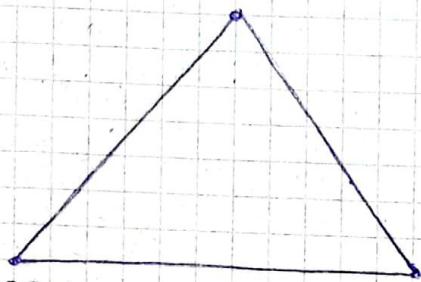
Teorema CAP

"Debemos sacrificar la consistencia
en cambio de disponibilidad".

De 3 propiedades,
solo se pueden elegir
2.

- Consistencia
- Disponibilidad
- Tolerancia a
particionado.

Availability: Los clientes siempre
pueden leer y escribir.



Consistency.

Todos los clientes ven
siempre la
misma información.

Partition tolerance:

} Es posible
particionar
la BD.

Tenant → Ing. lino.
→ (Administrador).

Cloud Computing Usage Patterns

Repasso. Definición.

Computación distribuida, provistos remotamente para recursos estables y medibles.

Características:

- On-demand usage
- Ubícuo (omnipresente) acceso
- Tenencia múltiple (muchos dueños del mismo elemento)
- Elástico
- Uso medida
- Resiliencia.

! Estilos
de arqui.
habituales.

Usage
patterns.

Ejemplos
AWS.

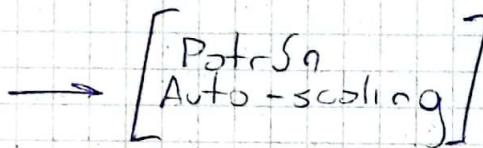
Cloud
Usage
Patterns
22/06.

Patrones populares en cloud computing

• ¿Cómo implementar Match Strategy?

↳ Si en X momento teníamos algún set escalando y descalando horizontalmente según mis necesidades conocidas (por ejemplo: consumo por momento del sistema).

Luego que escala y desciende automáticamente.



Performance
Scalability
Elasticity

Por ejemplo con reglas horarias

o para reaccionar a eventos

12'

• El sistema debe ser escalable horizontalmente!
(por ejemplo que los trabajos sean independientes entre ellos, o que el servicio sea stateless).

Necesitamos

- ↳ API / CLI del cloud provider
- ↳ Sistemas de monitoreo (para compararlos con nuestras reglas)
- ↳ Testing y tuning del auto-scaling implementado.

(Los cloud providers ofrecen auto-scaling y métricas)

(Cada componente)

- ↳ tier debe escalar independientemente.

(Por límites inferiores y superiores a los reglas.)

(Un worker está claudo si tiene un único proceso de larga duración. Aunque use un 1% de las capacidades del worker)

- ¿Cómo evitar los sobrecargas de un servicio sujeto a largos intermitentes (y grandes)?

Availability
Performance
Scalability.

[Queue based load leveling]

Encolar los pedidos, si llegan de forma constante al servicio nuestro servicio.



"Nivelar los largos tráves de una cola de mensajes"

Así no se sobrecarga nuestro servicio.

Pero los servicios bloquen los hacen esperar (dejando de ser sincrónico).

Registramos qué mensajes mandamos y su orden, por si tenemos que reenviarlos.

- ¿Cómo pueden múltiples workers procesar los pedidos de múltiples apps?

"producers" apps
"productors" de requests

[Competing Consumers]

- Cada worker debe poder procesar un mensaje de la app!

"consumers" workers que consumen y procesan requests

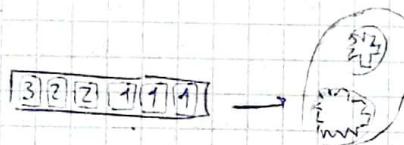
- Podemos auto-scatter los workers!

- Cómo priorizar algunos requests particulares?
(del 1º al 4º de mensajes)

[Priority queue]

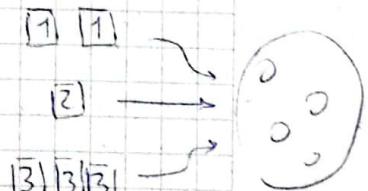
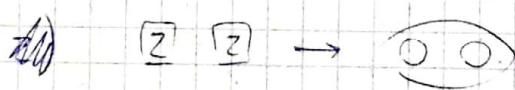
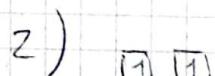
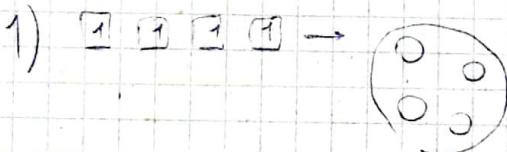
Necesario para cumplir con SLA (Service Level Agreement), por ejemplo.

Priorizar sin importar cuando llegan.



Performance
Scalability

Variantes



- Cómo verificar que se estén cumpliendo availability y performance?

[Health Endpoint Monitoring]

Availability
Performance

Tener un endpoint que hace checks de salud.

Si hubo un error, puede mandar una alerta o tomar una medida.

- ¿Cómo manejar fallas temporales?

Availability.

Patrón

[Retry]

Sí tengo algún problema de conexión, por ejemplo.

Implemento una lógica de reintentos.

- Cuidar que los reintentos no degraden el resto del servicio!

Si siempre me da error (dempotencia) evitar si sigue intentando!

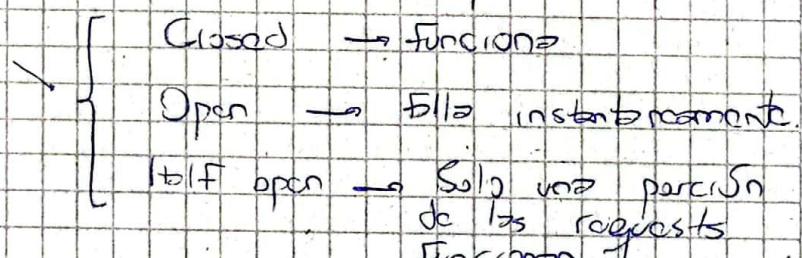
- ¿Cómo evitar llamar a servicios caídos?

Patrón

Circuit Breaker

Sí llego de muchos retries sigue sin funcionar excepto que el servicio esté caído y dejo de llamarlo. Y volver a abrirlo cuando se recupere.

Circuit Breaker



- Necesito detectar si se recupera el servicio

- Ademáts de auto-scaling
¿Cómo puedo controlar el uso de recursos?

[Throttling]

Availability
Performance

Imporar un límite de consumo (más que el auto-scaling esté listo).

Patrón

[Sharding]

Availability
Performance
Sobility

Distribuir el almacenamiento de datos en varias particiones (horizontales). Los datos van a un shard u otro según ~~la~~ ~~key~~ una cierta lógica.

¿Qué es microservice?
paga por
node y npm?

Estilo vs patrón!
(de arquitectura)

App / Service / Layer /

Platform

Stateless
Microservices

Patterns of Enterprise Application

Architecture

29/06/23

¿Por qué "Enterprise"? → Datos: Grandes cantidades
Persistentes
Acceso concurrente (simultáneo)
por varios usuarios

Patrón de
diseño.

• "Solución común
y predefinida
a un problema
de diseño común?"

→ Interfaces de → Muchas interfaces p/ distintos dispositivos
usuario.
o "vistas".

↳ Están integrados con (Por ejemplo
otras apps Enterprise usamos servicios
(REST). de Redis)

Lógica de negocio compleja
(porciones del dominio que debemos
modelar en nuestro sistema)

Discrepancia conceptual

Fronte → todo esto, la industria tiende a:

Layers. {
• Presentation
• Domain
• Data

Cada parte
Abstacta una
percepción.

Permite que cada área
se focalice en su layer, sin
molestar (mucho) en las otras
partes.

Ejemplo: Domain - Driven - Design.

Patrones

• Grupo de patrones para:

[Organización Lsg. de Negocio]

- Transaction Script
- Domain Model
- Table Module
- Service Layer

Patrones para

↳

[Persistencia]

• Los que vamos en este slide.

(acceso =)
datos

[Arquitecturas]

- Table Data Gateway
 - Row Data Gateway
 - Active Record
 - Data Mapper
- Identity field
Foreign key map
Association Table map
Single table inheritance
Class table inheritance
Concrete table inheritance

[Comportamiento]

[Metadatos]

- Lazy Load
- Unit of Work
- Identity map

Patrones para

[Presentación]

- MVC
- Page controller
- Front controller
- Application controller
- Template View
- MVP
- MVVM

[Distribución]

(como distribuir,
si y tenemos
capas logica
en capas fisicas?)

[Conurrencia]

- Optimistic Lock
- Pessimistic Lock
- Coarse - Grained Lock
- Implicit Lock

Como
tratar,
reducir
el
diseño
cuando

varios
usuarios
consumen la
misma info

Programación estructurada.

Aplican en los
casos de negocio

[Organización lógica de negocio.]

- 3 patrones de diseño principales según el tipo de programación que estamos usando:

1) Programación estructurada

Patrón:

[Transaction script]

Un procedimiento por cada transacción

Un mensaje o función para cada cosa que se quiere hacer.

Los parámetros no tienen lógica propia, todo se procesa dentro de la transacción.

Es probable que repitamos código!

2) Prog. orientada a objetos

[Domain model]

Objetos y mensajes

Cada elemento del dominio tiene un objeto en memoria.

"100 cursos"

↓
100 instancias de la clase Curso.

Todos los procedimientos se responden en distintos objetos.

Y los parámetros son objetos del dominio (si tienen lógica propia).

3) Modelo centrado en datos

[Table module]

Una clase por cada entidad de datos.

No aprovecha P.O.O.

- 4to. patrón. (que incluimos además de alguno de los otros 3):

[Service Layer]

Frontera de nuestra aplicación.
(API, Application Programming Interface)

Límite qué se expone al exterior. En

API solo se puede ver una porción del programa.

El exterior no tiene acceso al Domain Model, solo ve la Service Layer.

Business logic = Domain logic + App logic
(Domain model) (Service layer)

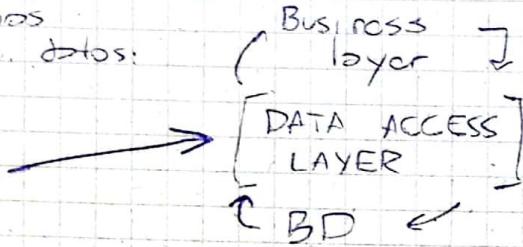
Todo el código no necesario manda rotulado, con

Persistencia] Patrones que se aplican a
[Db Access Layer.
(parte de la layer DB)

Patrones arquitectivos
de persistencia de datos:



YOU
ARE
HERE



ORM
Object -
Relational
Mapping.

↓
Relacionar
objetos P.O.O. con
BD relacionales.

- Table Data gateway } Estilo base gateway
- Row Data gateway }
- Data Mapper } Estilo base mapper.
- Active Record.

Gateway y
Mapper son
métodos de
pasar objetos
→ bases de
datos.

Preguntas entrevista.

Códigos de error, Rest API, complejidad algorítmica

CRUD (create, read, update, delete)

POST GET PUT DELETE

↓ ↓

Parámetros en body.

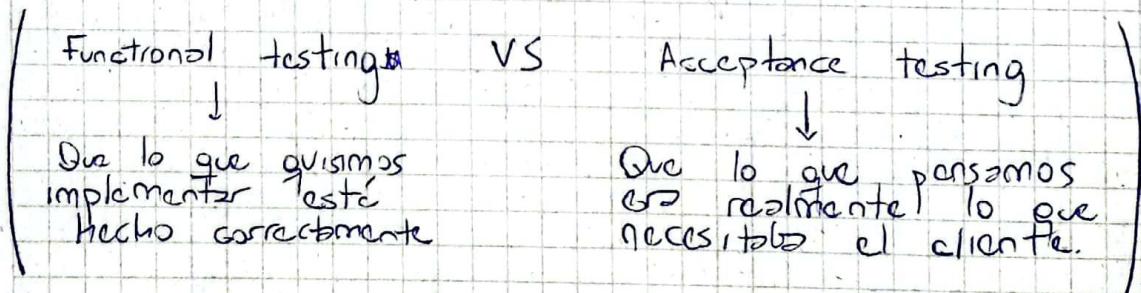
Parámetros en URL

Rest → REpresentation1 state Transfer.

HTTP → Protocolo usado na internet.

NoSQL → BD no relacional. No tabulado (como Excel).

- Continuous integration → Probar lo que acabas de implementar con el resto del sistema.
 - Continuous delivery
 - Continuous deployment.



Web service → La comunicación con / uso de un servicio a través de la web.
Esto permite encapsular el servicio,
y que se use a través de la web.

Virtual box
(VM classics) → { Host OS (por ej. Windows)
vs + Guest OS (" : Linux)

Docker → { Host OS
(más)

↓
Corre tu
APP directamente.

DevOps → Development + Operation

En la intersección
de ambos estás
Dev Ops.

Docker → Herramientas para deployar
(arrancar) containers de forma automática.

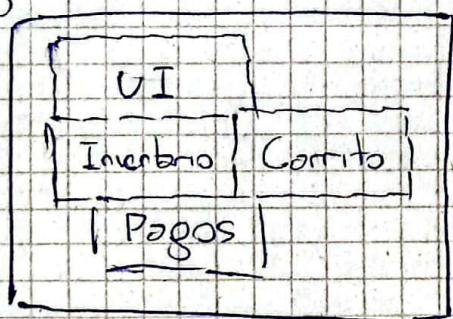
Containers: "Espacio" donde estás
todo lo necesario para correr una
APP, herramientas o Framework.

Monolith

VS

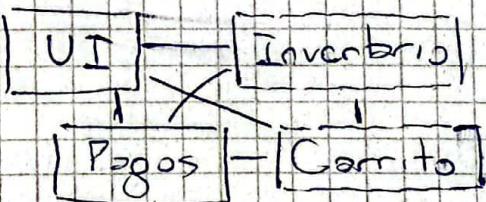
Microservices

Todo un programa
hecho en un
mismo bloque, y
en un mismo
programa.



Cada parte del sistema
tiene su propio servicio
en su propio container

y se comunican entre
ellas con un API.



No puedo escalar una
parte en particular,
tengo que reproducir
todo juntos.
O no puedo combinar
solo una parte, y
seguir usando el
resto.

Puedo reemplazar o
escalar alguna en
particular
y no tengo que usar
el mismo lenguaje
para todos.

ACID → Propiedades de BD relacionales

- Atomicity → La transacción se hace enteramente o no se hace.
- Consistency → Una transacción transforma la BD de una BD válida a otra BD válida.
- Isolation → Varias transacciones concurrentes tienen el mismo impacto que esos mismos, separadamente.
- Durability → Las transacciones son durables, incluso con pérdida de energía.

HTTP codes

1xx → Respuestas informativas

2xx → Éxito

3xx → Redirecciones

4xx → Error del lado del cliente

5xx → Error en el servidor.

User → Persona que usa el computador.

User agent → Software utilizado para comunicarse con el servidor (web browser).

Client → Código de la aplicación

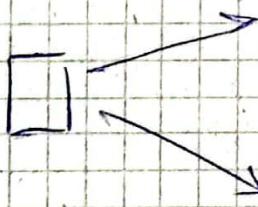
Stateless vs Stateful

Cuando me logeo, el servidor no retiene mi sesión sino que me da un token (que el cliente retiene).

Cada vez que quiero hacer algo, paso el token (y el nodo lo ve al consultar a un lugar central).

Al logearme, el servidor retiene todo la información de mi sesión. Falla si usamos un load balancer y NO vuelvo a ver donde estaba logeado.

Scaling.

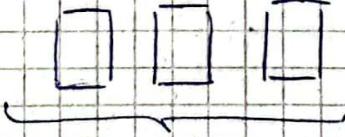


Vertical



llego más potente
lo que tengo.

Horizontal



llego más copias
de lo que tengo
(solo una con la
misma capacidad que
el original).

(regla mnemotécnica:
es como una
ciudad). Encuentro
vertical vs
horizontal).