Resumen de la 2da parte del curso de Arquitectura de Software

**Cap 5: Logrando Calidades (Tácticas)**

¿Qué es lo que imparte portabilidad a un diseño, alta eficiencia a otro, e integrabilidad a un tercero? El logro de estas cualidades se basa en decisiones fundamentales de diseño. Estas decisiones se denominan tácticas. Una táctica es una decisión de diseño que influye en el control de una respuesta de un atributo de calidad. Un conjunto de tácticas es denominado estrategia arquitectónica. Un patrón arquitectónico empaqueta tácticas también. Las tácticas pueden refinarse.

**Tácticas de Disponibilidad (Availability)**

Recordemos el vocabulario del capítulo anterior: una falla ocurre cuando el sistema no entrega un servicio que es consistente con su especificación; esta falla es observable por los usuarios del sistema. Una falta (o combinación de faltas) tiene el potencial de causar una falla. Recordemos también que la recuperación o reparación es un aspecto importante de la disponibilidad. Las tácticas que veremos a continuación evitarán que las faltas se conviertan en fallas, o al menos limitarán los efectos de las faltas haciendo posible una reparación.

**Detección de Faltas**

1. **Ping/Echo:** Un componente emite un ping a un componente que está en la mira y espera recibir un echo en un tiempo predeterminado. Esto puede usarse en un grupo de componentes que son mutuamente responsables de una misma tarea. También puede usarse por clientes para asegurar que un objeto del servidor y la ruta de comunicación al servidor están funcionando dentro de los límites de eficiencia esperados. Estos detectores de faltas pueden ser organizados en una jerarquía, donde un detector de bajo nivel emite un ping a los procesos de software con quienes comparte un procesador, y los detectores de alto nivel emiten pings hacia los de bajo nivel. Esto usa menos ancho de banda de comunicaciones que un detector de faltas remoto que emite pings a todos los procesos.
2. **Latido de Corazón (Heartbeat)**: En este caso, un componente emite un mensaje de latido de corazón periódicamente y otro componente espera ese mensaje. Si el latido de corazón falla, se asume que el componente que lo originó falló y un componente de corrección de faltas es notificado. El latido de corazón también puede cargar consigo datos. Por ejemplo, un cajero automático puede enviar periódicamente el registro de la última transacción a un servidor. Este mensaje no sólo actúa como un latido del corazón, sino que además lleva los datos a procesar.
3. **Excepciones:** Un método para reconocer faltas es encontrarse con una excepción, que aparece cuando se reconoce una de las clases de faltas mencionadas en el capítulo anterior. El manejador de excepciones típicamente se ejecuta dentro del mismo proceso que introdujo la excepción.

Las primeras dos tácticas operan entre procesos distintos, mientras que la táctica de excepciones opera en un solo proceso. El manejador de excepciones usualmente realizará una transformación semántica de la falta para poder procesarla.

**Recuperación de Faltas**

La recuperación de fallas consiste en la preparación para la recuperación y hacer que el sistema se repare. A continuación algunas tácticas de preparación y reparación.

1. **Votación (Voting):** Procesos que corren sobre procesadores redundantes toman entradas equivalentes y computan un simple valor de salida que es enviado a un votante. Si el votante detecta comportamiento desvíado de un procesador, lo hace fallar. El algoritmo de votación puede ser “la mayoría manda”, “componente preferido” o algún otro algoritmo. Este método es usado para corregir faltas en el funcionamiento de los algoritmos o el fracaso de un procesador y se utiliza a menudo en los sistemas de control. Si todos los procesadores utilizan los mismos algoritmos, la redundancia sólo detecta una falta del procesador y no una falta del algoritmo. Por lo tanto, si la consecuencia de un fallo es extrema, como la pérdida potencial de la vida, los componentes redundantes pueden ser diversos.
2. **Redundancia Activa (Hot Restart):** Todos los componentes redundantes responden a eventos en paralelo. En consecuencia, están todos en el mismo estado. La respuesta de sólo un componente es usada (usualmente el primero en responder), y el resto se descarta. Cuando ocurre una falta, el tiempo de inactividad de los sistemas que utilizan esta táctica es usualmente milisegundos, ya que la copia de seguridad es actual y el único tiempo de recuperación es el tiempo de conmutación. La redundancia activa es utilizada a menudo en la configuración de un cliente/servidor, como los sistemas gestores de bases de datos, donde respuestas rápidas son necesarias incluso al ocurrir una falta. En un sistema distribuido altamente disponible, la redundancia puede estar en las rutas de comunicacion. La sincronización se realiza al asegurar que todos los mensajes a un componente redundante son enviados a todos los componentes redundantes. Si existe la posibilidad de que la comunicación se pierda, un protocolo de transmisión confiable puede ser usado para la recuperación. Un protocolo de transmisión confiable requiere que todos los receptores acusen recibo con algún indicador de integridad (un checksum). Si el emisor no puede verificar que todos los receptores han recibido el mensaje, procederá a reenviar al mensaje a aquellos componentes que no hayan acusado recibo. El reenvío de mensajes no recibidos continúa hasta que el emisor marca al receptor como fuera de servicio.
3. **Redundancia Pasiva (Warm Restart / Dual Redundancy / Triple Redundancy):** Un componente (el primario) responde a un evento e informa al resto de los componentes (en espera) sobre actualizaciones de estado que deben realizar. Cuando ocurre una falta, el sistema debe asegurar primero que el estado de respaldo (backup) es suficientemente fresco antes de reanudar los servicios. Este enfoque se utiliza también en sistemas de control, donde a menudo las entradas vienen sobre canales de comunicación o desde sensores y deben ser cambiados al respaldo cuando ocurre una falla. Esta táctica depende de que los componentes en espera asuman el mando confiablemente. Forzando cambios periódicos incrementa la disponibilidad del sistema. Algunos sistemas de bases de datos fuerzan un cambio con el almacenamiento de cada nuevo dato. El nuevo dato es almacenado en una página sombra y la vieja página se vuelve respaldo para la recuperación. En este caso, el tiempo de inactividad usualmente puede ser limitado a segundos. La sincronización es responsabilidad del componente primario, que puede hacer uso de emisiones atómicas hacia los secundarios para garantizar la sincronización.
4. **Repuesto (Spare):** Una plataforma de computación de repuesto en espera está configurada para sustituir muchos componentes fallidos diferentes. Debe reiniciarse a la apropiada configuración de software y tener su estado incializado cuando ocurre una falla. Hacer un punto de control a un dispositivo persistente y registrar todos los cambios de estado en un dispositivo persistente permite establecer el estado adecuado del repuesto. Esto es a menudo usado como la estación de trabajo del cliente en espera, hacia donde el usuario puede moverse cuando ocurre una falla. El tiempo de inactividad de esta táctica usualmente es de minutos.

Existen tácticas de reparación que dependen de la reintroducción de un componente. Cuando un componente redundante falla, puede ser reintroducido luego de que haya sido corregido. Estas tácticas son operación sombra, resincronización de estado y rollback.

1. **Operación Sombra (Shadow Operation):** Un componente que previamente haya fallado puede ejecutarse en “modo sombra” por un corto tiempo para asegurar que imita el comportamiento del componente que funciona antes de que se lo reintegre al servicio.
2. **Resincronización de Estado (State Resynchronization):** Las tácticas de redundancia pasiva y activa requieren que el componente en restauración tenga su estado actualizado antes de que retorne al servicio. El enfoque de actualización dependerá de que el tiempo de inactividad pueda ser sostenido, el tamaño de la actualización, y el número de mensajes requeridos para la actualización. Un sólo mensaje conteniendo el estado es preferible, si es posible. Actualizaciones de estado incrementales, con períodos de servicio entre incrementos, llevan a un software complicado.
3. **Checkpoint/Rollback:** Un punto de control (checkpoint) es la grabación de un estado consistente creado periódicamente o en respuesta a eventos específicos. A veces un sistema falla de manera inusual, con un estado inconsistente detecable. En este caso, el sistema debe ser restaurado usando un checkpoint anterior con un estado consistente y un registro de las transacciones que ocurrieron desde que se tomó la instantánea.

**Prevención de Faltas**

1. **Retirada del Servicio (Removal from Service):** Esta táctica remueve un componente del sistema de la operación para someterse a algunas actividades para prevenir las fallas previstas. Un ejemplo es reiniciar un componente para prevenir que pérdidas de memoria causen una falla. Si la remoción es automática una estructura arquitectónica puede ser diseñada para apoyarla. Si es manual, el sistema debe estar diseñado para soportarlo.
2. **Transacciones (Transacciones):** Una transacción es la agrupación de varias etapas sucesivas de tal manera que todo el conjunto se puede deshacer de una vez. Las transacciones son usadas para prevenir que cualquier dato pueda ser afectado si un paso de un proceso falla y también para evitar colisiones entre varios procesos simultáneos que acceden a los mismos datos.
3. **Monitor de Proceso (Process Monitor):** Una vez que una falta en un proceso ha sido detectada, un monitor de proceso puede eliminar el proceso ineficiente y crear una nueva instancia del mismo, inicializado con un estado apropiado, al igual que en la táctica de repuesto.

**Tácticas de Modificabilidad**

Las tácticas que controlan modificabilidad tienen por meta controlar el tiempo y costo para implementar, probar y desplegar cambios. A continuación se organizan según sus metas: un conjunto tiene por meta reducir el número de módulos que son directamente afectados por un cambio (“localizar modificaciones”); otro conjunto tiene por meta limitar las modificaciones a los módulos localizados (“prevenir el efecto dominó”), donde existen módulos directamente afectados como también indirectamente afectados por el cambio; el tercer conjunto tiene como meta controlar despliege, tiempo y costo (“aplazar el tiempo de unión”).

**Localizar Modificaciones**

Aunque no hay necesariamente una precisa relación entre el número de módulos afectados por un conjunto de cambios y el costo de implementar esos cambios, restringir las modificaciones a un conjunto pequeño de módulos generalmente reducirá los costos. La meta de las tácticas en este conjunto es la de asignar responsabilidades a módulos durante el diseño tal que cambios previstos estarán limitados en alcance.

1. **Mantener Coherencia Semántica (Maintain Semantic Coherence):** La coherencia semántica refiere a las relaciones entre las responsabilidades de un módulo. La meta es asegurar que todas estas responsabilidades trabajan en conjunto sin dependencia excesiva de otros módulos. Esta meta se logra mediante la elección de responsabilidades que tienen coherencia semántica. Las métricas de acoplamiento y cohesión procuran medir la coherencia semántica, pero les falta el contexto del cambio. En cambio, la coherencia semántica debe medirse con un conjunto de cambios previstos. Una subtáctica es la de **abstraer servicios comunes**. Proveer servicios comunes a través de módulos especializados es usualmente visto como apoyo al reuso. Esto es cierto, pero también soporta la modificabilidad. Si los servicios comunes fueron abstraidos, las modificaciones que haya que hacerles se harán una sóla vez, en vez de hacerse en cada módulo donde se usa el servicio. Además, las modificaciones a los módulos que usan esos servicios no impactarán en otros usuarios. Esta táctica, entonces, soporta no sólo la localización de modificaciones sino también la prevención del efecto dominó. Ejemplos de abstracción de servicios comunes: el uso de marcos de trabajo de aplicaciones y el uso de otro software intermediario (middleware).
2. **Anticiparse a los Cambios Esperados (Anticipate Expected Changes):** Considerar el conjunto de cambios previstos provee una forma de evaluar una asignación particular de responsabilidades. La pregunta básica es: “Para cada cambio, ¿la disposición propuesta limita el conjunto de módulos que necesitan ser modificados para lograrlo?”. Una pregunta asociada es: “¿Cambios fundamentalmente diferentes afectan a los mismos módulos?”. ¿Como esto es diferente de la coherencia semántica? La asignación de responsabilidades basada en coherencia semántica asume que los cambios previstos serán coherentemente semánticos. La táctica de anticiparse a los cambios esperados no se preocupa por la coherencia de las responsabilidades de un módulo, sino más bien por la minimalización de los efectos de los cambios. En la realidad es difícil usar esta táctica por si sola, puesto que no es posible anticipar todos los cambios. Por esta misma razón, es usualmente usada en conjunción con la coherencia semántica.
3. **Generalizar el Módulo (Generalize the Module):** Hacer que un módulo sea más general le permite computar un rango más amplio de funciones basadas en entradas. La entrada puede ser considerada como la definición de un lenguaje para el módulo, que puede ser tan simple como hacer que constantes sean parámetros de entrada, o tan complicado como implementar el módulo como un intérprete y que los parámetros de entrada sean un programa en el lenguaje del intérprete. Cuánto más general es un módulo, es más probable que los cambios requeridos puedan hacerse ajustando el lenguaje de entrada y no a través de la modificación del módulo.
4. **Limitar Opciones Posibles (Limit Possible Options):** Las modificaciones, especialmente dentro de una línea de productos, pueden variar y así afectar a varios módulos. Restringir las opciones posibles reducirá el efecto de esas modificaciones. Por ejemplo, un punto de variación de una línea de productos puede permitir un cambio de procesador. Restringir los cambios de procesador a miembros de la misma familia limita las opciones posibles.

**Prevenir el Efecto Dominó**

El efecto dominó de una modificación es la necesidad de hacer cambios a módulos que no son directamente afectados por la modificación. Por ejemplo, si el módulo A tiene que cambiar para lograr una modificación particular, entonces el módulo B es cambiado sólo por el cambio en el módulo A. B tiene que modificarse porque depende, de alguna manera, de A.

Existen 8 tipos de dependencia entre módulos:

* Sintáxis de datos o de servicios
* Semántica de datos o de servicios
* Secuencia de datos o de control
* Identificación de una interfaz de A (cuando A tiene varias interfaces)
* Ubicación de A (en ejecución)
* Calidad de servicio/dato provisto por A
* Existencia de A
* Comportamiento del recurso de A

Hay que notar que ninguna de las tácticas necesariamente previenen el efecto dominó de cambios semánticos. Las primeras dos tácticas son relevantes a las interfaces de un módulo particular, y la última quiebra una cadena de dependencia.

1. **Ocultar Información (Hide Information):** El ocultamiento de información es la descomposición de las responsabilidades de una entidad (sistema o alguna descomposición del sistema) en piezas más pequeñas y eligiendo qué información hacerla privada y cuál pública. Las responsabilidades públicas están disponibles a través de interfaces especificadas. La meta es aislar cambios dentro de un módulo y prevenir que los cambios se propaguen a otros. Esta es la técnica más antigua para prevenir la propagación de cambios. Está fuertemente relacionada con la anticipación a los cambios previstos, puesto que usa esos cambios como base para la descomposición.
2. **Mantener Interfaces Existentes (Maintain Existing Interfaces):** Si B depende del nombre y firma de una interfaz de A, mantener esta interfaz y su sintáxis permite que B quede sin cambiar. Por supuesto que esta táctica no funcionaría si B tiene una dependencia semántica con A, ya que los cambios de los significados de los datos y servicios son difíciles de enmascarar. También es difícil enmascarar dependencias de calidad de datos o calidad de servicios, uso de recursos, o adueñamiento de recursos. La estabilidad de una interfaz también se puede lograr separando la interfaz de la implementación. Esto permite la creación de interfaces abstractas que enmascaran variaciones. Las variaciones pueden ser incorporadas dentro de las responsabilidades existentes, o pueden realizarse mediante la sustitución de la implementación de un módulo con otra. Patrones que implementan esta táctica incluyen agregar interfaces, agregar adapatador, y proporcionar un código auxiliar.
3. **Restringir Rutas de Comunicación (Restrict Communication Paths):** Restringir los módulos con los cuales un determinado módulo comparte información. Esto es, reducir el número de módulos que consumen datos producidos por el módulo en cuestión y reducir el número de módulos que producen datos que este módulo consume. Esto reducirá el efecto dominó ya que la producción y consumición de datos introduce dependencias que causan cascadas.
4. **Usar un Intermediario:** Si B tiene algún tipo de dependencia con A que no sea semántica, es posible insertar un intermediario entre A y B que maneje las actividades asociadas a la dependencia. Como antes, en el peor caso, un intermediario no puede compensar por cambios semánticos. Según los 8 tipos de dependencias:

* Sintáxis de Datos: Repositorios actúan como intermediarios entre el consumidor y el productor de datos. Los repositorios pueden convertir la sintáxis producida en A en la que es asumida por B. Algunos patrones de publicador/suscriptor (aquellos que tienen información fluyendo a través de un componente central) también lo pueden hacer. Los patrones MVC y PAC convierten datos de un formalismo a otro.
* Sintáxis de Servicios: Los patrones fachada, puente (bridge), mediador (mediator), estrategia (strategy), proxy, y fábrica (factory) proveen intermediarios que convierten la sintaxis de un servicio de una forma a otra.
* Identidad de una Interfaz de A: Un patrón broker puede usarse para enmascarar cambios en la identidad de una interfaz. Si B depende de la identidad de una interfaz de A y esa identidad cambia, agregando esa identidad al broker y haciendo que el broker haga la conexión hacia la nueva identidad de A, B puede mantenerse sin cambios.
* Ubicación de A (en ejecución): Un servidor de nombres permite cambiar la ubicación de A sin afectar a B. A es responsable de registrar su nueva ubicación con el servidor de nombres, y B recupera esa ubicación del servidor de nombres.
* Comportamiento de un recurso de A o recurso controlado por A: Un manejador de recursos es un intermediario que es responsable por la asignación de recursos. Ciertos manejadores de recursos pueden garantizar la satisfacción de todas las solicitudes dentro de ciertos límites. A, por supuesto, debe ceder el control del recuso para que el manejador de recursos lo tome.
* Existencia de A: El patrón fábrica (factory) tiene la habilidad de crear instancias a demanda, y de ese modo la dependencia de B de la existencia de A es satisfecha por las acciones de la fábrica.

**Aplazar el Tiempo de Unión**

Hasta ahora los escenarios de modificabilidad incluyen dos elementos que no son satisfechos por la reducción del número de módulos a cambiar y son: el tiempo para desplegar y permitir que quienes no sean desarrolladores puedan hacer cambios. Aplazar el tiempo de unión soporta estos dos escenarios con el costo de requerir infraestructura adicional para soportar la unión tardía. Varias tácticas están destinadas a tener impacto en tiempo de carga o de ejecución, como las que siguen:

1. **Registro en Ejecución (Runtime Registration):** Soporta opratividad plug and play con el costo de una sobrecarga adicional para gestionar el registro. La registración de publicador/suscriptor, por ejemplo, puede ser implementada tanto en tiempo de carga como en tiempo de ejecución.
2. **Archivos de Configuración:** Son destinados a establecer los parámetros al inicio.
3. **Polimorfismo:** Permite unión tardía de llamadas de métodos.
4. **Reemplazo de Componentes:** Permite unión en tiempo de carga.
5. **Adherencia a Protocolos Definidos:** Permite unión en tiempo de ejecución de procesos independientes.

**Tácticas de Eficiencia (Performance)**

La meta de las tácticas de eficiencia es la de generar una respuesta a un evento que llega al sistema dentro un límite de tiempo. El evento puede ser único o una cadena y es el detonante de una solicitud para realizar un cálculo. Puede ser la llegada de un mensaje, la expiración de un intervalo de tiempo, la detección de un cambio significativo del estado en el ambiente del sistema, u otro. El sistema procesa los eventos y genera una respuesta. Las tácticas de eficiencia controlan el tiempo dentro del cual se genera una respuesta. Latencia es el tiempo entre la llegada del evento y la generación de una respuesta.

Luego de que llega un evento, el sistema procesa el evento, o el procesamiento queda bloqueado por alguna razón. Esto nos lleva a los dos contribuyentes básicos del tiempo de respuesta: consumición de recursos y tiempo de bloqueo.

* Consumición de Recursos: Los recursos incluyen CPU, almacenes de datos, ancho de banda de la red de comunicación, y memoria, pero también pueden incluir entidades definidas por el sistema particular que se está diseñando. Los eventos pueden ser de varios tipos, y cada tipo pasa por una secuencia de procesamiento. Cada fase de esta secuencia contribuye a la letencia general del procesamiento del evento.
* Tiempo de Bloqueo: Un cálculo puede tener bloqueado el uso de un recurso porque hay una contienda sobre el recurso, porque el recurso no está disponible, o porque el cálculo necesita de resultados de otros cálculos que aún no están disponibles.

A continuación se verán tres tipos de categorías de tácticas: demanda de recursos, gestión de recursos y arbitraje de recursos.

**Demanda de Recursos**

Las cadenas de eventos son la fuente de la demanda de recursos. Dos características de la demanda son el tiempo entre eventos (cada cuanto hay una solicitud) y cuanto hay de consumición del recurso por cada solicitud.

Una táctica para reducir latencia es **reducir los recursos requeridos** para procesar una cadena de eventos. Se puede conseguir de una de las siguientes formas:

1. **Incrementar la Eficiencia Computacional:** Uno de los pasos en el procesamiento de un evento o mensaje es aplicar algún algoritmo. Mejorar los algoritmos usados en áreas críticas decrementa la latencia. Algunas veces un recurso puede ser negociado por otro. Esta táctica usualmente es aplicada al procesador pero también es efectiva cuando se aplica a otros recursos, como el disco.
2. **Reducir la Sobrecarga Computacional:** Si no hay una solicitud por un recurso, las necesidades de procesamiento se reducen. El uso de intermediarios (que son muy importantes para la modificabilidad) incrementa los recursos consumidos en el procesamiento de una cadena de eventos, por lo que quitándolos mejora la latencia. Esta es una clásica compensación de modificabilidad/eficiencia.

Otra táctica para reducir la latencia es reducir el número de eventos procesados. Esto se puede lograr de dos maneras:

1. **Administrar la Tasa de Eventos:** Si es posible reducir la frecuencia de muestreo en la que las variables ambientales son monitoreadas, la demanda puede ser reducida.
2. **Controlar la Frecuencia de Muestreo:** Si no hay ningún control sobre la llegada de los eventos generados externamente, las solicitudes en cola pueden ser muestreadas a una frecuencia más baja, posiblemente resultando en la decremento de peticiones.

Otras tácticas para reducir o gestionar la demanda involucran controlar el uso de recursos:

1. **Tiempos de Ejecución Limitados:** Poner un límite sobre cuanto tiempo de ejecución es usado para responder a un evento. Algunas veces tiene sentido, y otras no. Para algoritmos iterativos dependientes de datos, limitar la cantidad de iteraciones es una forma de hacerlo.
2. **Limitar los Tamaños de Colas:** Esto controla el número máximo de llegadas a la cola y consecuentemente los recursos usados para procesar las llegadas.

**Gestión de Recursos**

Aunque la demanda por recursos no pueda ser controlable, la gestión de estos recursos afecta el tiempo de respuesta. Algunas tácticas de gestión de recursos son:

1. **Introducir Concurrencia:** Si las solicitudes pueden ser procesadas en paralelo, el tiempo de bloqueo puede ser reducido. La concurrencia puede ser introducida al procesar diferentes cadenas de eventos en hilos diferentes o al crear hilos adicionales para procesar conjuntos distintos de actividades. Una vez que la concurrencia ha sido introducida, asignar apropiadamente los hilos a los recursos (balanceo de carga) es importante para maximizar la explotación de la concurrencia.
2. **Mantener Múltiples Copias de Datos o Cálculos:** Los clientes en el patrón cliente/servidor son réplicas del cálculo. El propósito de las réplicas es reducir la contienda que ocurriría si todos los cálculos se harían en un servidor central. El almacenamiento en caché es una táctica donde los datos son replicados en repositorios de diferentes velocidades o en repositorios separados para reducir la contienda. Dado que los datos que son almacenados en caché son usualmente una copia de datos que ya existen, mantener las copias consistentes y sincronizadas se convierte en una responsabilidad que el sistema debe asumir.
3. **Incrementar los Recursos Disponibles:** Procesadores más rápidos, procesadores adicionales, memoria adicional, y redes más rápidas, todos ellos tienen el potencial de reducir latencia. El costo es usualmente una consideración en la elección de recursos, pero incrementar los recursos es definitivamente una táctica para reducir latencia. Es una compensación de costo/eficiencia.

**Arbitraje de Recursos**

Siempre que haya una contienda por un recurso, el recurso debe ser programado. El objetivo del arquitecto es entender las características del uso de cada recurso y elegir una estrategia de programación que sea compatible con él.

Una política de progamación conceptualmente tiene dos partes: asignación de prioridades y despacho. Todas las políticas de programación asignan prioridades. En algunos casos la asignación es tan simple como FIFO (First In, First Out). En otros casos puede estar ligada a fin de plazo de la solicitud o su importancia semántica. Criterios de competencia para la programación incluyen uso óptimo del recurso, importancia de la solicitud, minimización del número de recursos usados, minimización de latencia, maximización de rendimiento, prevenir el hambre (starvation) para asegurar justicia, y otros. El arquitecto debe tener en cuenta estos criterios de posible conflicto y el efecto que tiene la táctica elegida al encontrarse con ellos.

Una cadena de eventos de alta prioridad puede ser despachada sólo si el recurso al cuál está asignada está disponible. A veces esto depende de adelantarse al usuario actual del recurso. Las opciones de adelanto posibles son las siguientes: puede ocurrir en cualquier momento, puede ocurrir sólo en puntos de adelanto específicos, procesos en ejecución no pueden ser adelantados. Algunas políticas de programación comunes son:

1. **FIFO (First In/First Out):** Las colas FIFO tratan a todas las solicitudes por recursos como iguales y las satisfacen en su debido turno. Una posibilidad con FIFO es que una solicitud quede tracada detrás de otra que está demorando mucho en generar una respuesta. Mientras todas las solicitudes sean realmente iguales, esto no es un probema, pero si algunas solicitudes tienen mayor prioridad que otras, es problemático.
2. **Programación de Prioridad Fija:** Esta política asigna a cada fuente de solicitudes de recursos una prioridad particular y asigna los recursos en ese orden de prioridades. Esta estrategia asegura un mejor servicio para solicitudes de mayor prioridad, pero admite la posibilidad de que una solicitud de baja prioridad aunque importante quede trancada detrás de una serie de solicitudes de mayor prioridad. Tres estrategias de priorización son:

* **Importancia Semántica:** Cada cadena (stream) es asignada una prioridad estáticamente de acuerdo a alguna característica de dominio de la tarea que la generó. Este tipo de programación es usado en sistemas mainframe donde la característica de dominio es el tiempo de inicio de la tarea.
* **Plazo Monotónico:** Es una asignación estática de prioridad que asigna mayor prioridad a aquellas cadenas con menor plazo. Esta política de programación es usada cuando cadenas de diferentes prioridades con plazos de tiempo real deben ser programadas.
* **Tasa Monotónica:** Es una asignación estática de prioridad para cadenas periódicas que asigna mayor prioridad a aquellas cadenas con períodos más cortos. Esta polítca de programación es un caso especial del plazo monotónico pero es más conocida y más probable de ser soportada por el sistema operativo.

1. **Programación Dinámica de Prioridad:**

* **Round Robin:** Es una estrategia de programación que ordena las solicitudes y luego, cada vez que existe la posibilidad de asignación, asigna el recurso a la próxima solicitud en ese orden. Una forma especial de Round Robin es un ejecutivo cíclico donde las posibilidades de asignación son fijas en ciertos intervalos de tiempo.
* **Plazo Más Cercano Primero:** Asigna prioridades en base a las solicitudes pendientes con el plazo más cercano.

1. **Programación Estática:** Una programación ejecutiva cíclica es una estrategia de programación donde los puntos de adelanto y la secuencia de asignaciones al recurso son determinados offline.

**Tácticas de Seguridad (Security)**

Las tácticas para lograr la seguridad pueden ser divididas entre aqullas preocupadas por resistir ataques, aquellas preocupadas por detectar ataques y aquellas preocupadas por la recuperación de los ataques. Las tres categorías son importantes.

**Resistir Ataques**

Identificamos el no repudio, la confidencialidad, la integridad y el aseguramiento como objetivos en la caracterización de seguridad. Las siguientes tácticas pueden usarse en combinación para lograr estos objetivos:

1. **Autenticar Usuarios:** Autenticar significa asegurar que el usuario o la computadora remota es quien dice ser. Contraseñas, contraseñas de un sólo uso, certificados digitales e identificaciones biométricas proveen autenticación.
2. **Autorizar Usuarios:** Autorizar es asegurarse que el usuario autenticado tiene los derechos de acceso y modificación de datos o servicios. Esto es usualmente gestionado cuando se proveen algunos patrones de control de acceso dentro del sistema. El contrl de acceso puede ser por usuario o por la clase del usuario. Las clases de usuario pueden ser definidas por grupos de usuarios, por roles de usuarios, o por listas de individuos.
3. **Mantener la Confidencialidad de los Datos:** Los datos deben estar protegidos del acceso no autorizado. La confidencialidad es usualmente lograda al aplicar cierta forma de encriptación a los datos y a los enlaces de comunicación. La encriptación provee protección extra a datos que son mantenidos persistentemente que va más allá de lo disponible por la autorización. Los enlaces de comunicación, por otro lado, típicamente no tienen controles de autorización. La encriptación es la única protección para pasar datos a través de enlaces de comunicación de acceso público. El enlace puede ser implementado a través de una red virtual privada (VPN) o a través de una capa segura de sockets (SSL) para un enlace basado en la web. La encriptación puede ser simétrica o asimétrica.
4. **Mantener Integridad:** Los datos deben ser enviados como se pretende. Pueden tener información redundante codificada en ellos, como checksums o resultados hash, que pueden ser encriptados con los datos propiamente dichos o por separado.
5. **Limitar la Exposición:** Los ataques típicamente dependen de explotar una sóla debilidad para atacar todos los datos y servicios de un anfitrión. El arquitecto puede diseñar el despliegue de los servicios a varios anfitriones, de modo que una cantidad limitada de servicios son disponibles en cada anfitrión.
6. **Limitar el Acceso:** Los contrafuegos (firewalls) restringen acceso basandose en la fuente del mensaje o el puerto de destino. Los mensajes que provienen de fuentes desconocidas pueden ser una forma de ataque. No siempre es posible limitar el acceso a fuentes conocidas. Un sitio web público, por ejemplo, puede esperar recibir solicitudes de fuentes desconocidas. Una configuración utilizada en este caso es la llamada Zona Desmilitarizada (DMZ). Es usada cuando el acceso debe ser provisto para los servicios de Internet pero no a una red privada. Se encuentra entre Internet y el cortafuegos delante de la red interna. La DMZ contiene dispositivos que esperan recibir mensajes de fuentes arbitrarias tal como servicios web, e-mail, y servioso de nombre de dominio.

**Detectar Ataques**

La detección de una ataque es usualmente a través de un sistema de detección de intrusiones. Tales sistemas comparan patrones de tráfico de redes contra una base de datos. En caso de detecto de desuso, el patrón de tráfico es comparado contra patrones históricos de ataques conocidos. En el caso de detección de anomalía, el patrón de tráfico es comparado con una línea base histórica de si mismo. Frencuentemente, los paquetes deben ser filtrados para poder realizar las comparaciones. El filtrado puede ser en base a protocolo, banderas TCP, tamaños de carga útil, dirección de destino o remitente, o número de puerto.

Los detectores de intrusión deben tener algún tipo de sensor para detectar ataques, manejadores para realizar fusión de sensores, bases de datos para gy uardar eventos para posterior análisis, herramientas para reportes y análisis offline, y una consola de control para que el analista pueda modificar acciones de detección de intrusión.

**Recuperación de Ataques**

Las tácticas involucradas en la recuperación de un ataque pueden ser divididas entre aquellas preocupadas por la restauración del estado y aquellas preocupadas por la identificación del atacante (tanto para objetivos preventivos como punitivos).

Las tácticas usadas en la restauración del sistema o de los datos a su estado correcto se solapan con las de disponibilidad, ya que ambas tienen que ver con la recuperación de un estado consistente. Una diferencia es la atención especial que se le da a mantener copias redundantes de datos administrativos del sistema como contraseñas, listas de control de acceso, servicios de nombres de dominio, y datos de perfil de usuario.

La táctica para identificar un atacante es la de **mantener una pista de auditoría** . Una pista de auditoría (audit trail) es una copia de cada transacción aplicada a los datos en el sistema junto con información identificatoria. La información de auditoría puede usarse para hacer un seguimiento de las acciones del atacante, soportar el no repudio (porque provee evidencia que una solicitud fue hecha), y soportar la recuperación del sistema. Las pistas de auditoría son a menudo blancos de ataque y por lo tanto deben mantenerse de forma fiable.

**Tácticas de Testeabilidad (Testeability)**

El objetivo de las tácticas de testeabilidad es permitir un testeo más fácil cuando se haya completado un incremento del desarrollo del software. Las técnicas arquitectónicas para mejorar la testeabilidad del software no han recibido tanta atención como otras áreas más maduras, pero como el testeo consume tan alto porcentaje del costo de desarrollo de un sistema, cualquier cosa que el arquitecto pueda hacer para reducir este costo será beneficioso.

Aunque se han incluido las revisiones del diseño como una técnica de testeo, en este caso nos interesa solamente el testeo de un sistema en ejecución. El objetivo de un régimen de testeo es encontrar faltas. Esto requiere que se provea de entradas al software y se capturen las salidas.

Ejecutar los procedimientos de prueba requiere de algún software para proveer las entradas que serán probadas y capturar las salidas. Esto se llama arnés de prueba.

A continuación, se discuten dos categorías de tácticas para el testeo: proveer entradas y capturar salidas, y monitoreo interno.

**Input/Output**

Existen tres tácticas para gestionar las entradas y las salidas para el testeo:

1. **Grabar/Reproducir (Record/Playback):** Refiere tanto a la captura de información que pasa a través de una interfaz como el uso de esa información como entrada en el arnés de prueba. La información que atraviesa una interfaz en una operación normal se guarda en algún repositorio y representa salida de un componente y entrada para otro. Grabar esta información permite que la entrada de un componente sea generada y que la salida de la prueba sea guardada para luego compararla.
2. **Separar Interfaz de la Implementación:** Separar la interfaz de la implementación permite la sustitución de implementaciones para objetivos varios de testeo. Tener código auxiliar para implementaciones permite que el resto del sistema pueda ser testeado en la ausencia del código sustituido. Sustituir un componente especializado permite que el componente sustituido actúe como un arnés de prueba para el resto del sistema.
3. **Especializar Rutas de Acceso/Interfaces:** Tener interfaces especializadas para realizar pruebas permite la captura o la especificación de valores de variables para un componente a través de un arnés de prueba así como también independientemente de su ejecución normal. Por ejemplo, metadatos pueden ser disponibles a través de una interfaz especializada que un arnés de prueba usaría para ejecutar sus actividades. Rutas de acceso e interfaces especializadas deben ser mantenidas separadas de las rutas de acceso e interfaces requeridas para la funcionalidad requerida. Tener una jerarquía de interfaces de prueba en la arquitectura significa que casos de prueba pueden ser aplicados en cualquier nivel de la arquitectura y que la funcionalidad de prueba está en su lugar para observar la respuesta.

**Monitoreo Interno**

Un componente puede implementar tácticas basado en su estado interno para apoyar el proceso de prueba.

* **Monitores Incorporados:** El componente puede mantener estado, carga de eficiencia, capacidad, seguridad, u otra información accesible a través de una interfaz. Esta interfaz puede ser una interfaz permanente del componente o puede ser introducida termporalmente mendiante una técnica de instrumentación tal como la programación orientada a aspectos o macros de procesador. Una técnica común es grabar eventos cuando los estados de monitoreo han sido activados. Los estados de monitoreo pueden incrementar el esfuerzo de prueba ya que las pruebas pueden tener que ser repetidas con el monitor apagado. La visibilidad incrementada de las actividades del componente usualmente tienen más peso que el costo de las pruebas adicionales.

**Tácticas de Usabilidad (Usability)**

La usabilidad trata de cuán fácil es para el usuario lograr determinada tarea y el tipo de soporte que el sistema brinda al usuario. Dos tipos de tácticas soportan usabilidad, cada uno está dirigido a dos tipos de usuarios. La primera categoría, en ejecución, incluye aquellas tácticas que brindan soporte al usuario durante el tiempo de ejecución. La segunda categoría está basada en la naturaleza iterativa del diseño de la interfaz de usuario y brinda soporte al desarrollador de interfaces durante mientras se realiza el diseño. Está fuertemente relacionado con las tácticas de modificabilidad previamente presentadas.

**Tácticas en Tiempo de Ejecución**

Una vez que el sistema está en ejecución, la usabilidad es mejorada brindando al usuario retroalimentación (feedback) sobre qué es lo que está haciendo el sistema y brindándole al usaurio la habilidad emitir comandos basados en usabilidad. Por ejemplo, cancelar, deshacer, agregar, y mostrar múltiples vistas brindan soporte al usuario en corrección de errores u operaciones más eficientes.

Cuando el usuario toma la inciativa, el arquitecto diseña una respuesta como para cualquier otra pieza de funcionalidad. El arquitecto debe enumerar las responsabilidades del sistema para responder al comando del usuario.

Cuando el sistema toma la inciativa, debe estar respaldado por algún tipo de información – un modelo – sobre el usuario, la tarea que está siendo realizada, o mismo sobre el estado del sistema. Cada modelo requiere varios tipos de entrada para lograr su iniciativa. Las tácticas de iniciativa del sistema son aquellas que identifican los modelos que el sistema usa para predecir su propio comportamiento o la intención del usuario. Encapsular esta información habilitará al arquitecto a crear y modificar esos modelos. La creación y modificación pueden ser dinámicos basados en el comportamiento histórico del usuario u offline durante el desarrollo.

* **Mantener un Modelo de la Tarea:** El modelo de la tarea es usado para determinar contexto para que el sistema pueda tener una idea de lo que el usuario está intentando hacer y proveer varios tipos de asistencia.
* **Mantener un Modelo del Usuario:** Este modelo determina el conocimiento que tiene el usuario del sistema, el comportamiento del usuario en términos de tiempos de respuesta esperados, y otros aspectos específicos a un usuario o clase de usuarios.
* **Mantener un Modelo del Sistema:** Este modelo determina el comportamiento esperado del sistema para que una retroalimentación adecuada pueda ser brindada al usuario. El modelo del sistema predice items tales como el tiempo necesario para completar la actividad actual.

**Tácticas en Tiempo de Diseño**

Las interfaces de usuario típicamente son revisadas frecuentemente durante el proceso de pruebas (testing). El ingeniero de usabilidad les dará a los desarolladores revisiones al diseño actual de la interfaz de usuario y los desarrolladores las implementarán. Esto nos lleva a una táctica que es un refinamiento de la táctica de coherencia semántica de modificabilidad:

* **Separar la Interfaz de Usuario del Resto de la Aplicación:** Localizando cambios esperados es la razón fundamental de la coherencia semántica. Dado que está previsto que la interfaz de usuario cambie frecuentemente tanto durante el desarrollo como después del despliegue, mantener el código de la interfaz de usuario por separado localizará sus cambios. Los patrones de arquitectura de software desarrollados para implementar esta táctica y brindar soporte a la modificación de la interfaz de usuario son: Model-View-Controller (MVC), Presentation-Abstraction-Control (PAC), Seeheim, y Arch/Slinky.

**Estilos y Patrones Arquitectónicos**

Los estilos o patrones arquitectónicos son un conjunto de principios que dan forma a una aplicación y que proveen un marco abstracto a una familia de sistemas. Están compuestos por: tipos de elementos, organización topológica de los elementos que determinan las interrelaciones, restricciones semánticas, y mecanismos de interacción.

Las tácticas representan los cimientos o bloques fundamentales del diseño a partir de los cuales se construyen los patrones o estilos arquitectónicos.

Los patrones arquitectónicos proveen un lenguaje común, independiente de la tecnología. Y además se enfocan en áreas particulares del sistema.

**Estilo o Patrón de Capas Lógicas (Layers)**

El patrón arquitectónico de capas ayuda a estructurar aplicaciones que pueden ser descompuestas en grupos de subtareas, de modo que cada uno de estos grupos se encueentes, a través de interfaces bien definidas.

* **Usos conocidos:** Modelo OSI (ISO), máquinas virtuales (JVM), sistemas de información, sistemas operativos.
* **Beneficios:** 
  + Soporta reuso: Las capas inferiores no dependen de las superiores, de modo que estas pueden ser reutilizadas en otros escenarios y permiten ser intercambiadas.
  + Soporte para la estandarización: Niveles de abstracción claramente definidos permiten el desarrollo de tareas e interfaces estandarizadas.
  + Eficiencia y disponibilidad: Las capas pueden ser distribuidas físicamente mejorando escalabilidad, tolerancia a fallos y eficiencia.
  + Testeabilidad: Interfaces bien definidas e intercambiabilidad soportan la testeabilidad del sistema.
* **Desventajas:**
  + Cambios en cascada: Cambios de comportamiento en una capa puede ser un problema severo.
  + Eficiencia: Datos, mensajes y excepciones deben ser transferidos entre un número de capas intermedias.
  + Dificultad para determinar la granularidad de las capas: Muy pocas capas no explotan correctamente el patrón. Excesivas capas agregan complejidad e impactan en la eficiencia.

**Estilo o Patrón en Capas Físicas (Tiers)**

Es un estilo de despliegue que describe la separación de funcionalidades en segmentos similares al patrón de capas lógicas, pero cada uno de estos segmentos ubicados en nodos físicos separados. De este modo, define el despliegue de las capas lógicas de una aplicación. Soporta escalabilidad, disponibilidad y optimización de recursos.

* **Beneficios:** 
  + Modificabilidad: Las capas físicas son independientes, de modo que pueden variar independientemente sin afectar otras capas.
  + Escalabilidad: Capas lógicas se despliegan en capas físicas, de modo que existe buen soporte para escalabilidad horizontal.
  + Flexibilidad: Cada capa física puede ser gestionada independientemente.
  + Disponibilidad: A partir de la escalabilidad horizontal.
* **Desventajas:**
  + Seguridad: Aumenta la cantidad de objetivos de ataques.
  + Eficiencia: Comunicación remota.
  + Complejidad: En la infraestructura, configuración y gestión.
  + Costos.

**Two Tier (2 Capas)**

Client Tier: Contiene la lógica de presentación (interfaz de usuario) y la lógica de negocio o dominio.

* Database Tier: Representa otra máquina que hospeda la base de datos de la aplicación.

**Three Tier (3 Capas)**

* Client Tier: Puede contener la lógica de presentación (interfaz de usuario) o puede ser un cliente fino (thin client), como por ejemplo, un browser.
* Web/App Tier: Contiene la lógica de presentación (web) y/o la lógica de dominio de la aplicación en un servidor independiente.
* Database Tier: Representa otra máquina que hospeda la base de datos de la aplicación.

**N-Tier (4 Capas o Más)**

* Client Tier: Representa el browser (thin client)
* Web Tier: Contiene la lógica de presentación (web) en un servidor independiente.
* Business Logic Tier: Contiene la lógica de negocio de la aplicación en otro servidor.
* Database Tier: Representa otra máquina que hospeda la base de datos de la aplicación.

**Estilo o Patrón de Publisher/Suscriber (Publicador/Suscriptor)**

* **Objetivo:** El patrón Publish/Suscribe permite sincronizar el estado de sistemas o componentes de forma desacoplada a partir de un mecanismo de propagación de cambios entre productores de datos (publicadores) y consumidores de datos (suscriptores).
* **Contexto:** Existe una arquitectura de integración compuesta por múltiples aplicaciones conectadas a partir de determinada infraestructura de comunicaciones. Algunas aplicaciones envían múltiples tipos de mensajes; otras aplicaciones están interesadas en consumir dichos mensajes.
* **Problema:** ¿Cómo lograr que las aplicaciones puedan enviar mensajes solamente a las aplicaciones interesadas sin conocer la identidad de los consumidores?
* **Solución:** Extener la infraestructura de comunicación definiendo tópicos o inspeccionando dinámicamente el contenido de los mensajes. Permitir a los consumidores suscribirse a esos tópicos o a los mensajes particulares en base a su contenido. Implementar un mecanismo que permita a los productores publicar mensajes a partir de dichos tópicos.
* **Beneficios:** 
  + Muy bajo acoplamiento: Los publicadores son completamente transparentes a los suscriptores y viceversa en cuanto a identidad, ubicación y tipos de mensajes.
  + Seguridad: Las diferentes infraestructuras de comunicación suelen soportar mecanismos de autenticación, autorización y confidencialidad.
  + Confiabilidad: Se soportan mecanismos de envío confiable y persistente de mensajes y soporta transacaccionabilidad.
* **Desventajas**
  + Complejidad: Identificación y clasificación de mensajes mediante tópicos; uso de APIs o herramientas especializadas para la infraestructura de comunicación.
  + Mantenibilidad: Mantenibilidad de tópicos.
  + Eficiencia: Aumenta sobrecarga al gestionar suscriptores, inspeccionar mensajes y/o tópicos.

**Estilo Repositorio**

* **Descripción del Problema:** Productores y consumidores de datos compartidos no conocen los detalles de implementación del otro.
* **Características:** Utiliza un mediador; se reduce el acoplamiento entre productores y consumidores.
* **Los Clientes:** Ejecutan en threads o procesos de control independientes; pueden ser productores, consumidores o ambos; conocen la estructura del repositorio.
* **El Repositorio:** Puede estar en memoria o en disco (base de datos o archivos); debe tener un esquema específico y conocido por los clientes; es pasivo.
* **Atributos:** Modificabilidad (escalabilidad).
* **Analizar:** Confiabilidad, eficiencia, seguridad.
* **Criterio de Selección:**
  + Cambios previstos en los productores y consumidores de datos.
  + Para aplicaciones en las cuales el problema central es establecer, aumentar y mantener un cuerpo central de información.
  + La información debe manipularse de distintas formas.
  + La información debe almacenarse por períodos prolongados de tiempo.
* **Heurísticas de Diseño:**
  + Este estilo NO se debe seleccionar si se espera agregar, eliminar o modificar tipos de datos del repositorio.
  + En caso de ser necesario agregar un nuevo tipo de datos se debe evitar afectar el acceso a datos existentes.

**Estilo Repositorio Abstracto**

* **Descripción del Problema:** Productores y consumidores de datos compartidos no conocen los detalles de implementación del otro.
* **Características:** Utiliza un mediador; se reduce el acoplamiento entre productores y consumidores; esconde a los clientes los detalles de implementación del repositorio mediante una interfaz abstracta.
* **Los Clientes:** Ejecutan en threads o procesos de control independientes; pueden ser productores, consumidores o ambos; NO conocen la estructura del repositorio.
* **El Repositorio:** Puede estar en memoria o disco (base de datos o archivos); provee interfaces para acceder al repositorio.
* **Atributos:** Modificabilidad (escalabilidad)
* **Analizar:** Confiabilidad, eficiencia, seguridad.
* **Criterio de Selección:**
  + Cambios previstos en el formato de los datos que los productores producen.
  + Para aplicaciones en las cuales el problema central es establecer, aumentar y mantener un cuerpo central de información; la información debe manipularse de de distintas formas; la información debe almacenarse por períodos prolongados de tiempo.
* **Heurísticas de Diseño:**
  + Este estilo se debe seleccionar si se espera cambiar a los productores o consumidores de datos.
  + Generalmente este estilo puede afectar la eficiencia.

**Estilo o Patrón Pipes & Filters (Canales y Filtros)**

* **Descripción del Problema:** Util para aplicaciones que requieren una serie definida de cálculos independientes a realizarse en forma ordenada sobre datos. Cada cálculo se puede encapsular en un filtro. Los datos se pueden pasar mediante canales de comunicación entre los filtros.
* **Criterio de Selección:**
  + Se requieren procesos o transformaciones independientes de flujos de datos de entrada.
  + Los requerimientos de procesamiento pueden cambiar.
  + Los procesos de transformación se quieren reusar.
* **Atributos:** Eficiencia, Modificabilidad
* **Estímulos:** Dos o más flujos de entrada periódicos o esporádicos.
* **Métricas:** Latencia, cambio en los procesos.
* **Los Filtros:** Implementan las unidades de procesamiento; consumen sus entradas en forma incremental; se conectan con otros filtros mediante canales; pueden ejecutar varios filtros en paralelo; son activados por el filtro siguiente (pull), el filtro previo (push), o el filtro es activo haciendo pull y push.
* **Los Canales:** Implementan las conexiones entre filtros, una fuente de datos y un filtro, el último filtro y el resumidero de datos. Si los filtros son activos, el canal los sincroniza, generalmente utilizando un buffer FIFO.
* **La Fuente de Datos (Data Source):** Representa la entrada al sistema; puede estar implementada mediante archivos, sockets, sensores, etc.; puede ser activa (push) o pasiva (pull).
* **El Resumidero (Data Sink):** Recoge los resultados de la línea de filtros; puede estar implementado mediante archivos, sockets, actuadores, etc.; puede ser activo (push) o pasivo (pull).
* **Heurísticas de Diseño:** Al seleccionar este estilo se debe tener en cuenta:
  + La definición de los filtros.
  + La definición de los canales.
  + El volúmen de información a procesar.
  + Estimar la eficiencia real al hacer paralelismo (considerar el costo de transferencia de datos, la velocidad de procesamiento de cada filtro para evitar cuellos de botella, y el consumo de recursos y la asignación de prioridades a los procesos).
  + Definir el mecanismo de manejo de errores en caso de falla de un componente.

**Patrón Reflection**

Se basa en proveer mecanismos para cambiar la estructura y el comportamiento de un sistema en forma dinámica.

* **Problemas que Soluciona:** Permite aplicaciones abiertas a una amplia gama de cambios y extensiones a demanda.
* **Solución:** El patrón prescribe separar la aplicación en dos partes, un nivel de metadatos que provee información sobre determinadas propiedades del sistema que hacen que la aplicación se conozca a sí misma, y un nivel base que incluye la lógica de la aplicación que se basa en la capa de metadatos.
  + **Metadatos:** Provee una autorepresentación del software (metaobjetos) que permite que la aplicación conozca su estructura y comportamiento (por ej., estructuras de datos, algoritmos). Cada metaobjeto encapsular información de algún aspecto de la estructura, estado o comportamiento de la aplicación que puede cambiar. Provee interfaces para modificar metaobjetos.
  + **Nivel Base:** Implementa la lógica de la aplicación. Utiliza los servicios que provee el nivel de metadatos independizándose de los aspectos que pueden cambiar o adaptarse.

**Protocolo de Metaobjetos:** Provee la interfaz para acceder y modificar los metaobjetos.

**Patrón Model-View-Controller (MVC)**

Divide las aplicaciones interactivas en tres componentes: Un modelo, que contiene la funcionalidad central (núcleo) y los datos; vistas, que despliegan la información; y controladores, que manejan las entradas generadas por los usuarios. Utiliza un mecanismo para asegurar la consistencia entre la interfaz de usuario y el modelo.

* **Problemas que Soluciona:** La interfaz de usuario frecuentemente cambia; es posible que sea necesario portar el sistema a otras plataformas; existen distintas necesidades de interfaz de usuario para distintos usuarios; la misma información se despliega de forma distinta; los cambios en los datos se deben reflejar en forma inmediata.
* **Solución:** 
  + **Modelo:** Encapsula la funcionalidad y los datos del sistema; es independiente de la forma en que se despliega la información y de la forma en que se ingresa. Responsabilidades: Exporta servicios que realizan el procesamiento sobre los datos; registra vistas y controladores dependientes; notifica a los componentes dependientes sobre un cambio en los datos.
  + **Vistas:** Despliegan la información; cada vista puede desplegar la misma información de forma distinta; obtiene los datos del modelo; cada vista tiene asociado un controlador. Responsabilidades: Crea e inicializa su controlador correspondiente; despliega la información; implementa un método de actualización (update); obtiene la información del modelo.
  + **Controladores:** Reciben las entradas, normalmente como eventos que codifican acciones y movimientos de botones, el ratón o el teclado; los eventos se traducen a servicios de las vistas o el dominio. Responsabilidades: Acepta eventos como entradas; los traduce a llamados a servicios del modelo o la vista; implementa un método de actualización (update) si fuera necesario.
* **Beneficios:**
  + Múltiples vistas sobre un modelo.
  + Sincronización de las vistas.
  + Independencia del modelo y las vistas.
* **Desventajas:**
  + Número excesivo de actualizaciones.
  + Acoplamiento entre la vista y el controlador con el modelo.

**Patrón SOA**

**Definición:** Arquitectura fuertemente desacoplada diseñada para alcanzar las metas de negocio de una organización. De alguna manera, liberar el negocio de las restricciones que imponen las tecnologías.

SOA **NO** es una tecnología, sino que es una filosofía independiente de cualquier vendedor, producto, tecnología o tendencia industrial. Ningún vendedor jamás ofrecerá un SOA completo, puesto que SOA debe variar de una organización a otra. Comprar la infraestructura para SOA de un sólo vendedor derrota el objetivo de invertir en SOA.

SOAs pueden realizarse a través de Servicios Web, pero los Servicios Web no son necesariamente requeridos para implementar SOA.

SOA no es nuevo ni revolucionario, pues EDI, CORBA y DCOM fueron ejemplos conceptuales de SOA.

SOA **NO** es una metodología, por lo que no puede asegurar el alineamiento de TI con el negocio.

SOAs son como copos de nieve, no hay dos iguales. Una referencia arquitectónica de SOA no necesariamente provea la mejor solución para la organización.

SOA debe ser incremental y ser construida sobre las inversiones actuales.

SOA es un medio, no un fin.

**¿Qué es un Servicio?**

* Un servicio ofrece una capacidad mediante la cual el consumidor de un servicio es satisfecho por el proveedor del mismo.
* El consumidor solicita el servicio.
* El proveedor tiene la capacidad de ejecutar el servicio.
* Un servicio de software provee un conjunto de operaciones requeridas por el consumidor.
* Una operación se enlaza con una interfaz expuesta por una unidad automatizada (la implementación, componentes J2EE, .NET, packages, otro servicio, etc.).
* El consumidor interactúa con el servicio mediante el intercambio de mensajes. El mensaje es enviado a un endpoint (la dirección del servicio).
* El consumidor sólo necesita conocer la operación, la estructura del mensaje a enviar (y obtener como respuesta) y la dirección a la cual lo debe enviar.

**Arquitectura de Servicios**

* La arquitectura de servicios puede conceptualizarse como una colección de servicios, clasificadas en tipos y gobernadas por patrones, principios y políticas. Las políticas son directivas independientemente de como son implementadas. Las políticas pueden ser, por ejemplo, para establecer un tipo de servicio.
* Modelo de Referencia:
  + Aplicaciones se integran consumiendo servicios de uno o más proveedores y se integran en un proceso de negocio.
  + La arquitectura de servicio provee el puente entre la implementación y los consumidores, ofreciendo un conjunto de servicios disponibles con interfaces estándar.
  + La arquitectura de componentes referencia a las diversas aplicaciones que dan soporte al negocio.
  + El modelo se puede leer desde la perspectiva del consumidor o desde la perspectiva de los proveedores. La implementación debe poder variar de proveedor a proveedor entregando el mismo servicio. De igual forma, el proveedor debe ignorar en las aplicaciones que consumen el servicio.

**Clasificación de Servicios (Taxonomía CBDI Forum)**

1. **Core Business Service**

* Provee una visión completa de un recurso de negocio.
* El servicio define la representación de los conceptos perdurables del negocio.
* Encapsula las operaciones que tienen que ver con la recuperación y administración de una entidad de negocio (otras clasificaciones denominan a este tipo de servicio: entity services, core services, o domain services).
* Debe ser lo suficientemente canónico para ser empleado por varios servicios de procesos.

1. **Process Service**

* Se corresponde con un proceso (o subproceso) de negocio de la organización.
* Puede aplicar reglas de negocio, puede almacenar datos específicos (como el estado del proceso) y orquesta las operaciones de los core business services.
* Este esl tipo de servicio que puede ser administrado mediante herramientas de BMP que generan scripts BPEL (u otro lenguaje de workflow).
* El deployment puede realizarse en un Process Engine o en un ESB.

1. **Utility Services**

* Se refieren a los servicios comunes (lógica común, identidad, seguridad, etc).

1. **Underlying Services**

* Se referen a aquellas funcionalidades que son difíciles de emplear como un servicio.
* Se suelen encapsular como un patrón Fachada.

**Principios de Diseño de Servicios**

* Los límites son explícitos.
* Los servicios deben ser autónomos.
* Los servicios comparten esquemas y contratos, no clases.
* La compatibilidad entre servicios se basa en políticas.
* Mientras los preceptos anteroires se focalizan en el diseño de servicios, los servicios por sí mismos **no** constituyen una solución de arquitectura.
* Las características de los servicios describen diversos aspectos de SOA y sus fundamentos, a modo de comprender el rol que estos tienen dentro de la solución arquitectónica.

**BPEL**

* Es un lenguaje markup utilizado para componer un flujo de proceso a partir de un conjunto de servicios.
* Provee mecanismos para:
  + Interacciones asíncronas.
  + Procesamiento paralelo.
  + Manejo de excepciones.
* Es un lenguaje estándar, implementado por la mayoría de los fabricantes: JBOSS, Microsoft, Oracle, IBM, etc.
* ¿Por qué usar BPEL para la orquestración de servicios?
  + Integra un set de servicios para implementar procesos de negocio.
  + Es un lenguaje XML estándar que permite definir la secuencia de actividades de un flujo de proceso.
  + Permite la invocación síncrona y asíncrona de servicios.
  + Habilita el intercambio de mensajes estructurados entre los servicios.
  + Puede ser desarrollado visualmente mediante herramientas de modelado.
* Llamadas Síncronas
  + Creación del Partner Link.
  + Uso de la actividad Assign para preparar el input.
  + Se agrega una actividad Invoke para invocar al servicio.
  + Se agrega una actividad Assign para obtener el output.
* Llamadas Asíncronas
  + Son iniciadas usando una actividad Invoke.
  + Pueden tomar cualquier cantidad de tiempo para completarse.
  + Usan información de direccionamiento de los servicios web para devolver la llamada a una actividad Receive de un proceso que espera la respuesta.
  + Se requiere NetBeans Enterprise Pack 5.5

**Algunos Escenarios**

1. **Integración de Información**
2. **Integración con Sistemas Legados**
3. **Gobierno de Procesos**

**Plataformas Nube (Cloud Platforms)**

Como su nombre lo sugiere, este tipo de plataformas permite a los desarrolladores escribir aplicaciones que ejecutan en la nube, utilizan servicios provistos por la nube, o ambas. Hoy en día se utilizan varios nombres para esta plataforma, incluyendo “plataforma a demanda” y “plataforma como servicio”.

**Plataformas Nube en Contexto: 3 Tipos de Servicios Nube**

* **Software Como Servicio (Software as a Service):** Una aplicación de este tipo ejecuta completamente en la nube (es decir, en servidores de un proveedor de servicio con acceso a Internet). El cliente en sus instalaciones (on-premises) es típicamente un navegador (browser) o algún otro cliente simple. El ejemplo más conocido de este tipo de aplicaciones hoy es probablemente Salesforce.com.
* **Servicios Adjuntos (Attached Services):** Cada aplicación en sus instalaciones provee funciones útiles en sí misma. Algunas veces, una aplicación puede mejorar éstas al acceder a servicios específicos de la aplicación provistos en la nube. Dado que estos servicios sólo pueden ser usados por una aplicación en particular, se los puede pensar como adjuntos a la aplicación. Un ejemplo popular de consumo de esto es iTunes de Apple, donde la aplicación de escritorio es útil para la reproducción de música, mientras que un servicio adjunto permite comprar nuevos contenidos de audio y video.
* **Plataformas Nube:** Una plataforma nube provee servicios basados en la nube para crear aplicaciones. En lugar de construir su base personalizada, por ejemplo, los creadores de una nueva aplicación SaaS podrían construir sobre una plataforma nube. Los usuarios directos de una plataforma nube son desarrolladores, no usuarios finales.

**Plataforma:** Una manera amplia de pensar la plataforma, es verla como cualquier software que provee servicios accesibles al desarrollador para crear aplicaciones.

Tanto sea en instalaciones o en la nube, una plataforma de una aplicación comprende tres partes:

* **Base (Foundation):** Casi toda aplicación usa alguna plataforma de software en la máquina donde ejecuta. Típicamente, esto incluye varias funciones de soporte, tales como librerías y almacenamiento estándar, y un sistema operativo de base.
* **Servicios de Infraestructura (Infrastructure Services):** En un ambiente distribuido moderno, las aplicaciones usan frecuentemente servicios básicos provistos en otras computadoras. Es común proveer almacenamiento remoto, por ejemplo, servicios de integración, servicio de identidad, y más.
* **Servicios de Aplicación (Application Services):** A medida que más y más aplicaciones se vuelven orientadas al servicio, las funciones que ofrecen se vuelven accesibles a nuevas aplicaciones. Aunque estás aplicaciones existen primariamente para proveer servicios a usuarios finales, esto hace que también sean parte de la plataforma de la aplicación.