**PATRONES**

**Layered Pattern (Modificabilidad)**

Contexto: Todos los sistemas complejos experimentan la necesidad de desarrollar y evolucionar partes del sistema de forma independiente. Por esta razón, los desarrolladores del sistema necesitan una separación clara y bien documentada de las responsabilidades, por lo que los módulos del sistema pueden ser desarrollados y mantenidos de forma independiente.

Problema: El software tiene que ser segmentado de tal manera que los módulos puedan ser desarrollados y evolucionen por separado con poca interacción entre las partes, contribuyendo a la portabilidad, modificabilidad, y la reutilización.

Solución:

Para lograr esta separación de responsabilidades, el patrón divide el software en unidades llamadas capas, cada capa es una agrupación de módulos que ofrece un conjunto coherente de servicios.

Como limitaciones en la implementación del patrón en capas, las relaciones deben ser unidireccionales. Las capas están completamente particionadas, cada partición se exponen a través de una interfaz pública, las capas se crean para interactuar de acuerdo con una relación de orden estricto (desde capas superiores a inferiores). Si (A,B) es una relación, se dice que a la aplicación de la capa A se le permite utilizar cualquiera de los servicios públicos prestados por la capa B. Algunas veces es necesaria la interacción entre capas no adyacentes aunque esto no es recomendado. El caso en el que una capa superior utiliza módulos en una capa inferior no adyacentes se denomina capa de puente. Si se producen muchos casos de la capa de puente, el sistema no puede cumplir con su portabilidad y modificabilidad justamente lo que se intenta lograr implementando el patrón en capas. La implementación del patrón no es gratis, ya que hay que diseñar y construir las capas lo cual a menudo tiene costo y complejidad, además, si las capas no se diseñan correctamente se pueden introducir ineficiencias al no proporcionar las abstracciones de nivel inferior que los programadores necesitan en los niveles superiores.

Las capas siempre añaden una penalización de rendimiento para un sistema. Si se realiza una llamada a una función en la capa de más arriba, esto puede tener que atravesar muchas capas inferiores antes de ser ejecutado por el hardware. Cada una de estas capas añade algo de sobrecarga, como mínimo, en la forma de cambio de contexto.

**VISION**

El patrón de capas (las capas definen la agrupación de módulos que ofrecen un conjunto coherente de servicios) y la unidireccionalidad permite el uso de relación entre las capas. El patrón en general se muestra gráficamente por el apilamiento de cajas que representan capas

**ELEMENTOS**

Capa, un tipo de módulo. La descripción de una capa debe definir qué módulos tiene la capa y contener una caracterización del conjunto cohesivo de la capa de servicios que ofrece

**RELACIONES**

El diseño debe definir cuáles son las reglas de uso de capas son (por ejemplo, "a una capa se le permite usar cualquier capa inferior" o "a una capa se le permite usar sólo la capa inmediatamente inferior ") y las excepciones permitidas

**RESTRICCIONES**

- Cada pieza de software se asigna a exactamente una capa

- Hay por lo menos dos capas (pero por lo general hay tres o más)

- No debe haber relaciones circulares (es decir , una capa inferior no puede utilizar la capa de arriba)

**DEBILIDADES**

- La adición de capas añade costos iniciales y complejidad a un sistema

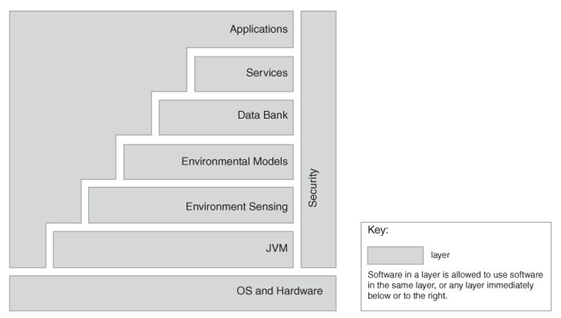
- Las capas reducen el rendimiento

Algunos detalles

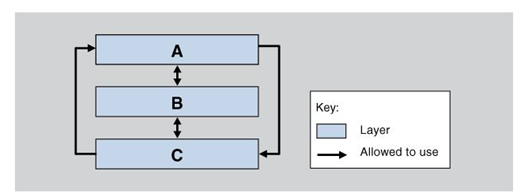
Una arquitectura en capas es una de las pocas donde las conexiones entre los componentes pueden ser mostrados por adyacencia.

El modelo de capas es uno de los patrones más utilizados en toda ingeniería de software, pero a menudo sorprende la cantidad de errores.

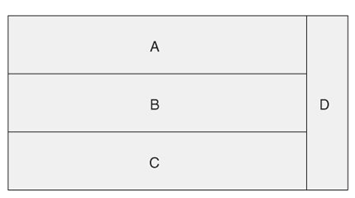
1. Es imposible ver una pila de cajas y saber si se permite o no la transición entre capas, es decir, si se puede utilizar una capa de cualquier capa inferior, o simplemente la siguiente más abajo. Todo arquitecto tiene que incluir la respuesta en la clave en la notación del diagrama



2. Cualquier conjunto de cajas apiladas no constituye una arquitectura en capas. En la figura a continuación, se utilizan flechas en lugar de adyacencia para indicar las relaciones entre las cajas. Aquí, está permitido que todos usen todo. Esto no es decididamente una arquitectura en capas. La razón es que si la capa A se sustituye por una versión diferente, la capa C (que utiliza a esta en la figura) bien podría tener que cambiar. No queremos que nuestra capa de máquina virtual cambie cada vez que cambiamos la capa de aplicación.

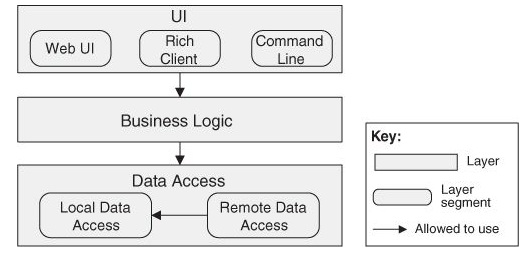


3. Muchas arquitecturas pretenden parecerse a la arquitectura en capas. El siguiente diagrama probablemente signifique que los módulos de A, B o C pueden usar módulos en D, pero sin una clave que nos diga con certeza, podría significar cualquier cosa. "Sidecars" como éste a menudo contienen utilidades comunes (a veces importados), como controladores de errores, protocolos de comunicación o mecanismos de acceso a bases de datos. Este tipo de diagrama sólo tiene sentido en el caso en el que no se permite ninguna capa de puente en la pila principal. De lo contrario, D sólo se podría hacer la capa más inferior de la pila principal, y la geometría "sidecar" sería innecesaria. Pero todavía sorprende la frecuencia con la que se ven este diseño.



Muchas veces los diagramas de capas incluyen segmentos direccionados para indicar cuál es la relación entre las capas.

En la siguiente imagen hay un diseño de capas, segmentado además, mostrando los módulos que contienen las capas, si se quitaran las flechas seria difícil entender el diagrama



Finalmente el punto más importante en el diseño en capas es no permitir el uso de cualquier capa superior, un módulo usa otro modulo cuando este depende de una respuesta del anterior

Pero una capa tiene permitido hacer un llamado hacia una capa superior siempre que no espere una respuesta de ella. (Por ejemplo al enviar una respuesta excepción)

Por ejemplo para el manejo de errores, A llama a B, si B falla llama a un método en A para que maneje los errores pero no tiene idea que es lo que hace, no depende del comportamiento ni contenido de A.

#### 

#### **Model-View-Controller Pattern (usabilidad - modificabilidad)**

Contexto: El software de interfaz de usuario suele ser la parte que se modifica con más frecuencia en una aplicación interactiva. Por esta razón, es importante que la UI este separada de la lógica. Los usuarios a menudo desean ver los datos desde diferentes perspectivas, como un gráfico de barras o un gráfico circular. Estas representaciones deben reflejar el estado actual de los datos.

Problema: Separa la UI de la lógica de la aplicación, manteniendo la sensibilidad en los datos. Crear múltiples vistas para los usuario y mantener coordinados los cambios y el mantenimiento de la aplicación.

Solución:

El patrón modelo-vista- controlador (MVC) separa funcionalidad de la aplicación en tres tipos de componentes:

· Un modelo, que contiene datos de la aplicación

· Una vista que muestra una parte de los datos subyacentes e interactúa con el usuario

· Un controlador, que media entre el modelo y la vista y gestiona las notificaciones de cambios de estado

MVC no es apropiado para cualquier situación. El diseño y la ejecución de tres tipos diferentes de componentes, junto con sus diversas formas de interacción, puede ser costoso, y este costo puede no tener sentido para las interfaces de usuario relativamente simples. Además, jugar con las abstracciones de herramientas de interfaz de usuario de MVC y el negocio no es perfecto. La vista y el controlador se separaron de entrada y de salida, pero estas funciones se combinan a menudo en aparatos individuales. Esto puede resultar en una falta de coincidencia conceptual entre la arquitectura y el conjunto de herramientas de interfaz de usuario.

**VISTA**

El patrón MVC distribuye funcionalidad del sistema en tres componentes: un modelo, una vista y un controlador que media entre el modelo y la vista

**ELEMENTOS**

El modelo es una representación de los datos de la aplicación o del estado, y que contiene (o proporciona una interfaz) lógica de la aplicación

La vista es un componente de interfaz de usuario que, o bien produce una representación del modelo o es una forma de entrada del usuario, o ambos

El controlador gestiona la interacción entre el modelo y la vista, la traducción de las acciones del usuario en los cambios en el modelo o cambios en la vista

**RELACIONES**

La relación de notificación conecta instancias de modelo, vista y controlador, notifica los cambios de estado de los elementos pertinentes

**RESTRICCIONES**

Debe haber por lo menos un caso de cada uno de Modelo, vista y el controlador el componente de modelo no debe conectar directamente con el controlador

**DEBILIDADES**

La complejidad no vale la pena para las interfaces de usuario sencillas

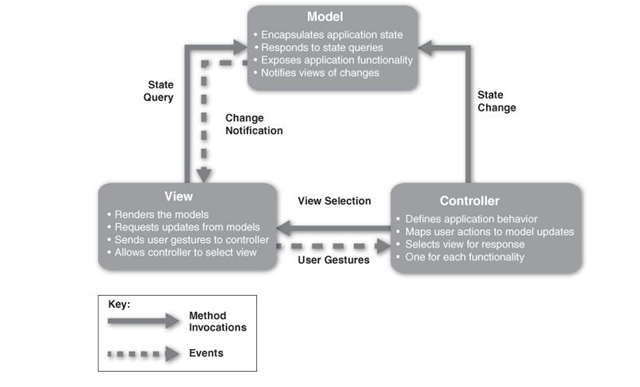
MVC pueden no ser buenas abstracciones para algunos kits de herramientas de interface de usuario

De hecho es posible asociar varias vistas y controladores a un modelo.

Por ejemplo, un conjunto de datos de negocios puede ser representado como columnas de números en una hoja de cálculo, como un gráfico de dispersión, o como un gráfico circular. Cada uno de ellos es una vista separada, y esta vista se puede actualizar dinámicamente a medida que cambia el modelo (por ejemplo, mostrar las transacciones en directo en un sistema de procesamiento de transacciones). Un modelo puede ser actualizado por diferentes controladores, por ejemplo, un mapa podría hacer zoom y panorámicas a través de los movimientos del ratón, movimientos trackball, los clics del teclado o comandos de voz, cada una de estas diferentes formas de entrada tiene que ser gestionado por un controlador.

Los componentes de MVC están conectados entre sí a través de una notificación, como eventos o devoluciones de llamada. Estas notificaciones contienen actualizaciones de estado. Un cambio en el modelo debe ser comunicado a las vistas para que puedan ser actualizados. Un evento externo, tal como una entrada de usuario, debe ser comunicada al controlador, que a su vez puede actualizar la vista y / o el modelo. Las notificaciones pueden ser recibidas o lanzadas.

Debido a que estos componentes están débilmente acoplados, es fácil de desarrollar y probar en paralelo, y los cambios en uno tienen un impacto mínimo en los demás.



#### **Publish-Subscribe Pattern (observer- modificabilidad)**

Contexto: Hay una serie de productores independientes y los consumidores de los datos que deben interactuar. El número exacto y la naturaleza de los datos de los productores y los consumidores no están predeterminados o fijos, ni son los datos que se comparten.

Problema: ¿Cómo podemos crear mecanismos de integración que soporten la capacidad de transmitir mensajes entre los productores y los consumidores de tal manera que no se den cuenta de la identidad de cada uno, o posiblemente incluso su existencia?

Solución: En el patrón publicación-suscripción, los componentes interactúan a través de mensajes, anuncios o eventos. Los componentes pueden suscribirse a un conjunto de eventos. En tiempo de ejecución el patrón se asegura que cada evento publicado se entrega a todos los suscriptores de ese evento. Por lo tanto, la principal forma de conector en estos patrones es un bus de eventos. Cualquier componente puede ser a la vez un editor y un suscriptor.

**VISION GENERAL**

Los componentes publican y se suscriben a los eventos. Cuando un evento es anunciado por un componente, se avisa a todos los suscriptores registrados

**ELEMENTOS**

Cualquier componente c & c con al menos una publicación o suscripción, posee un punto de entrada- salida (una conexión) donde incluyen los eventos que se publican y a los que está suscrito, y el nivel de detalle de los eventos

El conector publish - suscribe, que se anunciará y escuchara los roles para los componentes que deseen publicar y suscribirse a los eventos

**RELACIONES**

Componentes anuncian eventos y otros componentes que están registrados para recibir eventos, los reciben

**RESTRICCIONES**

Todos los componentes están conectados a un distribuidor de eventos que puede ser visto como un autobús - conector - o un componente. Se conoce las formas de anunciar y escuchar eventos.

Las restricciones pueden existir para los componentes pueden ser, que cosas pueden escuchar, si un componente puede escuchar a sus propios actos, número de conectores de publicación-suscripción que pueden existir dentro de un sistema.

Un componente puede ser a la vez un editor y un suscriptor, porque tienen puertos de ambos tipos

**DEBILIDADES**

Aumenta la latencia y tiene un efecto negativo en la escalabilidad y la previsibilidad del tiempo de entrega de mensajes

Menor control sobre la ordenación de los mensajes, y la entrega de mensajes no está garantizada.

El patrón añade una capa de direccionamiento indirecto entre emisores y receptores. Esto tiene un efecto negativo sobre la latencia y potencialmente sobre la escalabilidad, dependiendo de cómo se implementa. Uno normalmente no querría usar publicación-suscripción en un sistema que tiene plazos de tiempo real estricto que cumplir, ya que introduce incertidumbre en los plazos de entrega de mensajes.

Además, el patrón de publicación-suscripción sufre, ya que proporciona un menor control sobre ordenamiento de mensajes y el envío de mensajes no está garantizado (porque el remitente no puede saber si el receptor está escuchando) . Esto puede hacer al patrón de publicación-suscripción inapropiado para las interacciones complejas en el estado compartido.

Hay algunos refinamientos específicos de este patrón que son de uso común.

A continuación se describen varios de estos más:

El modelo computacional para el patrón de publicación-suscripción es mejor como un sistema de procesos u objetos independientes, que reaccionan a los eventos generados por su entorno, y que a su vez causan reacciones en otros componentes como un efecto secundario de sus anuncios de eventos.

Ejemplos típicos de los sistemas que emplean el patrón de publicación-suscripción son los siguientes:

* Interfaces gráficas de usuario, en el que las acciones de entrada de bajo nivel de un usuario se tratan como eventos que se envían a los manipuladores de entrada apropiados
* Aplicaciones basadas en MVC, en la que los componentes de visión son notificados cuando el estado de un objeto de modelo de los cambios
* Los sistemas de planificación de recursos empresariales (ERP), que integran varios componentes, cada uno de los cuales sólo está interesada en un subconjunto de los eventos del sistema
* Entornos de programación extensible, en el que las herramientas se coordinan a través de eventos
* Listas de correo, donde un conjunto de abonados puede registrar interés en los temas específicos
* Las redes sociales, donde los "amigos" se notifican cuando se producen cambios en el sitio web de una persona

El patrón de publicación-suscripción se utiliza para enviar eventos y mensajes a un conjunto indeterminado de destinatarios. Debido a que el conjunto de destinatarios de eventos es desconocido para el productor del evento, no se tiene exactitud de los productores, en general, dependerá de los destinatarios. Por lo tanto, los nuevos receptores se pueden añadir sin modificaciones a los productores.

Habiendo componentes que ignoran la identidad de otro las modificaciones del sistema son fáciles (añadiendo o quitando productores y consumidores de datos), pero a costa de rendimiento en tiempo de ejecución, ya que la infraestructura de publicación-suscripción es una especie de indirecta, que añade latencia. Además, si el conector de publicación-suscripción falla por completo, esto es un punto único de fallo para todo el sistema.

El modelo de publicación-suscripción puede tomar varias formas:

* Basado en listas de publicación-suscripción es una realización del patrón en el que todos los editores mantiene una lista de suscripción que hayan registrado su interés en recibir el evento. Esta versión es menos desacoplada que otros, como veremos más adelante, y por lo tanto no proporciona tanta modificabilidad, pero puede ser muy eficiente en términos de sobrecarga de tiempo de ejecución. Además, si se distribuyen los componentes, no hay ningún punto único de fallo.

· Difusión sistema distribuido o servicios que actúan en nombre de los suscriptores, examinan cada caso, que llega y determinan si el evento publicado es de interés. Esta versión tiene el potencial de ser muy ineficiente si hay un montón de mensajes y la mayoría de los mensajes no son de interés para un abonado particular.

Publicación-suscripción se distingue de las dos variantes anteriores, que son ampliamente clasificados como " basado en temas. " Temas · basadas en contenido son eventos predefinidos o mensajes, y un componente suscriba a todos los eventos en el tema. Contenido, por otra parte, es mucho más general. Cada evento está asociado con un conjunto de atributos y se entrega a un abonado únicamente si dichos atributos coinciden con los patrones definidos por sus suscriptores.

En la práctica, el patrón de publicación-suscripción es típicamente realizado por alguna forma de middleware orientado a mensajes, donde el middleware se realiza como un corredor, la gestión de las conexiones y canales de información entre productores y consumidores. Este middleware es a menudo responsable de la transformación de los mensajes (o protocolos de mensajes) ​​, además de enrutamiento y, a veces el almacenamiento de los mensajes. Así, el patrón de publicación-suscripción hereda las fortalezas y debilidades del modelo de corredor.

#### **Multi-tier Pattern (modificabilidad, escalabilidad, facilidad para deployar, disponibilidad)**

“SE DISTRIBUYEN LOS NIVELES EN DIFERENTES CAPAS FISICAS”

El patrón de múltiples niveles es un patrón de C & C o un patrón de asignación, dependiendo de los criterios usados ​​para definir los niveles. Los niveles pueden ser creados para los componentes de un grupo de funcionalidad similar, en cuyo caso se trata de un patrón de C&C. Sin embargo, en muchos, si no la mayoría, de los casos los niveles se definen con la mirada puesta en el entorno informático en el que se ejecutará el software: A nivel de cliente en un sistema de la empresa no se ejecuta en el equipo que aloja la base de datos . Eso hace que sea un modelo de asignación, la cartografía elementos - quizá de software producidos por la aplicación de patrones de C&C. Por esa razón, hemos decidido incluirla como un patrón de asignación.

Contexto: En una implementación distribuida, a menudo hay una necesidad de distribuir la infraestructura de un sistema en distintos subconjuntos. Esto puede ser por razones operativas o comerciales (por ejemplo, las diferentes partes de la infraestructura pueden pertenecer a diferentes organizaciones).

Problema: ¿Cómo podemos dividir el sistema en un número de ejecución de cómputo independiente de las estructuras de los grupos de software y hardware conectado por algunos medios de comunicación? Esto se hace para proporcionar entornos de servidor específica,

Optimizada para los requisitos operativos y uso de recursos.

Solución: Las estructuras de ejecución de muchos sistemas se organizan como un conjunto de agrupaciones lógicas de los componentes. Cada agrupación se denomina nivel. La agrupación de los componentes en capas puede basarse en una variedad de criterios, tales como el tipo de componente, que comparten el mismo entorno de ejecución, o que tengan el mismo propósito de tiempo de ejecución.

El uso de niveles se puede aplicar a cualquier colección (o patrón) de los componentes de tiempo de ejecución, aunque en la práctica se utiliza más a menudo en el contexto de los patrones de cliente-servidor. Tiers inducen restricciones topológicas que restringen qué componentes pueden comunicarse con otros componentes. En concreto, pueden existir conexiones sólo entre componentes en el mismo nivel o que residen en niveles adyacentes. El patrón de varios niveles se encuentra en muchos de Java EE y Microsoft. NET es un ejemplo de organización en los niveles derivados del modelo cliente-servidor.

Además, los niveles pueden limitar los tipos de comunicación que pueden tener lugar en todas las capas adyacentes. Por ejemplo, algunos patrones escalonados requieren una comunicación llamada de retorno en una dirección pero notificación de eventos basado en el otro.

La principal debilidad de la arquitectura de varios niveles es su costo y complejidad. Para los sistemas simples, los beneficios de la arquitectura de varios niveles no pueden justificar sus costos iniciales y en curso, en términos de hardware, software, y diseño y complejidad de la implementación.

Los niveles no son componentes, son como agrupaciones lógicas de los componentes. Además, no hay que confundir los niveles con capas, las capas son un patrón de módulos (una unidad de aplicación), mientras que niveles se aplica únicamente a las entidades de tiempo de ejecución .

**VISION GENERAL**

Las estructuras de ejecución de muchos sistemas se organizan como un conjunto de agrupaciones lógicas de los componentes, cada grupo se denomina nivel. La agrupación de los componentes en capas puede basarse en una variedad de criterios, tales como el tipo de componente, que comparten el mismo entorno de ejecución, o que tengan el mismo propósito de tiempo de ejecución

**ELEMENTOS**

Nivel, que es una agrupación lógica de componentes de software

Los niveles, se pueden formar sobre la base de PLATAFORMAS de computación comunes, en cuyo caso esas plataformas también son elementos del patrón

**RELACIONES**

Son parte integrante de los componentes del grupo en niveles. Muestran cómo los niveles y los componentes que contienen interactúan uno con otro, en el caso de que los niveles se asignan a plataformas de computación

**RESTRICCIONES**

Un componente de software pertenece exactamente a un nivel

**DEBILIDADES**

Costo por adelantado y la complejidad

Aumenta la latencia por comunicación remota

Los niveles sirven para que sea más fácil garantizar la seguridad, y para optimizar el rendimiento y la disponibilidad de formas especializadas. También mejora la modificabilidad del sistema, como los subgrupos computacionalmente independientes necesitan ponerse de acuerdo sobre los protocolos para la interacción, reduciendo así su acoplamiento.

