Arquitectura de Software

Arquitectura de software (Universidad ORT Uruguay)

Resumen de Arquitectura de Software

Arquitectura de Software: 2

Atributos de calidad 9
- **Modificabilidad** 11
- **Performance** 15
- **Security** 17
- **Disponibilidad** 19
- **Interoperabilidad** 25

Patrones 27
- **Pipes and Filters** 27
- **Publish and Subscribe** 28
- **CQRS** 29
- **Multi Tiers** 31
- **Gatekeeper** 33
- **Federated identity** 35
- **SOA** 36

Documentando arquitecturas 40

Arquitectura de Software:

La arquitectura de software de un sistema es el conjunto de estructuras necesarias para razonar sobre el sistema que comprende los elementos del software, las relaciones entre ellas y las propiedades de ambos.
Algunas decisiones de arquitectura se hacen temprano pero no todas.
Una estructura es un conjunto de elementos mantenidos juntos por una relación.
Los sistemas están compuestos por muchas estructuras.
La arquitectura es un conjunto de estructuras de software.
Hay tres categorías de estructuras de arquitectura que juegan un rol importante en diseño, documentación y análisis:

1) **Módulos.**
    - Algunas estructuras dividen los sistemas en unidades de implementación (módulos).
    - Son asignados a determinadas responsabilidades computacionales y son la base de asignaciones de trabajo para los equipos de programación.
    - En proyectos grandes los módulos se subdividen y se asignan subequipos.
2) **Componentes y conectores.**
    - Otras estructuras se centran en la forma que interactúan los elementos entre ellos en tiempo de ejecución para obtener las funciones del sistema.
    - En nuestro uso, un componente siempre es una entidad de tiempo de ejecución
3) **Asignación.**
    - Describe el mapeo de estructuras de software a los ambientes del sistema organizacionales, desarrollo, instalación y ejecución.

Una estructura es arquitectónica si apoya el razonamiento entre el sistema y las propiedades de él.
El razonamiento debería ser sobre un atributo del sistema que sea importante para algunos stakeholders.
Incluye las funcionalidades alcanzadas por el sistema, la disponibilidad del sistema ante las fallas, la dificultad para hacer ciertos cambios, el nivel de respuesta del sistema a los requests del usuario, etc.
Todos los sistemas tienen una arquitectura. Tienen elementos y relaciones entre ellos, pero la arquitectura puede no ser conocida por todos. Puede ser que la gente que diseña el sistema ya se fue, o la documentación desapareció o el código fuente se perdió. La arquitectura puede existir independientemente de su descripción o especificación.
La arquitectura incluye comportamiento. Este comportamiento engloba cómo los elementos interactúan entre ellos.

Vistas

Una vista es una representación de un conjunto coherente de elementos arquitectónicos, como escritos y leídos por los stakeholders.
Una estructura es un conjunto de elementos que existen en el software o hardware. En resumen, una vista es una representación de una estructura. Los arquitectos diseñan estructuras y documentan vistas de esas estructuras.

Estructuras de módulo

Engloban decisiones sobre cómo el sistema será estructurado como un conjunto de código o unidades de datos que deben ser construidas o adquiridas.

En cualquier estructura de módulo los elementos son módulos de algún tipo (como clases, capas o divisiones de funcionalidades). A los módulos se les asignan áreas de responsabilidad funcional. Refiere a código fuente o datos que deben construirse.

Responde:

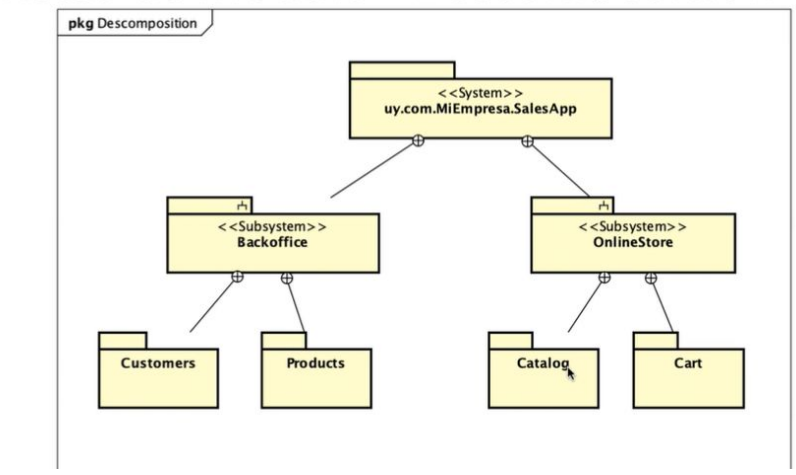
- Cuáles son las funcionalidades primarias de cada módulo
- Cuales otros elementos del software son un módulo permitido para usar
- Qué otro software usa y depende de
- Qué módulos son relacionados a otros módulos por relaciones de generalización y especialización

Estructuras de descomposición

Unidades de módulos que se relacionan entre ellas con la relación de "is-a-submodule-of". Muestra cómo los módulos se descomponen en módulos más pequeños.

Determina la modificabilidad.

Muestra cómo se organizan jerárquicamente los módulos de un proyecto. Los módulos son los de abajo del todo.

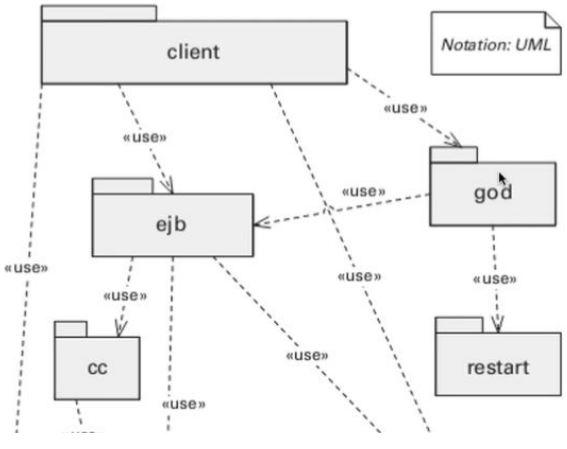


Estructuras de uso

Las unidades son módulos y tal vez clases.

Las unidades se relacionan por las relaciones de uso. Una unidad usa otra si la primera necesita la presencia de una versión funcional de la segunda.

Permite ver el impacto del cambio en un paquete.

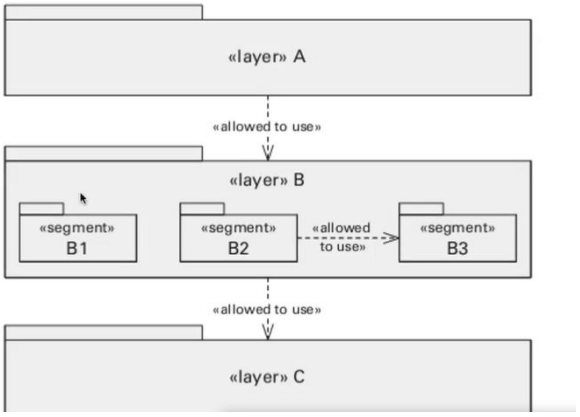


Estructura de capas (layers)

Una capa es una “máquina virtual” abstracta que provee un conjunto de servicios cohesivos a través de una interfaz.

Esta estructura está impregnada a un sistema con portabilidad y la capacidad de cambiar la plataforma de cómputo.

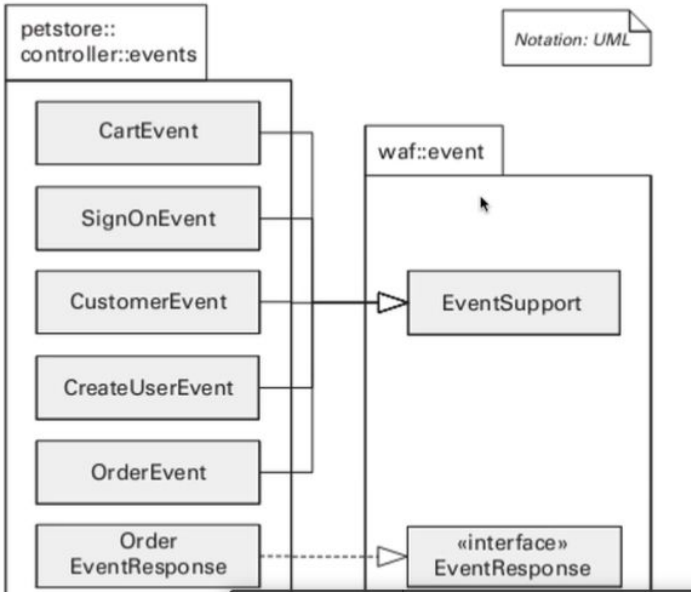
Muestra cómo se organizan los módulos respecto a responsabilidades.



Estructuras de clase

La relación es de herencia de o es una instancia de.

Si existe alguna documentación para un proyecto, que haya seguido un análisis y diseño de proceso orientado a objetos, es esta estructura.



Modelo de datos

Describe la estructura de información estática con respecto a las entidades de datos y sus relaciones.

Estructuras de componentes y conectores

Engloban decisiones sobre cómo el sistema va a estar estructurado como un conjunto de elementos que tienen comportamiento de runtime (componentes) e interacciones (conectores).
Los elementos son componentes de runtime como servicios, peers, clientes, servidores, filtros, etc.
Los conectores son los de la comunicación como call return, procesos de sincronización, pipes, etc.
Responde:

- Cuáles son los componentes más importantes ejecutándose y cómo interactúan en tiempo de ejecución
- Cuáles son los almacenes de datos más importantes
- Cuáles partes del sistema están replicadas
- Cómo progresan los datos en el sistema
- Qué partes del sistema pueden correr en paralelo
- Puede la estructura del sistema cambiar mientras se ejecuta y cómo

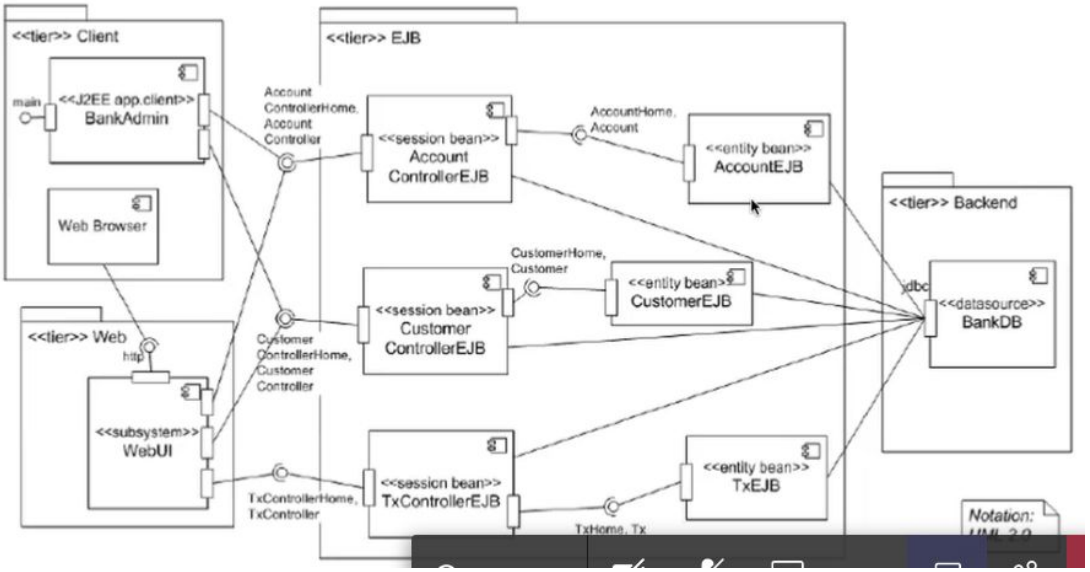
Son importantes para responder perguntas sobre performance, seguridad, disponibilidad, etc.

Estructuras de servicio

Las unidades son servicios que interoperan entre ellas por mecanismos de coordinación de servicios como SOAP.

Estructuras de concurrencia

Ayuda a determinar las oportunidades para paralelismo y la ubicación donde la contención de recursos ocurriría.



Estructuras de asignación

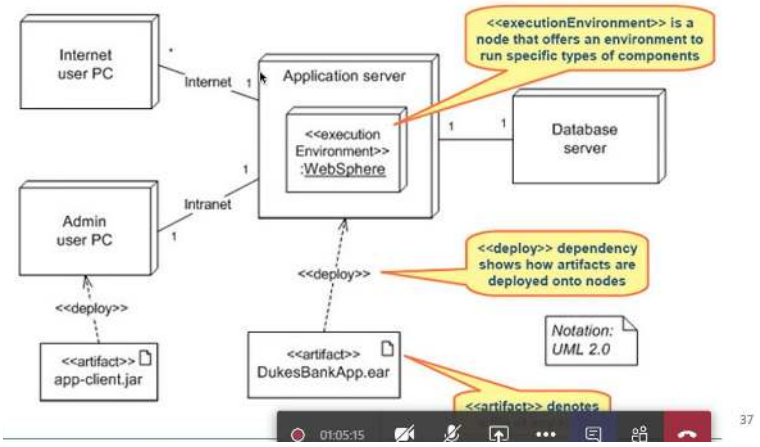
Muestran la relación entre los elementos del software y elementos en uno o más ambientes externos para los cuales el software es creado y ejecutado.
Responde:

- En qué procesador se ejecuta cada elemento del software
- En qué directorios o archivos se guardan los elementos durante el desarrollo, testing y construcción del sistema
- Cuál es la asignación de cada elemento del software a equipos de desarrollo

Estructura de despliegue

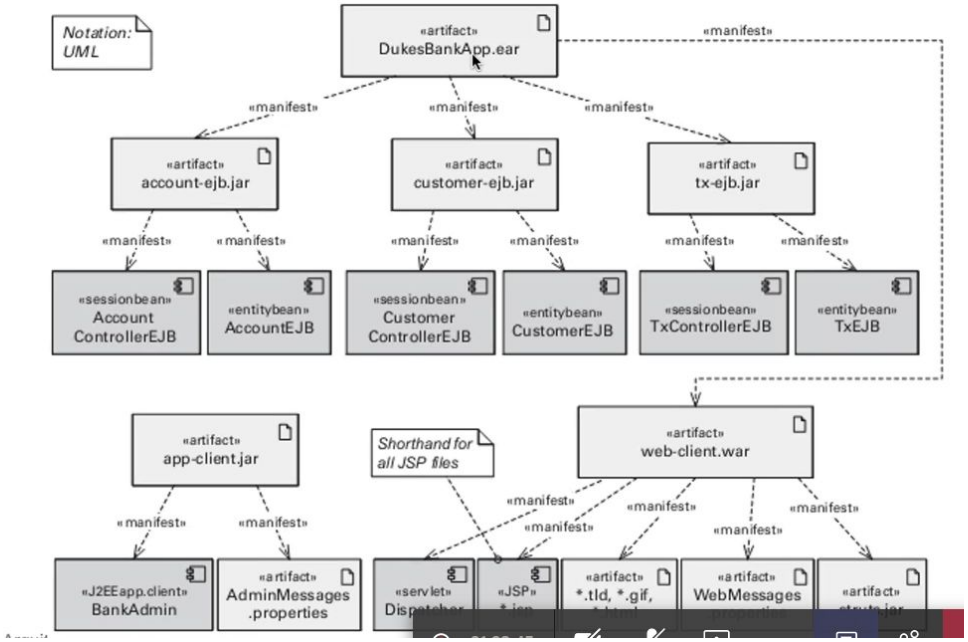
Muestra cómo el software se asigna al procesamiento del hardware y la comunicación de elementos.

Los elementos son del software, entidades del hardware y caminos de comunicación.



Estructura de implementación

Muestra cómo los elementos del software están mapeados a las estructuras de archivos en el desarrollo del sistema, integración o control de configuración. Es importante para administrar las actividades de desarrollo y procesos de construcción.



Estructura de asignación de trabajo

Asigna responsabilidades para implementar y integrar los módulos a los equipos que van a llevarlo a cabo. También determina la comunicación entre equipos (teleconferencias, email, etc).

Patrones de arquitectura

Un patrón de arquitectura delinea los tipos de elementos y sus formas de interacción usadas para resolver el problema.

Módulos:

Layered Pattern. Cuando las relaciones de uso entre los elementos del software son estrictamente unidireccionales, un sistema de capas emerge.

Componentes y conectores:

Shared-data pattern. Comprende componentes y conectores que crean, guardan y acceden a datos persistentes. El repositorio toma forma de base de datos. Los conectores son protocolos que manejan datos como SQL.

Client-Server pattern. Los componentes son los clientes y servidores. Los conectores son protocolos y mensajes que comparten.

Allocation:

Multi-tier pattern. Describe cómo distribuir y asignar los componentes de un sistema en subconjuntos de hardware y software.

Competence center pattern y platform pattern. Estos patrones se especializan en la estructura de asignación de trabajo.

Rules of thumb

- La arquitectura debería ser el producto de un solo arquitecto o un pequeño grupo con un líder técnico.
- El arquitecto debe basar la arquitectura en una lista con atributos de calidad de requerimientos especificados con prioridades.
- La arquitectura debe ser documentada usando vistas.
- La arquitectura debe ser evaluada por su capacidad para entregar los atributos de calidad más importantes.
- La arquitectura se debe prestar para implementación incremental.
- La arquitectura debe presentar módulos bien definidos que sus responsabilidades funcionales son asignadas a los principios de información escondiendo y separando importancias.
- La arquitectura nunca debería depender de una versión del producto.
- Los módulos que producen datos, deben estar separados de los que consumen datos.
- No hay que esperar una correspondencia uno a uno entre módulos.
- Todos los procesos deben ser escritos para que su asignación a un procesador específico pueda ser cambiado fácilmente, quizás en tiempo de ejecución.

TABLE 1.1 Useful Architectural Structures

	Software Structure	Element Types	Relations	Useful For	Quality Attributes Affected
Module Structures	Decomposition	Module	Is a submodule of	Resource allocation and project structuring and planning; information hiding, encapsulation; configuration control	Modifiability
	Uses	Module	Uses (i.e., requires the correct presence of)	Engineering subsets, engineering extensions	"Subsetability," extensibility
	Layers	Layer	Requires the correct presence of, uses the services of, provides abstraction to	Incremental development, implementing systems on top of "virtual machines"	Portability
	Class	Class, object	Is an instance of, shares access methods of	In object-oriented design systems, factoring out commonality; planning extensions of functionality	Modifiability, extensibility
	Data model	Data entity	{one, many}-to-{one, many}, generalizes, specializes	Engineering global data structures for consistency and performance	Modifiability, performance
C&C Structures	Service	Service, ESB, registry, others	Runs concurrently with, may run concurrently with, excludes, precedes, etc.	Scheduling analysis, performance analysis	Interoperability, modifiability
	Concurrency	Processes, threads	Can run in parallel	Identifying locations where resource contention exists, or where threads may fork, join, be created, or be killed	Performance, availability
Allocation Structures	Deployment	Components, hardware elements	Allocated to, migrates to	Performance, availability, security analysis	Performance, security, availability
	Implementation	Modules, file structure	Stored in	Configuration control, integration, test activities	Development efficiency
	Work assignment	Modules, organizational units	Assigned to	Project management, best use of expertise and available resources, management of commonality	Development efficiency

Atributos de calidad

Los requerimientos del sistema se pueden categorizar en:

Requerimientos funcionales:

Siempre tienen que estar. Define capacidades del sistema, sus servicios y comportamiento.

Atributos de calidad:

Impactan en todos lados en la arquitectura. Es medible y testeable. Mide que tan bueno es el producto de acuerdo a la dimensión. Califican los funcionales. Si satisfacen las necesidades de los interesados.

Se dividen en tiempo de ejecución y tiempo de desarrollo:

A. Disponibilidad (qué tan a menudo falla), performance (que tan rápido se muestra el diálogo), usabilidad (qué tan fácil es aprender esta función), escalabilidad, etc.
B. Modificabilidad, testabilidad, portabilidad, etc.

Especificamos los atributos de calidad como escenarios:

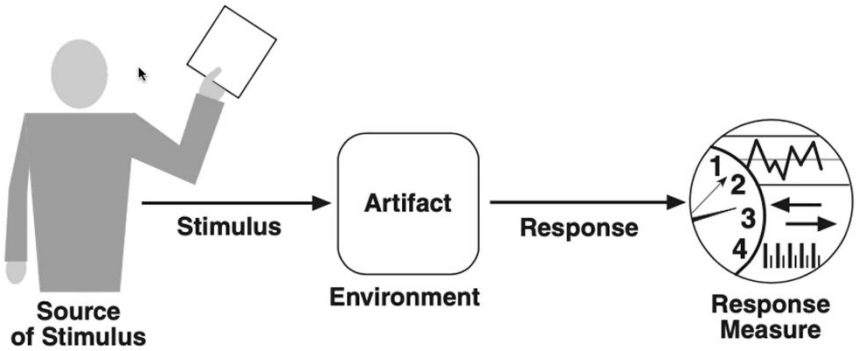
1. Estímulo: describe un evento que llega al sistema.
2. Fuente del estímulo: es una entidad que generó un estímulo (humano, computadora, etc).
3. Respuesta: actividad que se hace cuando llega un estímulo
4. Medida de respuesta: cuando llega la respuesta, debe ser medible para poder testear el requerimiento. De esa forma sabemos si la respuesta es satisfactoria.
5. Ambiente: el estímulo ocurre bajo ciertas condiciones. El sistema puede estar sobrecargado, en estado normal, etc.
6. Artefacto: puede ser una colección de sistemas, todo el sistema, o algunas partes de él.

Son cualidades del producto que están asociadas a la disponibilidad. Existen interacciones entre ellas.

Ejemplo: Hacer tal cosa en menos de 20 milisegundos

(modificabilidad, performance, seguridad) Es importante para cada requerimiento funcional elegir algunos y tenerlos por separado.

Escenarios de calidad

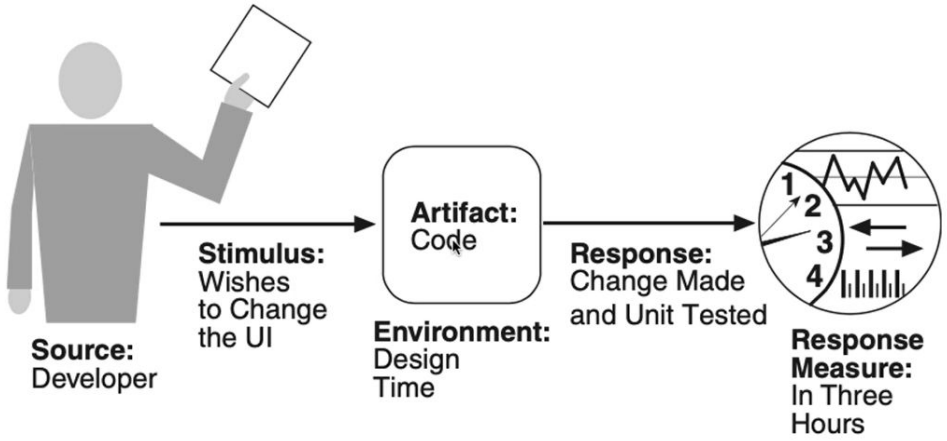


Fuente del estimulo estimulo contexto artefacto respuesta medida de respuesta.

Tácticas

Existen tácticas que ayudan al arquitecto a cumplir con los atributos de calidad. Se usan porque los patrones de diseño son complejos de usar y puede ser que no haya ningún patrón que le sirva. Además de esta forma el diseño se hace más sistemático.
Categorización de decisiones de diseño:

- Asignación de responsabilidades: identificar las responsabilidades principales y determinar cuáles son de runtime y cuáles no.
- Coordination model: identificar los elementos del sistema que deben coordinar y los que estrictamente no deben, determinar las propiedades de coordinación (timelines, completitud, correctitud, consistencia), elegir los mecanismos de comunicación entre esas propiedades.
- Data model: determinar cómo se crean los datos, se inicializan, persisten, manipulan, traducen y destruyen. También se encargan de la organización de la información (su se guarda en una base de datos relacional, colección de objetos o ambas)
- Management of resource: identificar los recursos que deben administrarse y determinar los límites de cada uno, determinar que elementos del sistema manejan cada recurso, cómo se comparten los recursos y el impacto de saturación de cada uno.
- Mapping among architectural elements: el mapeo de módulos con los elementos del runtime, es decir qué elemento es creado por qué módulo, la asignación de elementos del runtime a procesadores, asignación de elementos del data model a data stores, mapeo de módulos y elementos del runtime a units of delivery
- Binding time decisions: ejemplo, para la decisión de coordination model se pueden diseñar protocolos de negociación en tiempo de ejecución.
- Choice of technology: decidir qué tecnologías hay para tomar decisiones hechas en otras categorías, determinar si las herramientas para manejar la tecnología son adecuadas, determinar si es correcta la familiaridad con las tecnologías (cursos, tutoriales, etc), determinar si una nueva tecnología es compatible con las ya existentes.



Restricciones:

Son decisiones de diseño impuestas, el arquitecto no puede decidir, tiene que acatarlas. Por ejemplo: Esto lo tenes que hacer en 3 meses.

Modificabilidad

Refiere al grado en que el sistema permite realizar cambios minimizando el riesgo y el costo del cambio.

Responde las preguntas:

- Qué puedo cambiar?
- Cuál es la probabilidad de cambio?
- Cuándo se hace el cambio y quién lo hace?

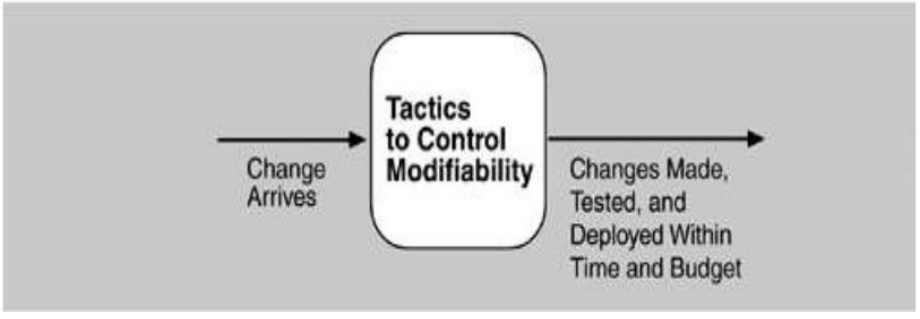
Cambios refiere a:

- Agregar, modificar, eliminar características o funcionalidades
- Reparar defectos, mejorar atributos de calidad
- Incorporar nuevas tecnologías o plataformas, etc

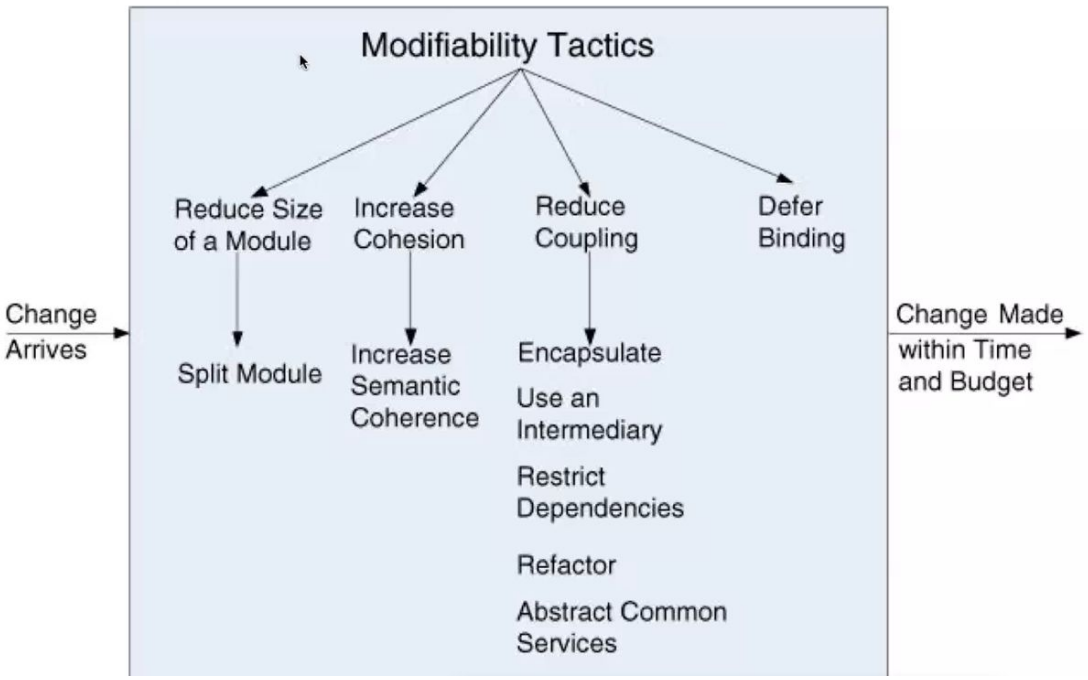
Modifiability General Scenario

Portion of Scenario	Possible Values
Source	End user, developer, system administrator
Stimulus	A directive to add/delete/modify functionality, or change a quality attribute, capacity, or technology
Artifacts	Code, data, interfaces, components, resources, configurations, ...
Environment	Runtime, compile time, build time, initiation time, design time
Response	One or more of the following: . make modification . test modification . deploy modification
Response Measure	Cost in terms of: . number, size, complexity of affected artifacts . effort . calendar time . money (direct outlay or opportunity cost) . extent to which this modification affects other functions or quality attributes . new defects introduced

Las tácticas son para controlar la complejidad de hacer cambio, su tiempo y su costo.



Modifiability tactics



Reducir el tamaño de un módulo

Dividirlo: Si el módulo va a ser modificado que tiene mucha capacidad, el cambio va a ser muy costoso. Para eso se divide en módulos más pequeños.

Aumentar la cohesión

Aumentar la coherencia semántica: Si las responsabilidades A y B en un módulo no tienen el mismo propósito, deberían estar en distintos módulos. Puede que se tenga que crear un nuevo módulo.

Reducir el acoplamiento

Encapsular: introduce una interfaz explícita a un módulo que incluye una API y tiene responsabilidades asociadas.

Usar un intermediario: dada una dependencia entre dos responsabilidades, la dependencia se puede romper agregando un intermediario.

Restricción de dependencias: restringir los módulos con los que un módulo interactúa o depende.

Refactorizar: cuando dos módulos son afectados por el mismo cambio porque son duplicados.

Abstract common services: cuando dos módulos brindan dos servicios similares, puede servir implementar los servicios en una forma más general.

Defer Binding

En general, cuanto más tarde en el ciclo de vida se puedan enlazar los valores, mejor. Si diseñamos artefactos con flexibilidad, ejercitar esa flexibilidad es usualmente más barato que codificar a mano un cambio específico.

- Asignación de responsabilidades: qué cambios son más probables de ocurrir por consideración de cambios por aspectos técnicos, legales, sociales, de negocio, o por el cliente.
- Coordination model: que funcionalidad o atributo de calidad puede cambiar en runtime y cómo afecta a la coordinación.
- Data model: determinar que cambios a las abstracciones de datos, sus operaciones o sus propiedades son más probables de ocurrir.
- Management of resource: determinar cómo agregar, borrar o modificar una responsabilidad o AC va a afectar el uso de recursos.
- Mapping among architectural elements: determinar si es deseable cambiar el sentido en que las funcionalidades son mapeadas a los elementos computacionales en runtime, tiempo de compilación, tiempo de diseño o tiempo de construcción.
- Binding time decisions: determinar la ultima vez que un cambio va a necesitar hacerse, introducir un mecanismo para diferir la unión, determinar el costo de introducir el mecanismo y el costo de hacerlo.
- Choice of technology: determinar qué modificaciones son más fáciles de hacer o más difíciles por las elecciones de tecnología

Ejercicios

1) A modificabilidad
   B interoperabilidad, disponibilidad
   C portabilidad y modificabilidad
   D disponibilidad(que funcione correctamente)
2) Escenario A: disponibilidad
   Escenario B: Si ”puede deshabilitar un conjunto de funcionalidades que utilizan los usuarios” aunque no se porque es en tiempo de ejecución
   Escenario C: --
   Escenario D: si, pues hay que modificar cosas
3) A quiero que las clases que hacen lo mismo estén juntas
4) B
5)
6)

Performance

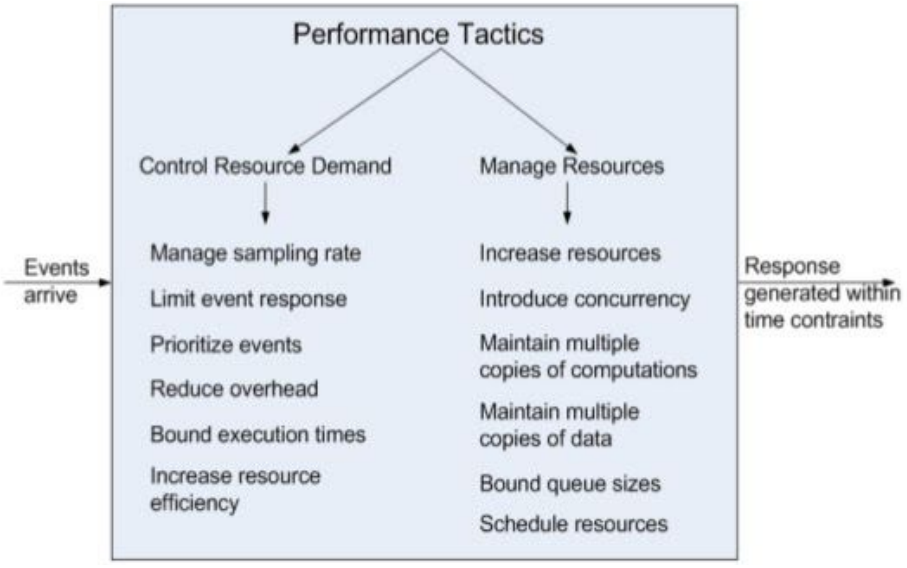
Es la habilidad del sistema para hacer los requerimientos en tiempo. Cuando ocurre un evento, el sistema debe responder a tiempo. Se trata de reconocer los eventos que pueden ocurrir, y cuando, para tener una respuesta en tiempo. Puede medirse en latencia o en capacidad de respuesta.

Factores que pueden influir:

- Acceso a datos
- Capacidad del hardware
- La cantidad de datos que recibe, es decir cuánto lo están demandando

Portion of Scenario	Possible Values
Source	Internal or external to the system
Stimulus	Arrival of a periodic, sporadic, or stochastic event
Artifact	System or one or more components in the system.
Environment	Operational mode: normal, emergency, peak load, overload.
Response	Process events, change level of service
Response Measure	Latency, deadline, throughput, jitter, miss rate

Las tácticas son para generar la respuesta a un evento en un tiempo restringido.



Tácticas:

- Controlar la demanda de recursos: a menos demanda de los recursos más capacidad para procesar los eventos.
- Gestionar los recursos: cuando no se puede controlar la demanda de recursos se pueden gestionar.

Controlar la demanda de recursos:

Manejar sampling rate: si es posible reducir la frecuencia de muestreo en la que el flujo de datos se captura, entonces la demanda puede ser reducida con alguna pérdida de fidelidad. Ignorando eventos que son similares. Por ejemplo, ignorando eventos porque son similares.

Limitar la respuesta de eventos: procesar eventos solo hasta una velocidad máxima permitida para asegurar más predictibilidad procesando cuando los eventos son en realidad procesados. Se pueden manejar con un buffer, para procesarlos más lento.
Priorizar eventos: si no todos los eventos tienen la misma importancia, se puede imponer un esquema de prioridad que ordene los eventos de acuerdo a la importancia de hacerlos.
Reducir el overhead: el uso de intermediarios incrementa los recursos consumidos al procesar un event stream, eliminarlos mejora la latencia.
Limitar el tiempo de ejecución: para responder a un evento a costo de resultados menos precisos.
Incrementar la eficiencia de recursos: mejorar los algoritmos en áreas críticas para reducir la latencia.

Gestionar los recursos:
Incrementar los recursos: más procesadores, más rápidos, más memoria, redes más rápidas que tengan el potencial para reducir la latencia.
Incrementar la concurrencia: si las solicitudes pueden hacerse en paralelo, el tiempo de bloqueo puede reducirse. La concurrencia puede introducirse usando threads o algoritmos en paralelo.
Mantener varias copias de cómputos: el propósito de réplicas es reducir la contención que ocurriría si todos los cómputos se hicieran en un solo servidor. Replicar los componentes en distintos procesadores.
Mantener varias copias de los datos: mantener copias guardadas con diferentes tiempos de acceso, teniendo en cuenta la sincronización.
Limitar el tamaño de los buffers o MQ: controlar el máximo número de cosas encoladas y consecuentemente los recursos usados para procesarlos.
Schedule de recursos: dar más tiempo de procesador a recursos más solicitados.

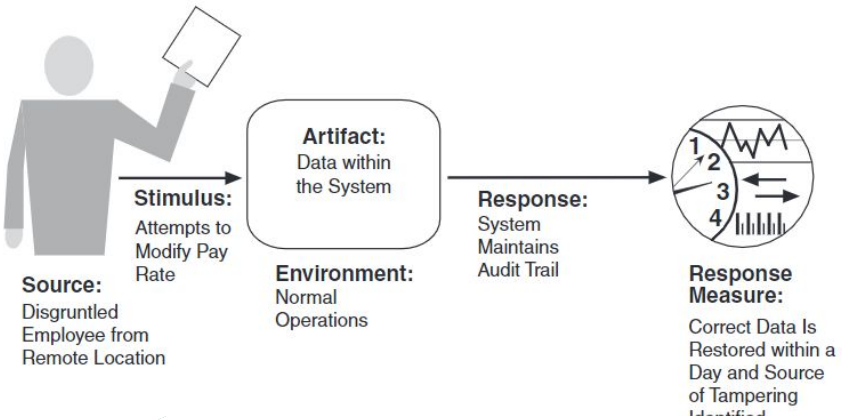
Cuando un sistema no tiene más memoria, lleva memoria a disco del estado de un programa para poder ejecutar otro. Hay algunos sistemas operativos que no pueden hacer eso y es cuando se empieza a degradar el sistema. Los procesos pueden bloquearse.

- Las tácticas se usan según lo que uno quiere lograr.
- Cuando uno usa una táctica siempre implica usar otra, incluso hay algunas que están asociadas.
- El pipes and filters incrementa la coherencia semántica, encapsula porque solo se ve lo que tienen que pasar, se usan los pipes de intermediarios, restringe dependencias y los enlaces se difieren en tiempo de desarrollo o iniciación.

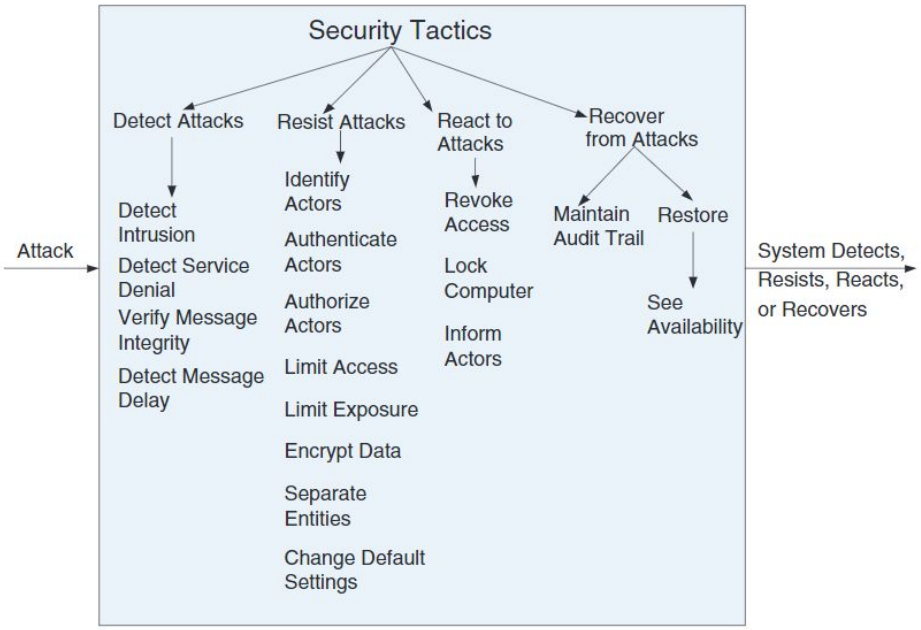
Security

Es la habilidad de un sistema de proteger los datos e información de accesos no autorizados, mientras se sigue brindando acceso a personas y sistemas autorizados.
Tres características que caracterizan la seguridad: confidencialidad, integridad, disponibilidad.
Confidencialidad: los datos e información están protegidos de acceso no autorizado.
Integridad: los datos o servicios no pueden ser manipulados por accesos no autorizados.
Disponibilidad: el sistema va a estar disponible para uso legítimo (si hay un ataque, yo lo puedo seguir usando).
Otras características:
Autenticación: verifica si las identidades son quienes realmente dicen ser
No repudio: el que envía el mensaje no puede luego negar haber enviado el mensaje, y el que lo recibe no puede negar haberlo recibido.
Autorización: le da al usuario el privilegio de realizar una tarea

Se hacen árboles de ataque para determinar los posibles ataques, donde la raíz son los ataques exitosos y los nodos son posibles causas del ataque, las hojas son los estímulos en el escenario. Un ataque es un intento de romper CIA.
Fuente del estímulo: la fuente puede ser humano o otro sistema. Puede haber sido previamente identificado o no. Un humano puede ser de afuera o de adentro de la organización.
Estímulo: es un ataque. Lo caracterizamos como un intento no autorizado de mostrar datos, cambiar o eliminar datos, accesos a los servicios del sistema, cambiar el comportamiento o reducir la disponibilidad.
Artefacto: es el objetivo del ataque que puede ser los servicios del sistema, los datos, o la los datos producidos o consumidos por el sistema. Algunos ataques se hacen en ciertos componentes del sistema que se sabe que son vulnerables.
Ambiente: el ataque puede venir cuando el sistema está online o offline, conectado o desconectado de la red, detrás de un firewall o abierto a una red, totalmente operativo, en parte o no operativo.
Respuesta: el sistema debe asegurar que las transacciones se hacen de forma que los datos o servicios son protegidos de accesos no autorizados. Los datos o servicios no son manipulados sin autorización.
Medida de respuesta: que tanto un sistema está comprometido cuando un componente particular o datos están comprometidos, cuánto tiempo pasó desde antes que se detectara el ataque, cuantos ataques se resistieron, cuanto tiempo tomó recuperarse de un ataque exitoso y cuantos datos fueron vulnerables en ese ataque.



Tácticas:



Detectar el ataque

Detectar intruso: Detectar patrones dentro del sistema anormales a distinto nivel (red, transacciones etc.), detectar que hay un intruso

Detectar negación del servicio: Detectar patrones desde fuera del sistema anormales a distinto nivel (red, transacciones etc.), detectar que están atacando

Verificar la integridad del mensaje: Verificar que los mensajes request no estén alterados

Detectar delay del mensaje: Detectar que la intercepción de los mensajes

Resistir ataques

Identificar los actores: Identificar las fuentes de entradas la sistema

Autenticar los actores: lograr saber que la persona es quien dice que es

Autorizar los actores: que puedo hacer yo en el sistema

Limitar el acceso: poner barreras, por ejemplo firewall, para estar revisando constantemente que es lo que llega.

Limitar la exposición: esconder cosas.

Encriptar información: para asegurar confidencialidad.

Separar entidades: físicamente, por ejemplo servidores, datos, etc.

Cambiar las configuraciones por defecto

Reaccionar a ataques

Revocar el acceso: la parte que están atacando, impedir que entre más gente

Bloquear la computadora: no se usa más nada, esa computadora la aparto.

Informar a los actores: notificar de un ataque.

Recuperarse de ataques

Log disponibilidad: guardar que usuario hace que acción y los efectos.

Log auditoría: registra lo que hace cada usuario.

https://onedrive.live.com/?authkey=%21ANGoTmQfH9LiWUs&cid=5047D9112BE92C8A&id=5047D9112BE92C8A%21329773&parId=5047D9112BE92C8A%21329771&o=OneUp

Disponibilidad

https://onedrive.live.com/?authkey=%21ABEowpihy3zSDRU&cid=5047D9112BE92C8A&id=5047D9112BE92C8A%21329774&parId=5047D9112BE92C8A%21329771&o=OneUp

Es la propiedad del software que refiere a que estará allí pronto para hacer sus tareas cuando se necesite. Cuando el sistema se rompe, se recupera solo. Enmascara defectos o imperfecciones para que el período acumulado fuera de servicio no exceda un valor especificado para un intervalo de tiempo.

Confianza (dependability) es la habilidad de evitar las fallas que son más frecuentes y severas de lo aceptable.

Falla: desviación del sistema de su especificación. Es visible

Fault: causa de la falla.

Error: estados intermedios entre la falla y el culpable.

La disponibilidad se calcula: **MTBF/(MTBF + MTTR)**

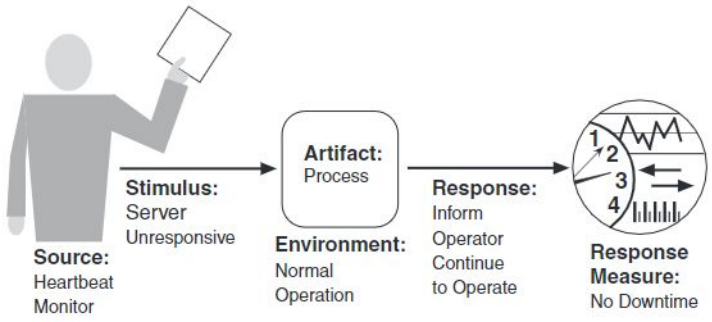
MTBF: tiempo medio entre fallas.

MTTR: tiempo medio en repararse.

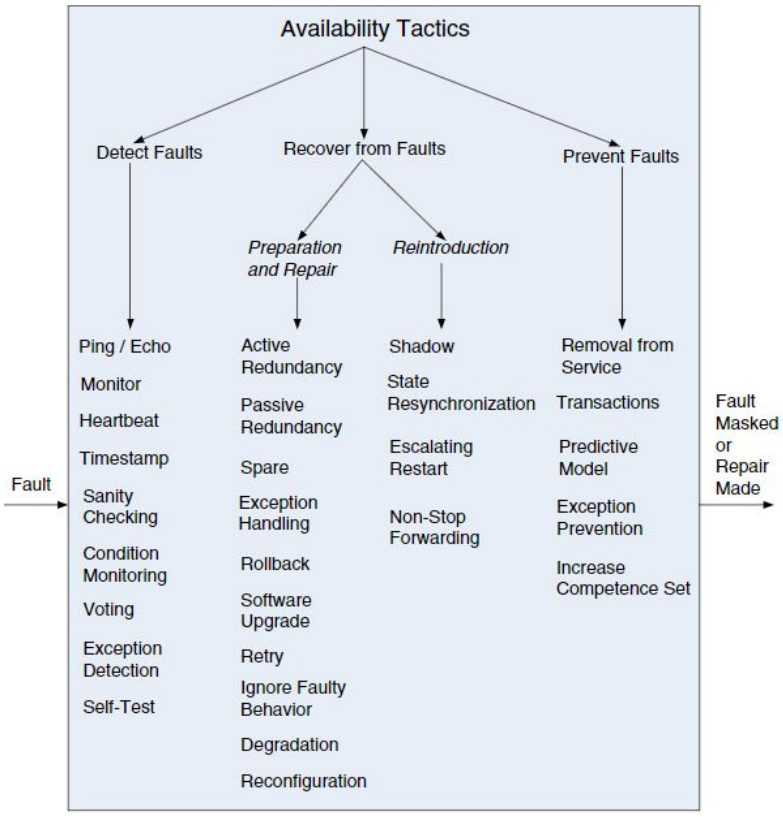
Implica confiabilidad y recuperabilidad. Las fallas no siempre implican la caída del sistema.

Availability	Downtime/90 Days	Downtime/Year
99.0%	21 hours, 36 minutes	3 days, 15.6 hours
99.9%	2 hours, 10 minutes	8 hours, 0 minutes, 46 seconds
99.99%	12 minutes, 58 seconds	52 minutes, 34 seconds
99.999%	1 minute, 18 seconds	5 minutes, 15 seconds
99.9999%	8 seconds	32 seconds

Se considera aceptable 5 nueves.



Tácticas



Detectar defectos

Esta categoría de tácticas intenta detectar defectos y tomar acciones

Ping/Echo: intercambios de request / response asíncronos entre nodos. Para ver si hay delay y si el elemento pingeado está activo. En gral lo usa el Monitor.

Monitor: componente usado para ver el estado de salud del otras partes del sistema. Orquesta otras tácticas de esta categoría. Ejemplo hospital (cuando deja de latir)

Heartbeat: envío de mensajes periódicos entre el monitor y los procesos monitoreados. El intercambio lo inicia el monitoreado. Se puede usar el mensaje para hacer piggybacking

Timestamp - permite detectar secuencias incorrectas de eventos

Sanity check: valida operaciones específicas en base al diseño interno del componente y su estado.

Condition monitoring: se usa para verificar las condiciones en procesos o dispositivos en base a supuestos del diseño (es más específico que sanity check)

Voting: emplear 3 componentes para hacer lo mismo y que envían al respuesta a la lógica de voto. En caso de inconsistencias reporta la falla. Normalmente decide que resultado tomar.

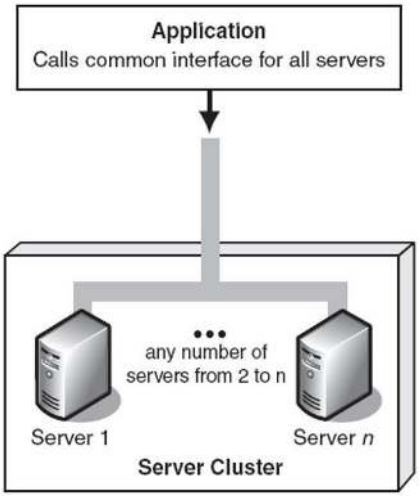
Replication: forma simple de voto, los componentes son los mismos (clones). Útil para hardware

Functional redundancy: forma de voto pero con distintas implementaciones.

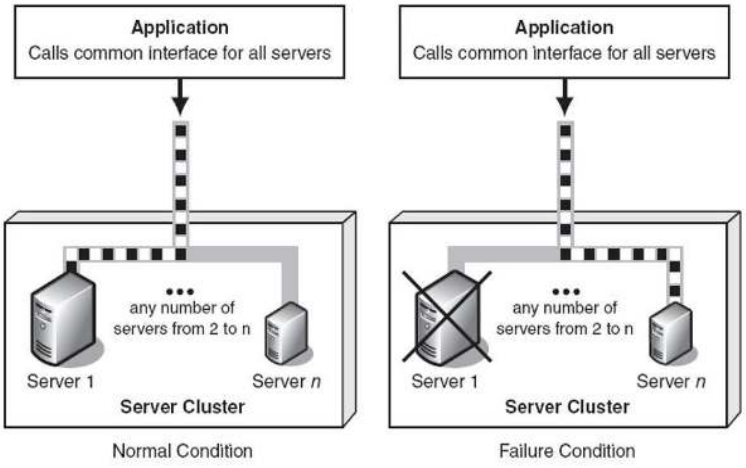
Analytical redundancy: redundancia basada en distintas entradas y salidas e implementaciones, es para tolerar errores en especificaciones de requerimientos. Ej formas de calcular algo en base a la información de distintos tipos de sensores

Exception detection: es la detección de una condición del sistema que altera el flujo normal de ejecución
Self test: verifica el correcto funcionamiento, puede ser en startup, a pedido del monitor, o periódicamente realizada por el propio componente

Recuperar ante defectos
Esta categoría de tácticas se divide en:
Preparación y reparación
Se basa principalmente en combinar redundancia y retries
Active redundancy:nodos procesan en paralelo manteniendo ambos el estado. Permite la recuperación en ms.
Passive redundancy: uno procesa y actualiza el estado para que el secundario se pueda recuperar. Demora más que el hot spare
Spare: elemento fuera de servicio hasta que se produzca la falla. Es más para tolerancia al fallo por la demora en sincronización
Clustering



Cluster asimétrico



Manejo de excepciones: Una vez detectada la excepción debe manejarse. La forma puede ir desde uso de códigos de error al uso de clases de excepción con información para arreglar el problema

Rollback: cuando se detecta el defecto se vuelve a un estado correcto (checkpoint) Se combina con redundancia activa o pasiva.
Software upgrade: es cambiar código ejecutable en tiempo de ejecución, hacer parches para fixes, hitless service para nuevas funcionalidades.
Retry: asume faltas transitorias y que luego de algunos intentos se va a conseguir respuesta
Ignore faulty behavior: ignorar request de un elemento que sabemos que está defectuoso
Degradation: mantener funcionando elementos críticos. Se reduce la funcionalidad con el fin de que no falle todo el sistema
Reconfiguration: pasa responsabilidades de elementos defectuosos a otros que están funcionando manteniendo lo mejor posible el funcionamiento general
Reintroducción
Trata de reintroducir componentes rehabilitados al sistema
Shadow: poner a operar un elemento que falló y que se arregló en paralelo antes de ponerlo en producción real
State resynchronization – guardar el estado para que otros elemento pueda operar. ej warm y cold spare
Escalating restart- tratar de balancear el restart de forma de minimizar la afectación del servicio. Definir una estrategia de restart de forma de no interrumpir el servicio
Nonstop forwarding – dividir la funcionalidad en dos grupos: uno de control y otro de datos. tratar de seguir enviando datos hasta que se recupere la parte de control

Prevenir defectos
Intenta prevenir la ocurrencia de defectos
Removal from service: poner componentes fuera de servicio para reintroducirlos
Transactions: asegurar ACID en intercambios asíncronos
Predictive model: usar valores estadísticos para en un monitor, detectar variaciones y tomar acciones
Exception prevention – tomar precauciones para que las excepciones no ocurran. Smart pointers, bound checks, etc.
Increase comptence set – diseñar para manejar más casos excepcionales.

Ejercicios de aplicación de modificabilidad

1. En una evaluación de arquitectura uno de los interesados preguntó con curiosidad sobre la disponibilidad que el arquitecto había especificado (99.9%). La respuesta que recibió fue: "... es la disponibilidad que nos asegura el proveedor de servicios de la nube"

1.1 ¿Cuánto tiempo fuera de servicio admite la especificación de 99.9%?

   a. 3 días y 15,6 horas en un año
   b. 52 minutos y 34 segundos en un año
   c. 8 horas, 0 minutos y 46 segundos en un año
   d. 32 segundos en un año

1.2 ¿Cómo considera la respuesta del arquitecto? **Justifique** su respuesta

   a. Correcta
   b. Incorrecta

La respuesta es la c, es incorrecta la respuesta del arquitecto porque es lo que te da la nube, no la de su aplicación.

2. Dadas las siguientes decisiones que tomó un arquitecto de software durante el diseño de un sistema en el cual se le solicitó **una buena disponibilidad**. Identifique el nombre de las tácticas que seleccionó.

Táctica	Descripción de la decisión
	Incorporó un mecanismo en los componentes mediante el cual cada 20 segundos los componentes envían una señal indicando que están disponibles
	Diseño un componente que, mediante el uso de pulso, ping, e información de *timestamp* detecta y corrige problemas con componentes que fallan o que están respondiendo en forma degradada
	Solicitó a todos los desarrolladores que utilicen bloques try/ catch en el código e incorporen dentro de los mismos mecanismos que intenten recuperar al sistema
	Incorporó un mecanismo que permite marcar a un componente como fuera de servicio cuando el mismo no responde correctamente y que no afecta funciones críticas del sistema

- Heartbeat
- Monitor
- Los de excepciones. detectar, manejo, prevenir
- Removal from service

3.1 Pulso

3.2) Los programadores de una aplicación notaron que el manejo de logeo de los distintos tipos de excepciones en cada capa de la aplicación podría generalizarse en una capa lógica transversal a ellas, de modo que la lógica común al manejo de errores y su registro estuviera centralizada.

Abstraer servicios comunes

Interoperabilidad

https://onedrive.live.com/?authkey=%21AGyK0PaIrjBY7Xo&cid=5047D9112BE92C8A&id=5047D9112BE92C8A%21330050&parId=5047D9112BE92C8A%21330049&o=OneUp

Definición

Refiere al grado en que dos o más sistemas pueden intercambiar información significativa mediante interfaces bajo determinado contexto. Intercambiar info sintáctica y semánticamente, o sea que puedan hablar y entenderse.
La idea es intercambiar información vía interfaces. El diseño se puede basar en conocimiento previo o mecanismos para descubrirlos en tiempo de ejecución.

Escenario

Source of stimulus

. A system that initiates a request.

Stimulus

. A request to exchange information among systems.

Artifacts

. The systems that wish to interoperate.

Environment

. The systems that wish to interoperate are discovered at run-time or are known prior to runtime.

Response

. The request to interoperate results in the exchange of information. The information is understood by the receiving party both syntactically and semantically. Alternatively, the request is rejected and appropriate entities are notified. In either case, the request may be logged.

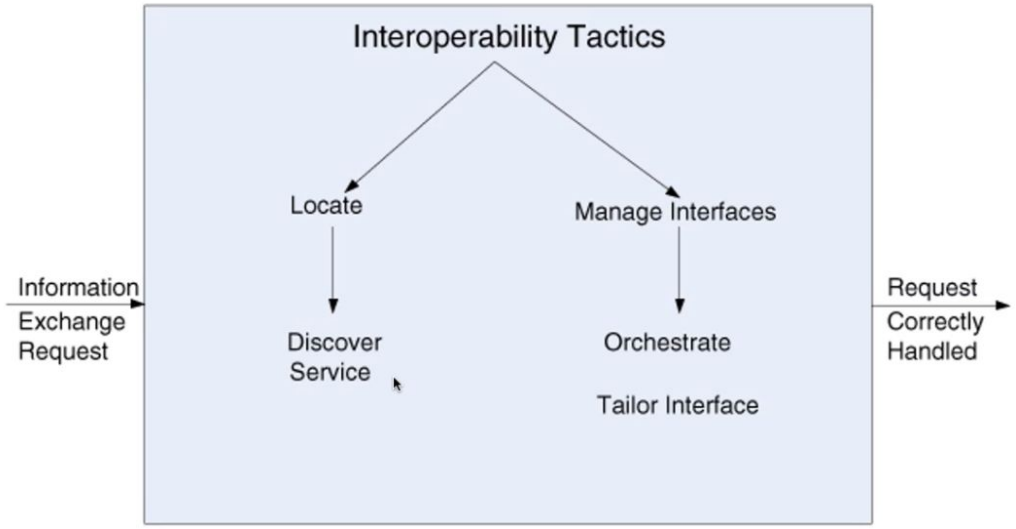
Response measure

. The percentage of information exchanges correctly processed or the percentage of information exchanges correctly rejected.

SOAP

Protocolo de especificación para información en XML que pueden usar aplicaciones para intercambiar info e interoperar. Casi siempre está con un conjunta de estándares middlewares SOA. Incluyen los siguientes estándares: An infrastructure for service composition, transacciones, service discovery y reliability.
REST es una arquitectura basada en cliente-servidor.

Tácticas

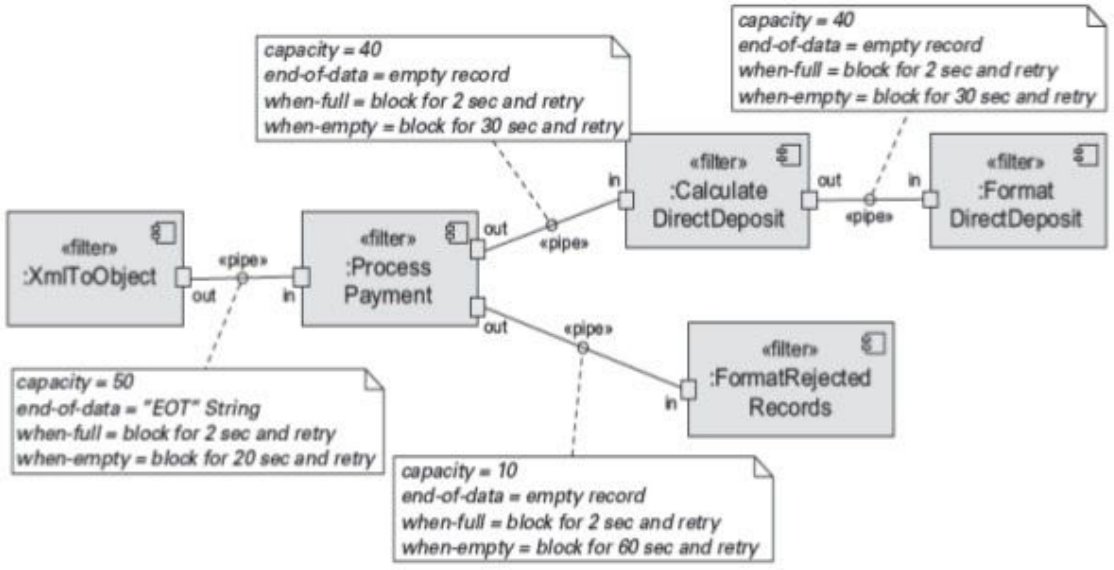


Locate: poner la ubicación para localizar los servicios. Servicios son un conjunto de capacidades accesibles mediante alguna forma de interfaz. No se debe confundir Orientación a servicios y micro servicios.
Discover services: para localizar los servicios y sus interfaces. UDDI tiene que información uno necesita para poder acceder a los servicios web.

Manejar las interfaces que expongo: invoco una interfaz y después tengo que llamar a otro, entonces alguien en el sistema se encarga de manejar la llamada.
Orquestar: alguien ante una llamada la **orquesta**.
Coreografía, alguien ante un llamada le pasa la llamada a otro, cada uno hace su parte. Ordenar la llamada de una api
Adaptar interfaces: agregar o remover capacidades. Las interfaces deben tener la posibilidad de exponerlas o cambiarlas, por ejemplo ponerle distintos mecanismos de seguridad, buffers, etc.

Patrones
Pipes and Filters

Contexto: los datos pueden llegar en distintos formatos
Problema: se tienen que aplicar una serie de ordenadas pero independientes operaciones
Solución: el patrón de interacción se caracteriza por transformaciones exitosas de flujos de datos. Los datos llegan en los puertos de entrada del filtro, se transforman, y se pasan por puertos de salida a través de un pipe al próximo filtro. Un solo filtro puede consumir data o producir a uno o varios puertos.
Ejemplo:



Overview: los datos son transformados de las entradas externas de un sistema a las salidas externas a través de una serie de transformaciones hechas por filtros y conectadas por pipes.
Elementos: filtro, se encarga de transformar la data leida en el puerto de entrada a data escrita en el puerto de salida. Pipe, es el conector que junta la data del puerto de salida del filtro a otro puerto de entrada de otro filtro. Preserva la secuencia de los elementos y no altera la información.
Relaciones: se asocia el output de los filtros con el input de los pipes y viceversa.
Restricciones:

- Los pipes conectan los puertos de output de los filtros con los puertos de input de los filtros.
- Los filtros conectados deben estar de acuerdo en el tipo de data que se pasa en el pipe.

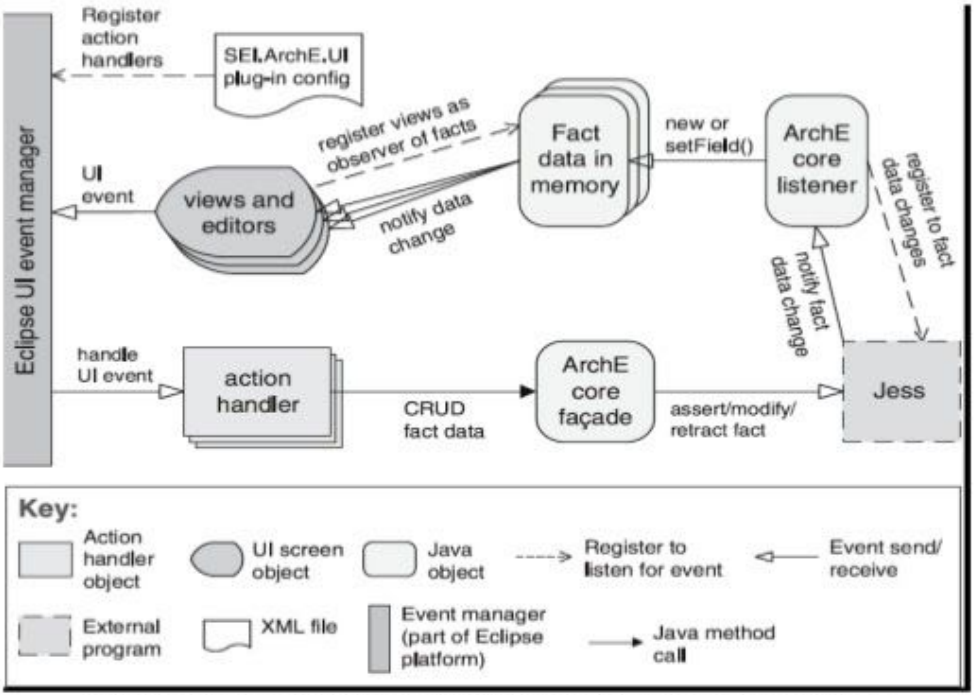
Publish and Subscribe

Contexto: hay un número de productores y consumidores que deben interactuar. El número de productores y consumidores no se puede determinar y tampoco información que comparten.

Problema: cómo podemos crear mecanismos de integración que tengan la habilidad de transmitir mensajes entre los productores y consumidores para que no sepan su identidad ni su existencia?

Solución: los componentes interactúan entre mensajes anunciados o eventos. Los componentes pueden suscribirse a una serie de eventos. Los componentes publisher ponen eventos en el bus avisandoles. El conector envía esos eventos a los suscriptores que han registrado interés en esos eventos.

Ejemplo:



Overview: los componentes publican y se suscriben a eventos. Cuando un evento es anunciado por un componente, el conector despacha un evento a todos los suscriptores registrados.

Elementos: cualquier C&C con al menos un puerto de publicación o suscripción. El conector entre el publish y suscribe que va a anunciar y escuchar roles para componentes que quieren publicar y suscribirse a eventos.

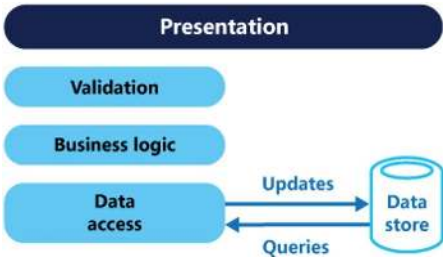
Relaciones: se preservan qué componentes anuncian sus eventos y cuales componentes están registrados para recibir eventos.

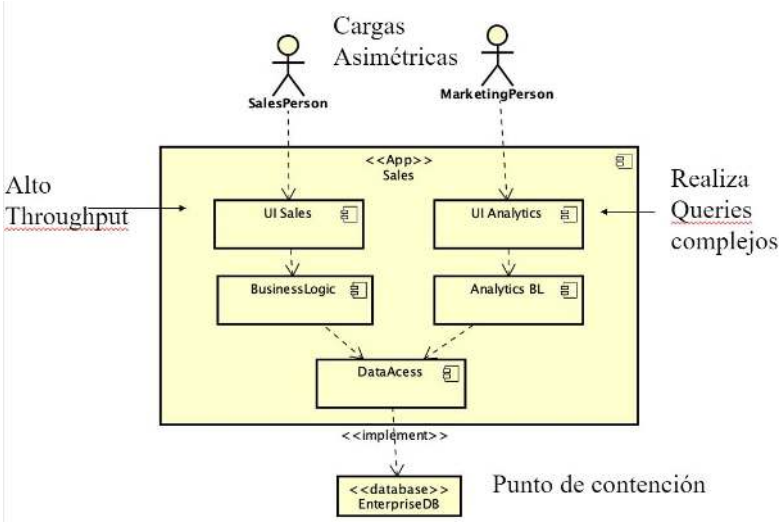
Restricciones: Todos los componentes están conectados a un evento que distribuye que puede ser visto como un bus (conector) o un componente. Los puertos de publicación tienen que anunciar roles y los puertos de suscripción tienen que escuchar roles.

Debilidades: aumenta la latencia y tiene un efecto negativo en la escalabilidad y predecibilidad del tiempo de envío de un mensaje. Hay menos control en el orden de los mensajes y el envío no está garantizado.

CQRS

https://martinfowler.com/bliki/CQRS.html
https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs
Command and Query Responsibility Segregation
Contexto y problema:
En sistemas que gestionan datos los comandos que actualizan los datos y la consultas se realizan sobre las mismas entidades en un repositorio.
Este esquema es útil cuando la lógica que se aplica a los datos es principalmente para hacer operaciones CRUD.

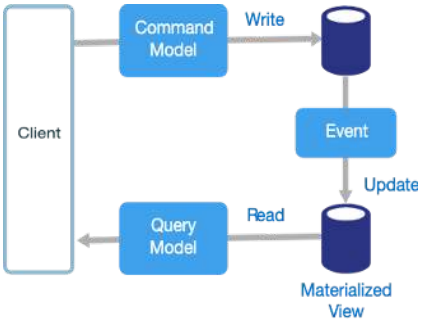


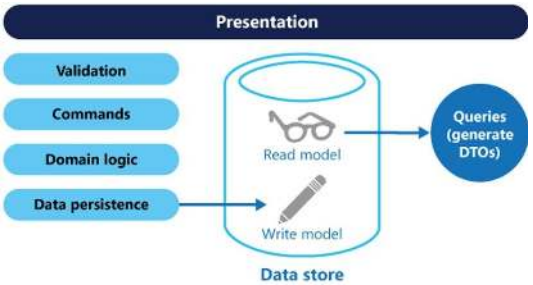


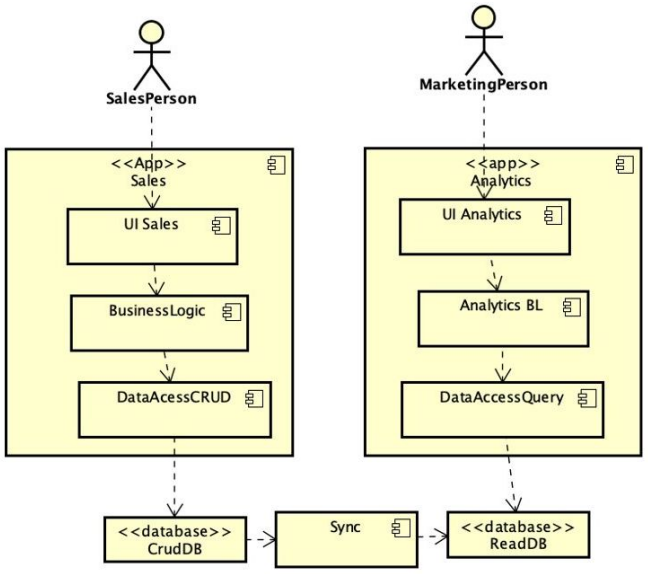
Contexto y problema:
En otros casos tiene las siguientes desventajas:

- Pueden existir diferencias entre los datos que se leen y escriben, es decir no se trabaja sobre los mismos atributos.
- Puede existir contención si se intenta actualizar el mismo recurso afectando la performance.
- La carga de operaciones de actualización y lectura pueden ser asimétricas y estar limitadas por usar el mismo modelo.
- Pueden existir problemas de seguridad al existir distintos permisos sobre operaciones de actualización y lectura.

Solución:
El patrón separa las operaciones de lectura (query) de las operaciones de actualización (commands) utilizando diferentes interfaces y distintos modelos.







Características:

- Permite escalar en forma independiente las lecturas de las escrituras
- Se necesitan optimizar los esquemas de datos de lectura y escritura
- Se necesita tener distintos permisos para operaciones de lectura y escritura
- Cuando por mantenibilidad es mejor tener modelos de lectura y escritura separados
- Cuando es necesario separar el desarrollo del equipo por tipo de operación
- Es un patrón difícil de diseñar, en general aplica a partes del sistemas
- La sincronización de ambos modelos puede ser difícil de lograr

Multi Tiers

https://onedrive.live.com/?authkey=%21AJ%5FnJJKPvRMMe5I&cid=5047D9112BE92C8A&id=5047D9112BE92C8A%21329778&parId=5047D9112BE92C8A%21329771&o=OneUp

Puede considerarse un patrón de la estructura de componentes y conectores o asignación.

Tier:

- A nivel de C&C es el agrupamiento de componentes con responsabilidad similar.
- A nivel de asignación, el ambiente de ejecución en el cual se ejecuta el software generalmente componentes. Este es el agrupamiento más usado.

Contexto:

En el despliegue distribuido, frecuentemente es necesario distribuir la infraestructura en distintos subconjuntos.

Problema:

Cómo podemos dividir el sistema en un número de estructuras de ejecución independientes de ejecución (software y hardware) conectados mediante distintos medios de comunicación.

Solución:

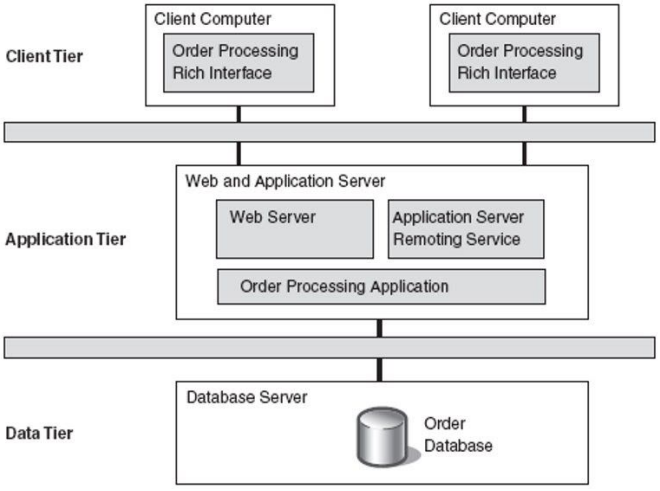
La estructura de ejecución de los sistemas se puede organizar en grupos lógico de componentes. Cada agrupamiento es un tier (nivel).

El criterio de agrupación del tier puede ser variado (componentes y responsabilidades), o en ambientes de ejecución o por el propósito en tiempo de ejecución.

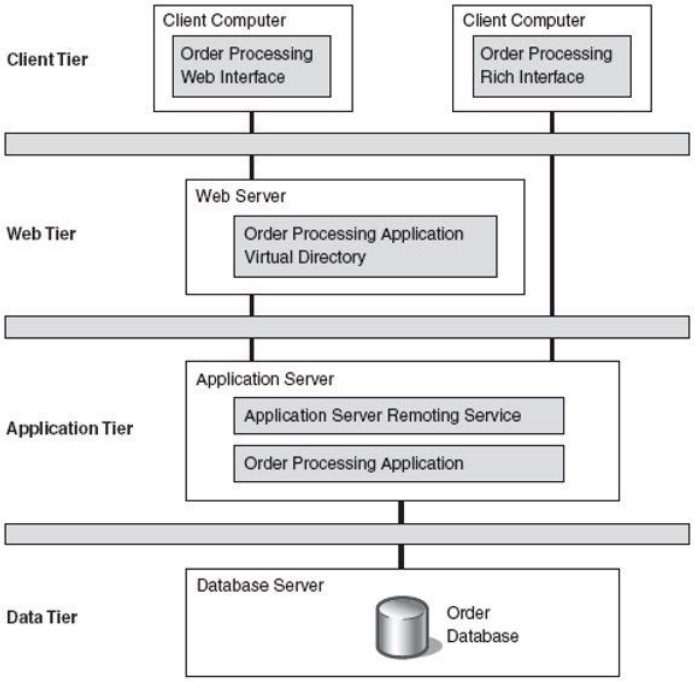
Restricciones:

- Los componentes normalmente se comunican con componentes en el mismo nivel o en adyacentes
- Puede describir el tipo de conector entre niveles.
- Puede ser muy caro de implementar
- Hace más fácil incorporar seguridad a nivel de tier, o mejorar la performance y disponibilidad.

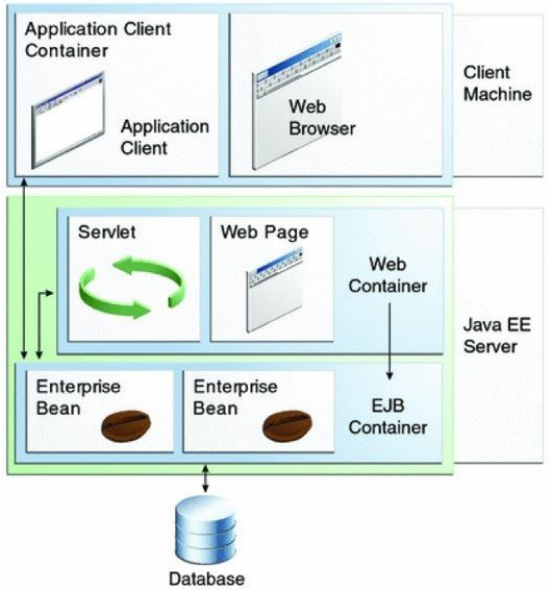
Tipo de arquitectura tres niveles:



Tipo de arquitectura cuatro niveles



Tipo de arquitectura JEE



Gatekeeper

https://docs.microsoft.com/en-us/azure/architecture/patterns/gatekeeper

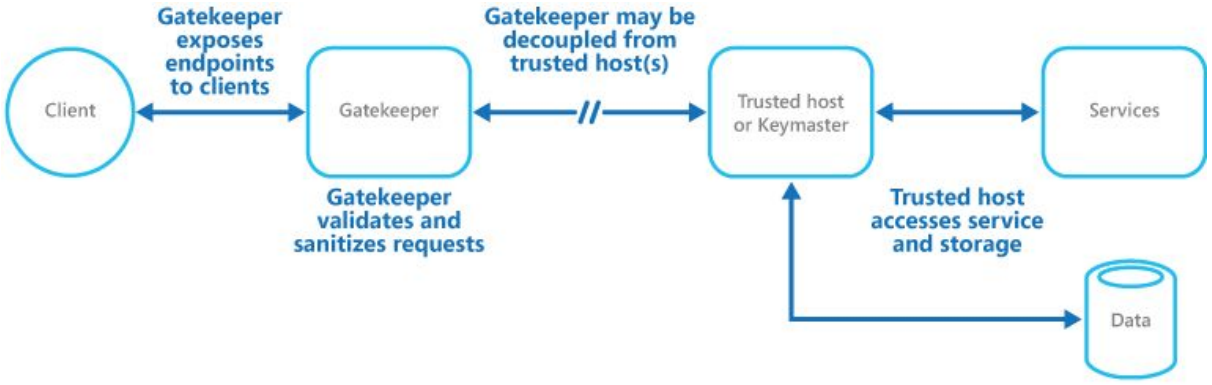
Problema y contexto

Las aplicaciones exponen sus funcionalidades a los clientes aceptando y procesando requests. En los escenarios alojados en la nube, las aplicaciones exponen los endpoints a los que se conectan los clientes y típicamente incluyen el código para manejar los requests de los clientes. El código sirve como autenticación y validación, para algunos o todos los requests y es probable acceder al storage y otros servicios a favor de los clientes.

Si un usuario malicioso puede comprometer el sistema y tener acceso al almacenamiento de la aplicación, los mecanismos de seguridad y los servicios y datos que accede están expuestos. Así el usuario malicioso puede tener acceso no restringido para información sensible y otros servicios.

Solución

Para minimizar el riesgo de que los clientes tengan acceso a información sensible, se desacopla las tareas que exponen los endpoints públicos del código que procesa requests y accede a la memoria. Se puede lograr usando una fachada o tarea dedicada que interactua con clientes y entrega el request (quizá con una interfaz desacoplada) a las tareas que manejan el request.



Puede usarse para proteger los datos o como una fachada para proteger las funciones de la aplicación.

- Validación controlada: el gatekeeper valida los requests y rechaza los que no se validan con los requisitos.
- Riesgo y exposición limitada: el gatekeeper no tiene acceso a todas las credenciales o claves que se usan por el host para acceder al almacenamiento y servicios. Si el gatekeeper está comprometido, no se le da acceso al atacante a las credenciales.
- Seguridad apropiada: el gatekeeper corre en un modo privilegiado limitado, mientras que el resto de la aplicación tiene un modo de confianza total requerido para acceder al almacenamiento y servicios. Si el gatekeeper está comprometido no puede acceder directamente a la los servicios y datos.

Actúa como firewall y permite examinar los request y decidir si pasarlos al host o no.

No debe guardar información de autenticación, eso lo hace el trusted host. Tampoco hace procesamientos de acceso a datos, tiene privilegios limitados. Podría tener listas blancas y negras para dejar pasar por ejemplo una IP. Nada relacionado a la seguridad o al negocio.

Hace validaciones de los tipos, del tamaño de los datos, de las consultas, el content type, que me llamen con verbos que no deberían usarse, el formato del token, estructura de Json, etc.

Debilidades:

- Agregar una capa extra para implementar el gatekeeper, es probable que tenga impacto en el performance, por los procesos nuevos y comunicación que necesita.
- La instancia de gatekeeper puede ser un solo punto de falla. Se puede minimizar el impacto.

Cuando usarlo:

- Las aplicaciones que manejan sensible información, exponen servicios que deben tener un alto grado de protección contra hackers, o tienen operaciones críticas que no deben ser interrumpidas.
- Aplicaciones distribuidas donde es necesario hacer requests de validación separados de las tareas principales, o centralizar la validación para simplificar el mantenimiento y administración.

Federated identity

https://docs.microsoft.com/en-us/azure/architecture/patterns/federated-identity

Contexto y problema

Los usuarios necesitan trabajar con varias aplicaciones proveídas por diferentes organizaciones. Deben tener que usar diferentes credenciales para cada uno.

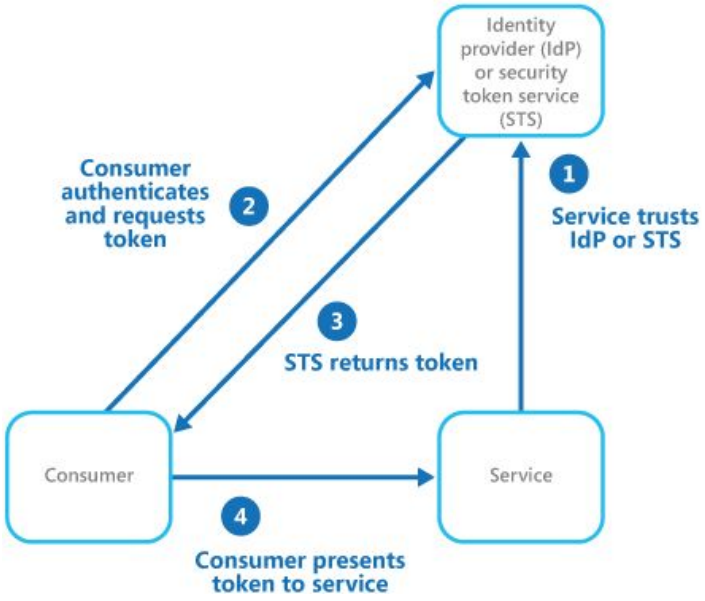
- Eso puede causar una mala experiencia de usuario, a veces se olvidan las credenciales de signin
- Expone las vulnerabilidades de seguridad, cuando un usuario se va de la empresa.
- Complica la administración de usuarios, para los administradores.

Los usuarios generalmente prefieren las mismas credenciales para todas las aplicaciones.

Solución

Implementar un mecanismo de autenticación que pueda usar una identidad federada. Separar la autenticación de usuario del código de aplicación y delegar la autenticación a un proveedor de confianza. Puede simplificar el desarrollo y permitir a los usuarios autenticarse usando más variedad de proveedores de identidad, minimizando la administración. Desacopla la autenticación de la autorización.

A este modelo a veces se le llama claims-based access control.



Applications and services authorize access to features and functionality based on the claims contained in the token. The service that requires authentication must trust the IdP. The client application contacts the IdP that performs the authentication. If the authentication is successful, the IdP returns a token containing the claims that identify the user to the STS (note that the IdP and STS can be the same service). The STS can transform and augment the claims in the token based on predefined rules, before returning it to the client. The client application can then pass this token to the service as proof of its identity.

Debilidades

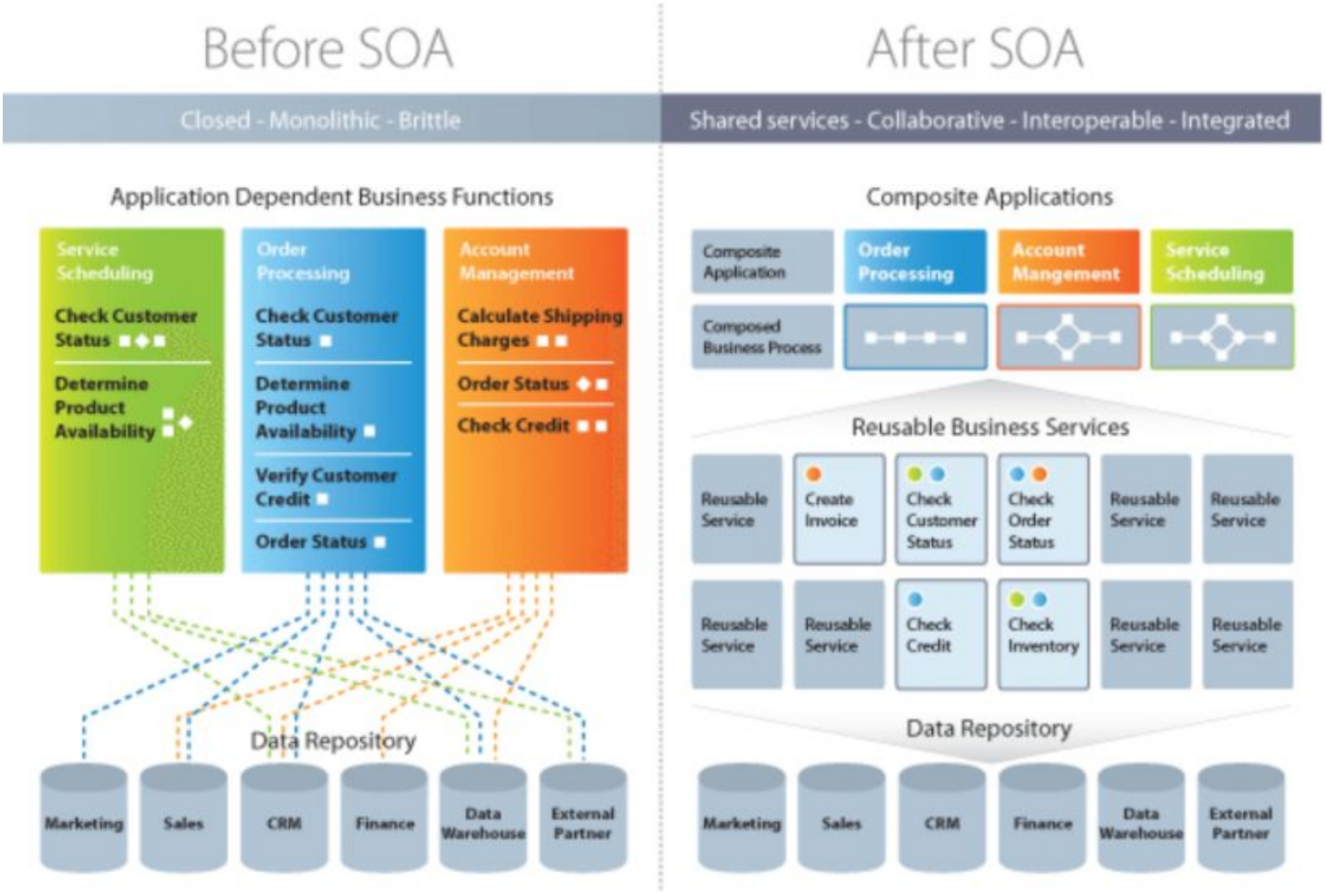
- La autenticación puede ser un solo punto de falla. Se puede minimizar el impacto.

Cuando usarlo

- Para que haya un solo sign-on en la empresa.
- Federated identity con varias aplicaciones.

SOA

Service-oriented architecture



¿Como podemos reusar las “funcionalidades” que implementan las actividades del negocio? Teniendo en cuenta que:

- Las funcionalidades pueden estar desarrolladas en distintos lenguajes
- Las funcionalidades se utilizan en diversos procesos del negocio
- Las procesos del negocio y las aplicaciones pueden estar distribuidos en internet y ser provistos por distintas unidades del negocio o proveedores

servicios

- Un servicio en interoperabilidad es un conjunto de capacidades accesibles mediante alguna forma de interfaz.

En SOA los servicios:

- Representan una actividad del negocio con un resultado específico.
- Son autocontenidos (se pueden desplegar solos).
- Los que lo usan no tienen idea de qué pasa adentro ni la tecnología que usan.
- Hay servicios que son compuestos por otros servicios, pueden depender de otros.

Contexto:

Un conjunto de servicios son ofrecidos y descritos por proveedores de servicios. Estos son consumidos por consumidores de servicios.

Los consumidores deben ser capaces de entender y utilizar estos servicios sin conocimiento detallado de su implementación

- Relacionado con la táctica de interoperabilidad Locate – Discover Service

Problema:

¿Como podemos soportar la interoperabilidad entre componentes distribuidos (la idea es que los servicios estén en cualquier lado de internet)

- ejecutando en distintas plataformas,
- desarrollados en distintos lenguajes de programación,
- provistos por distintas organizaciones
- distribuidos en distintos lados de internet?

¿Cómo podemos localizar los servicios y combinarlos en forma dinámica y coherente al tiempo que logramos performance, seguridad y disponibilidad?

Solución:

El patrón describe una colección de componentes distribuidos que proveen o consumen servicios.

Los servicios (en más detalle)

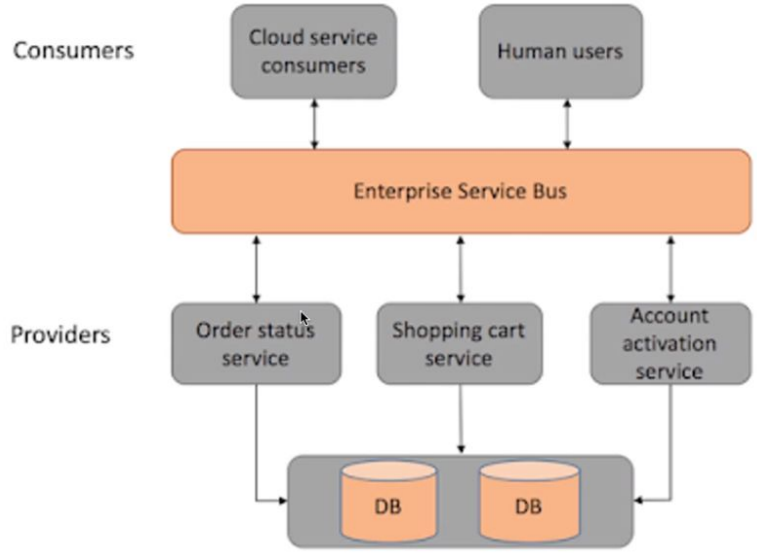
- Son independientes y auto-contenidos
- Pueden estar desarrollados en distintos lenguajes de programación y ejecutan en distintas plataformas
- Normalmente se despliegan de forma independiente
- Pueden estar compuestos por otros servicios
- Existen interfaces que permiten acceder a los servicios que un componente expone

Normalmente se usan como conectores como

- SOAP (Simple Object Access Protocol)
- REST (Representational State Transfer)
- Mensajería Asíncrona

Existen componentes especializados.

- Servicios de invocación (ESB)
    - Rutea mensajes entre consumidores y proveedores
    - Pueden realizar transformaciones
    - Hacen tareas de seguridad (quien llama al servicio de facturación tiene permiso de llamar al servicio de productos)
    - Cuando no se cuenta con un ESB los consumidores y proveedores se comunican punto a punto.

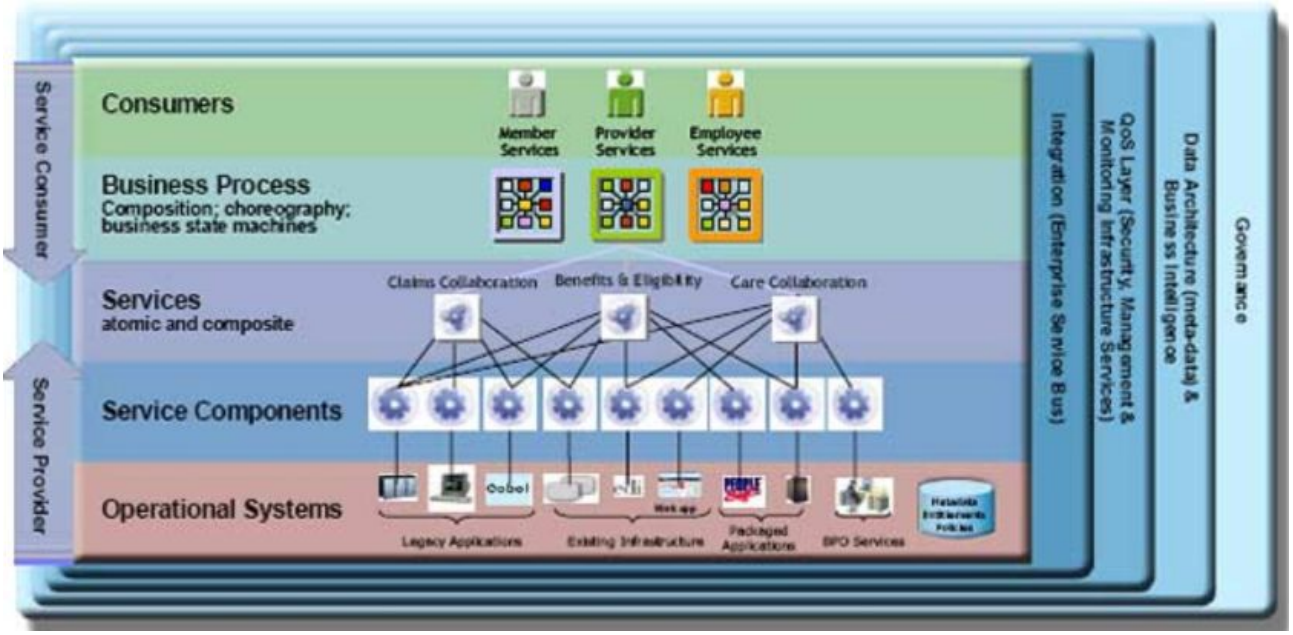


ESB y VETRO
Vetro:

- Validation
- Enrich (sacar más información)
- Transform (cambiar de un formato a otro)
- Rout & operate

Otros componentes de SOA:
Servicios de registro: permiten registrar los servicios en tiempo de ejecución y localizar o descubrir los servicios en tiempo de ejecución.
Servicios de orquestación: manejan la interacción entre consumidores y proveedores. (Dice en qué orden llaman a los servicios)

Consideraciones
En general los sistemas basados en este patrón son complejos de desarrollar. Se utilizan productos de software como MuleSoft, WSO2
En general tiene impacto negativo en la **performance**

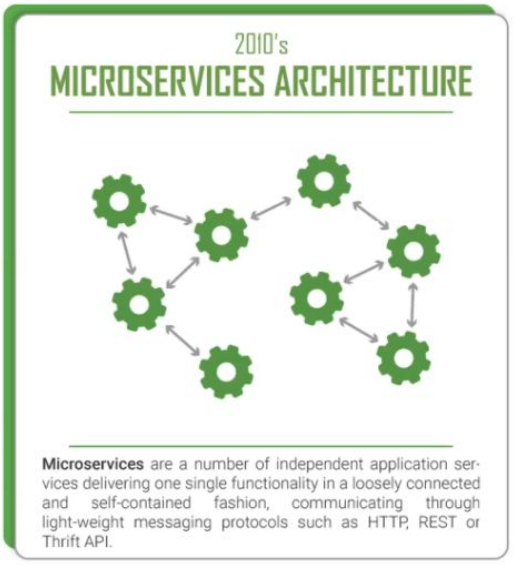


SOA VS Microservicios

Los microservicios intentan responder a cómo podemos romper las apps monolíticas grandes en cosas pequeñas

Responde a otros atributos de calidad.

Los cambios implican que yo tenga que desplegar toda la aplicación nuevamente, lo cual es muy complejo.



En microservicios rara vez comparten sus datos los distintos servicios.

- **SOA**
  - **Interoperabilidad**
    - Mediante un ESB
  - Modificabilidad
    - Favorece el desacoplar los servicios
  - Deployability
    - Los servicios NO son pequeños (son similares a los monolitos) y no es fácil desplegarlos independientemente
  - Escalabilidad
    - Los servicios no son pequeños ni cohesivos la replicación es mas costosa
  - Integración de sistemas legados

- **Micro Servicios**
  - Interoperabilidad
    - Conectores livianos (sin ESB), no es fácil realizar coreografía o orquestación
  - Modificabilidad
    - Favorece la cohesión y el bajo acoplamiento entre servicios (SRP)
  - **Deployability**
    - Los servicios son pequeños (Micro) y se despliegan en forma independiente en contenedores
  - Escalabilidad
    - Los servicios normalmente ejecutan en contenedores y son fáciles de replicar

Ninguno de los dos patrones es fácil de implementar.

Documentando arquitecturas

https://onedrive.live.com/?authkey=%21AEAgn4h08fgYs7A&cid=5047D9112BE92C8A&id=5047D9112BE92C8A%21330196&parId=5047D9112BE92C8A%21330178&o=OneUp

Crear la arquitectura no es suficiente, se debe comunicar de forma que los interesados puedan usarla para hacer sus trabajos.

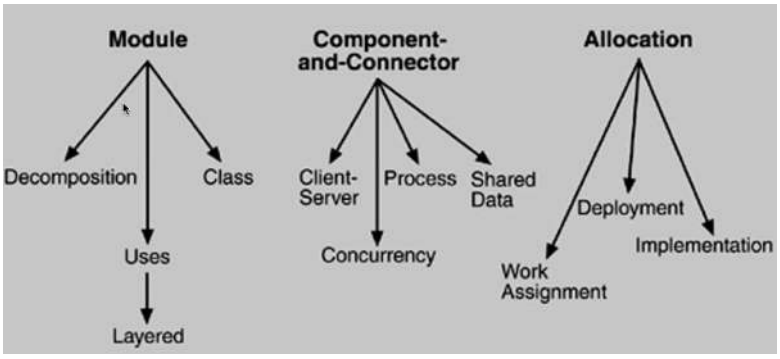
Se documenta porque es esencial para producir un producto de alta calidad con el menor re trabajo posible. En el software hay muchos interesados. También para capturar las principales decisiones que tomar y para razonar.

Hay que documentar las cosas que realmente cambian el sistema, y tienen efecto a largo plazo.

La arquitectura de un sistema es el conjunto de estructuras necesarias para razonar sobre el sistema, que comprenden elementos de software, la relación entre ellos y las propiedades de ambos (elementos y relaciones).

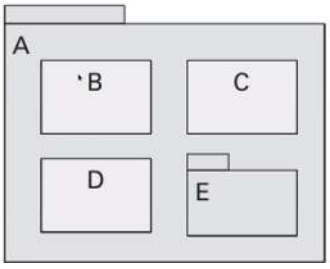
Vistas: es una representación de un conjunto de elementos del sistema y de sus relaciones. No incluyen todos los elementos, solo los más relevantes.

Tipos de vistas:

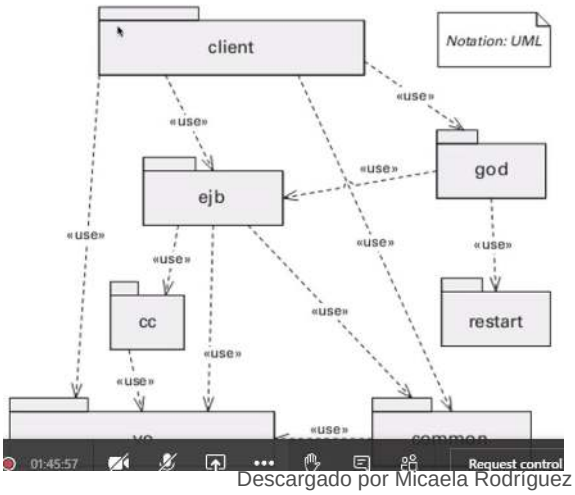


Módulo: en js el equivalente a un namespace es las carpetas.

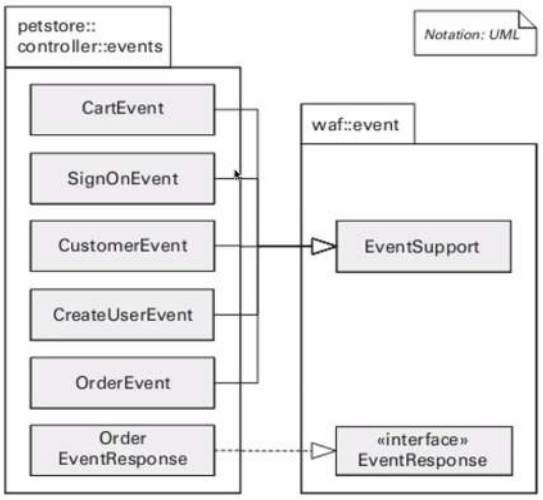
Descomposición: para mostrar cómo se organiza el código jerárquicamente. Por ejemplo por namespaces. Sirve para que los integrantes del equipo de desarrollo sepan en donde poner las cosas cuando están programando.



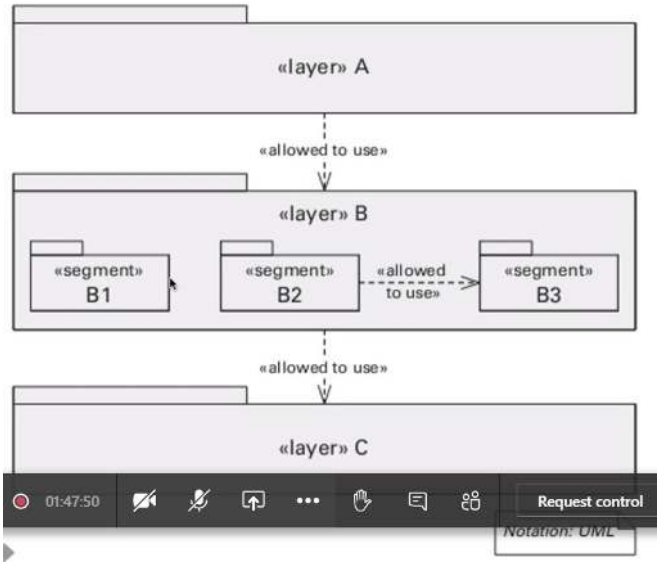
Usos: para evaluar impacto, cuando cambia algo, saber en donde cambia.



Clases:



Capas lógicas: agrupar por responsabilidad.



Modelos de datos:

Las vistas de módulos son útiles en la gestión de proyecto para la presupuestación y estimaciones, asignación de tareas y seguimiento. En la construcción proveen el plano del código fuente. Modificabilidad: facilita realizar análisis de impacto ante modificaciones. Comunicación: instrucción de nuevos desarrolladores.

Componentes y conectores:
Cliente-servidor:
Concurrencia:
Proceso:
Datos compartidos:

Asignación:
Asignación de trabajo:
Desarrollo:

Implementación:

Para hacer la arquitectura del sistema solo se necesitan el 20% de los requerimientos, se les llama requerimientos significativos.
La documentación es descriptiva y prescriptiva en función de la audiencia. Tiene que ser transparente y accesible para que se entienda, y es como un plano para construir el sistema.

Modelo Views and Beyond
Módulos, componentes y conectores y asignación.
Las vistas son una representación de un conjunto de elementos del sistemas y las relaciones. No incluyen todos los elementos, sólo los relevantes.

Módulos: conjunto de unidades cohesivas que representan unidades de implementación.

Vistas de módulos descomposición se pueden descubrir problemas de cohesión.
Vistas de módulos uso permiten ver problemas de modificabilidad, acoplamiento.
Vistas de módulos layers permiten también ver problemas de modificabilidad.

Vistas de componentes y conectores los procesos indican que pueden tener instancias separadas. Indican tácticas de performance.