Ferrando Eduardo – Ferrando Matias

PATRONES DE SOFTWARE

Apunte de Catedra

INTRODUCCIÓN

Patrones de Diseño

Los patrones de diseño son el esqueleto de las soluciones a problemas comunes en el desarrollo de software. En otras palabras, brindan una solución ya probada y documentada a problemas de desarrollo de software que están sujetos a contextos similares. Debemos tener presente los siguientes elementos de un patrón: su nombre, el problema (cuando aplicar un patrón), la solución (descripción abstracta del problema) y las consecuencias (costos y beneficios).

Los patrones de diseño facilitan la reutilización de arquitecturas y diseños de software exitosos.

Patrones de Arquitectura

Son patrones de diseño de software que ofrecen soluciones a problemas de arquitectura de software en ingeniería de software. Dan una descripción de los elementos y el tipo de relación que tienen junto con un conjunto de restricciones sobre cómo pueden ser usados. Un patrón arquitectónico expresa un esquema de organización estructural esencial para un sistema de software, que consta de subsistemas, sus responsabilidades e interrelaciones. En comparación con los patrones de diseño, los patrones arquitectónicos tienen un nivel de abstracción mayor.

Patrones de Arquitectura Vs. Patrones de Diseño

Si queremos creer que realmente existe diferencia, entonces es fácil verla midiendo el impacto al aplicar el patrón: si este es relevante a la totalidad del sistema, entonces hablamos de un patrón de arquitectura; en cambio, si este sólo concierne a un subcomponente, nos referimos a un patrón de diseño.

Tomen como caso el patrón de Layers (capas), este es claramente un patrón arquitectónico, ya que concierne al diseño general de la aplicación. Mientras que el patrón Active Record, que lidia con los mecanismos de persistencia de datos es un patrón de diseño.

Pero, ¿qué pasa cuando lo que era totalidad se vuelve un subcomponente? ¿Cambiarían entonces sus patrones de arquitectura a diseño? Aquí es donde no es tan fácil la respuesta.

Hay otros patrones que solapan responsabilidades de todo-parte, por ejemplo el MVC. Según como se aplique puede ser un patrón de diseño o arquitectura.

PATRONES DE ARQUITECTURA

Los patrones arquitectónicos, o patrones de arquitectura, también llamados arquetipos ofrecen soluciones a problemas de arquitectura de software en ingeniería de software. Dan una descripción de los elementos y el tipo de relación que tienen junto con un conjunto de restricciones sobre cómo pueden ser usados. Un patrón arquitectónico expresa un esquema de organización estructural esencial para un sistema de software, que consta de subsistemas, sus responsabilidades e interrelaciones. En comparación con los patrones de diseño, los patrones arquitectónicos tienen un nivel de abstracción mayor.

Aunque un patrón arquitectónico comunica una imagen de un sistema, no es una arquitectura como tal. Un patrón arquitectónico es más un concepto que captura elementos esenciales de una arquitectura de software. Muchas arquitecturas diferentes pueden implementar el mismo patrón y por lo tanto compartir las mismas características. Además, los patrones son a menudo definidos como una cosa estrictamente descrita y comúnmente disponible. Por ejemplo, la arquitectura en capas es un estilo de llamamiento-y-regreso, cuando define uno un estilo general para interaccionar. Cuando esto es descrito estrictamente y comúnmente disponible, es un patrón.

Uno de los aspectos más importantes de los patrones arquitectónicos es que encarnan diferentes atributos de calidad. Por ejemplo, algunos patrones representan soluciones a problemas de rendimiento y otros pueden ser utilizados con éxito en sistemas de alta disponibilidad.

Algunos patrones de arquitectura comunes:

- Patrón de capas
- Patrón cliente-servidor
- Patrón maestro-esclavo
- Patrón de filtro de tubería
- Patrón de igual a igual
- Modelo-vista-controlador

Arquitectura basada en capas

La arquitectura basada en capas se enfoca principalmente en el agrupamiento de funcionalidad relacionada dentro de una aplicación en distintas capas que son colocadas verticalmente una encima de otra, la funcionalidad dentro de cada capa se relaciona con un rol o responsabilidad especifica.

Capa de presentación

Es la responsable de la presentación visual de la aplicación. La capa de presentación enviará mensajes a los objetos de la capa de negocios o intermedia, la cual o bien responderá entonces directamente o mantendrá un diálogo con la capa de la acceso a datos, la cual proporcionará los datos que se mandarían como respuesta a la capa de presentación.

Podemos decir que es la que se presenta al usuario, llamada también formulario o interfaz de presentación, esta captura los datos del usuario en el formulario e invoca a la capa de negocio, trasmitiéndole los requerimientos del usuario, ya sea de almacenaje, edición, o de recuperación de la información para la consulta respectiva.

Capa de negocio

Es la responsable del procesamiento que tiene lugar en la aplicación. Por ejemplo, en una aplicación bancaria el código de la capa de presentación se relacionaría simplemente con la monitorización de sucesos y con el envío de datos a la capa de procesamiento. Esta capa intermedia contendría los objetos que se corresponden con las entidades de la aplicación. Esta capa intermedia es la que conlleva capacidad de mantenimiento y de reutilización.

Contendrá objetos definidos por clases reutilizables que se pueden utilizar una y otra vez en otras aplicaciones. Estos objetos se suelen llamar objetos de negocios y son los que contienen la gama normal de constructores, métodos para establecer y obtener variables, métodos que llevan a cabo cálculos y métodos, normalmente privados, en comunicación con la capa de acceso a datos.

Es en esta capa donde se reciben los requerimientos del usuario y se envían las respuestas tras el proceso, a requerimiento de la capa de presentación. Se denomina capa de negocio o lógica del negocio, es aquí donde se establecen todas las reglas que deben cumplirse. En realidad, se puede tratar de varias funciones, por ejemplo, puede controlar la integridad referencial, otro que se encargue de la interfaz, tal como abrir y cerrar ciertos formularios o funcionalidades que tengan que ver con la seguridad, menús, etc.; tiene los métodos que serán llamados desde las

distintas partes de la interfaz o para acceder a la capa de datos, tal como se apreciará en el ejemplo.

Esta capa interactúa con la capa de presentación para recibir las solicitudes y presentar los resultados, y con la capa de datos, para solicitar al manejador de base de datos que realice una operación de almacenamiento, edición, eliminación, consulta de datos u otra.

Capa de datos

Esta capa se encarga de acceder a los datos, se debe usar la capa de datos para almacenar y recuperar toda la información de sincronización del Sistema.

Es aquí donde se implementa las conexiones al servidor y la base de datos propiamente dicha, se invoca a los procedimientos almacenados los cuales reciben solicitudes de almacenamiento o recuperación de información desde la capa de negocio.

Capas y Niveles

Es importante distinguir los conceptos de “Capas” (Layers) y “Niveles” (Tiers). Las capas se ocupan de la división lógica de componentes y funcionalidad y no tienen en cuenta la localización física de componentes en diferentes servidores o en diferentes lugares. Por el contrario, los Niveles se ocupan de la distribución física de componentes y funcionalidad en servidores separados. Teniendo en cuenta la topología de redes y localizaciones remotas.

Las arquitecturas de N niveles facilitan la presencia de sistemas distribuidos en los que se pueden dividir los servicios y aumentar la escalabilidad y mantenimiento de los mismos.

El dividir en capas una aplicación, permite la separación de responsabilidades lo que proporciona una mayor flexibilidad y un mejor mantenimiento.

Por ejemplo, en una aplicación con una capa de presentación, una capa de lógica y una capa de acceso a datos, la responsabilidad de la capa de presentación es la de interactuar con el usuario, solicitando y proporcionando la información que el usuario requiere.

La responsabilidad de La capa de Lógica de negocio es la de hacer cumplir las reglas de negocio o requerimientos de la aplicación mientras que la responsabilidad de la Capa de acceso a datos es la de recuperar y modificar datos del origen de datos.

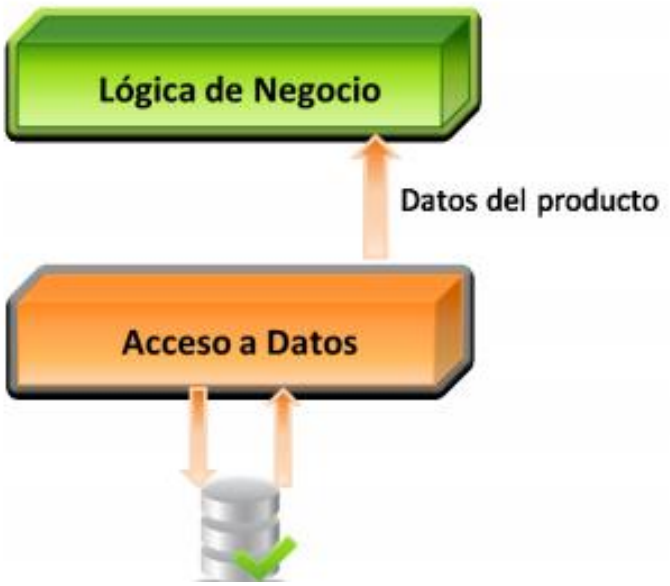
Ejemplo:

Supongamos que tenemos un sistema de ventas:

1. La capa de presentación solicita el ID del producto que el usuario desea comprar y se lo proporciona a la capa de lógica de negocio.

2. La capa de lógica de negocio debe verificar si aún hay existencias del producto requerido por el usuario y para esto solicita los datos del producto a la capa de acceso a datos pasándole el ID del producto a buscar.

3. La capa de acceso a datos busca la información del producto en la fuente de datos y lo devuelve a la capa de lógica de negocio.



4. Con los datos recibidos, la capa de lógica de negocio determina si el producto puede ser vendido o no y proporciona la respuesta de la consulta a la capa de presentación.

5. La capa de presentación muestra al usuario la respuesta que la capa de lógica le ha proporcionado.

Relación entre capas:

- ALTA COHESION (relación estrecha)
- BAJO ACOPLAMIENTO (Una capa **NO** debe hacer lo que hace otra)

Ventajas de programar en capas:

- El primer punto y uno de los más importantes es la reutilización del código, al poder reutilizar una parte del código ya escrito y verificado su funcionamiento, ahorraremos mucho tiempo valioso en escribir código y verificar el funcionamiento del mismo, evitando además cantidad de errores.
- Al reutilizar código, nuestro sistema será más chico en cuanto a dicho código, reduciendo su tamaño y en consecuencia su "peso", será más fácil de comprender, y de encontrar posibles errores o fallas.
- Al lograr obtener un bajo acoplamiento entre las capas, se facilita tanto la corrección del sistemas como así también las sucesivas modificaciones, por ejemplo al tener que realizar una modificación en la base de datos, solamente deberemos corregir la capa de Acceso a Datos, sin tener necesidad de modificar nada en las demás capas, o si un cliente cambia por ejemplo la manera de realizar un descuento, o agrega nuevas maneras de hacerlos, solo modificaremos la parte que corresponda dentro de la capa de Lógica de Negocio.

Patron Cliente – Servidor

Esta arquitectura consiste básicamente en un cliente que realiza peticiones a otro programa (el servidor) que le da respuesta. Aunque esta idea se puede aplicar a programas que se ejecutan sobre una sola computadora es más ventajosa en un sistema operativo multiusuario distribuido a través de una red de computadoras. La interacción cliente-servidor es el soporte de la mayor parte de la comunicación por redes. Ayuda a comprender las bases sobre las que están construidos los algoritmos distribuidos.

Partes que componen el sistema

Cliente: Programa ejecutable que participa activamente en el establecimiento de las conexiones. Envía una petición al servidor y se queda esperando por una respuesta. Su tiempo de vida es finito una vez que son servidas sus solicitudes, termina el trabajo.

Servidor: Es un programa que ofrece un servicio que se puede obtener en una red. Acepta la petición desde la red, realiza el servicio y devuelve el resultado al solicitante. Al ser posible

implantarlo como aplicaciones de programas, puede ejecutarse en cualquier sistema donde exista TCP/IP y junto con otros programas de aplicación. El servidor comienza su ejecución antes de comenzar la interacción con el cliente. Su tiempo de vida o de interacción es “interminable”.

Los servidores pueden ejecutar tareas sencillas (caso del servidor hora día que devuelve una respuesta) o complejas (caso del servidor ftp en el cual se deben realizar operaciones antes de devolver una respuesta). Los servidores sencillos procesan una petición a la vez (son secuenciales o interactivos), por lo que no revisan si ha llegado otra petición antes de enviar la respuesta de la anterior.

Los más complejos trabajan con peticiones concurrentes aun cuando una sola petición lleve mucho tiempo para ser servida (caso del servidor ftp que debe copiar un archivo en otra máquina). Son complejos pues tienen altos requerimientos de protección y autorización. Pueden leer archivos del sistema, mantenerse en línea y acceder a datos protegidos y a archivos de usuarios. No puede cumplir a ciegas las peticiones de los clientes, deben reforzar el acceso al sistema y las políticas de protección. Los servidores por lo general tienen dos partes:

- Programa o proceso que es responsable de aceptar nuevas peticiones: Maestro o Padre.
- Programas o procesos que deben manejar las peticiones individuales: Esclavos o Hijos.

Tareas del programa maestro:

- Abrir un puerto local bien conocido al cual pueda acceder los clientes.
- Esperar las peticiones de los clientes.
- Elegir un puerto local para las peticiones que llegan en informar al cliente del nuevo puerto, (innecesario en la mayoría de los casos).
- Iniciar un programa esclavo o proceso hijo que atienda la petición en el puerto local, (el esclavo cuando termina de manejar una petición no se queda esperando por otras).
- Volver a la espera de peticiones mientras los esclavos, en forma concurrente, se ocupan de las anteriores peticiones.

Características de la arquitectura Cliente – Servidor

- Combinación de un cliente que interactúa con el usuario, y un servidor que interactúa con los recursos a compartir. El proceso del cliente proporciona la interfaz entre el usuario y el resto del sistema. El proceso del servidor actúa como un motor de software que maneja recursos compartidos tales como bases de datos, impresoras, Módem, etc.
- Las tareas del cliente y del servidor tienen diferentes requerimientos en cuanto a recursos de cómputo como velocidad del procesador, memoria, velocidad y capacidades del disco e input-output devices.
- Se establece una relación entre procesos distintos, los cuales pueden ser ejecutados en la misma máquina o en máquinas diferentes distribuidas a lo largo de la red.
- Existe una clara distinción de funciones basadas en el concepto de"servicio", que se establece entre clientes y servidores.
- La relación establecida puede ser de muchos a uno, en la que un servidor puede dar servicio a muchos clientes, regulando su acceso a los recursos compartidos.
- Los clientes corresponden a procesos activos en cuanto a que son estos los que hacen peticiones de servicios. Estos últimos tienen un carácter pasivo, ya que esperan peticiones de los clientes.
- No existe otra relación entre clientes y servidores que no sea la que se establece a través del intercambio de mensajes entre ambos. El mensaje es el mecanismo para la petición y entrega de solicitudes de servicios.
- El ambiente es heterogéneo. La plataforma de hardware y el sistema operativo del cliente y del servidor no son siempre los mismos. Precisamente una de las principales ventajas de esta arquitectura es la posibilidad de conectar clientes y servidores independientemente de sus plataformas.
- El concepto de escalabilidad tanto horizontal como vertical es aplicable a cualquier sistema Cliente-Servidor. La escalabilidad horizontal permite agregar más estaciones de trabajo activas sin afectar significativamente el rendimiento. La escalabilidad vertical permite mejorar las características del servidor o agregar múltiples servidores.

Ventajas

Existencia de plataformas de hardware cada vez más baratas. Esta constituye a su vez una de las más palpables ventajas de este esquema, la posibilidad de utilizar máquinas mucho más baratas que las requeridas por una solución centralizada, basada en sistemas grandes (mainframes). Además, se pueden utilizar componentes, tanto de hardware como de software, de varios fabricantes, lo cual contribuye considerablemente a la reducción de costos y favorece la flexibilidad en la implantación y actualización de soluciones.

- Facilita la integración entre sistemas diferentes y comparte información, permitiendo por ejemplo que las máquinas ya existentes puedan ser utilizadas pero utilizando interfaces más amigables el usuario. De esta manera, se puede integrar PCs con sistemas medianos y grandes, sin necesidad de que todos tengan que utilizar el mismo sistema operativo.
- Al favorecer el uso de interfaces gráficas interactivas, los sistemas construidos bajo este esquema tienen una mayor y más intuitiva con el usuario. En el uso de interfaces gráficas para el usuario, presenta la ventaja, con respecto a uno centralizado, de que no siempre es necesario transmitir información gráfica por la red pues esta puede residir en el cliente, lo cual permite aprovechar mejor el ancho de banda de la red.
- La estructura inherentemente modular facilita además la integración de nuevas tecnologías y el crecimiento de la infraestructura computacional, favoreciendo así la escalabilidad de las soluciones.
- Contribuye además a proporcionar a los diferentes departamentos de una organización, soluciones locales, pero permitiendo la integración de la información.

Desventajas

- El mantenimiento de los sistemas es más difícil pues implica la interacción de diferentes partes de hardware y de software, distribuidas por distintos proveedores, lo cual dificulta el diagnóstico de fallas.
- Cuenta con muy escasas herramientas para la administración y ajuste del desempeño de los sistemas.
- Es importante que los clientes y los servidores utilicen el mismo mecanismo (por ejemplo sockets o RPC), lo cual implica que se deben tener mecanismos generales que existan en diferentes plataformas.
- Hay que tener estrategias para el manejo de errores y para mantener la consistencia de los datos.

- El desempeño (performance), problemas de este estilo pueden presentarse por congestión en la red, dificultad de tráfico de datos, etc.

Patrón Maestro – Esclavo

El patrón de diseño Maestro/Esclavo es otra arquitectura fundamental que los desarrolladores de LabVIEW utilizan. Es usada cuando se tienen dos o más procesos que necesitan ejecutarse simultánea y continuamente, pero a diferentes velocidades. Si estos procesos corren en un único bucle, pueden suceder problemas de temporización severos. Estos problemas de temporización ocurren cuando una parte del bucle tarda más en ejecutarse de lo esperado. Si esto sucede, la sección restante del bucle se atrasa. El patrón Maestro/Esclavo consiste en múltiples bucles paralelos. Cada bucle puede ejecutar tareas a velocidades distintas. De estos bucles paralelos, un bucle actúa como el maestro y los otros como esclavos. El bucle maestro controla todos los bucles esclavos y se comunica con ellos utilizando arquitecturas de mensajería.

El patrón Maestro/Esclavo es comúnmente utilizado cuando se requiere responder a controles de interfaz de usuario y adquirir datos simultáneamente. Suponga que quiere escribir una aplicación que mida y registre una vez cada 5 segundos una tensión eléctrica que cambia lentamente. Que además adquiera una forma de onda de una línea de transmisión y la despliega en un gráfico cada 100ms y que también provea una interfaz de usuario que le permita al usuario cambiar los parámetros de cada adquisición.

El patrón de diseño Maestro/Esclavo es adecuado para esta aplicación. En esta aplicación, el bucle maestro contiene la interfaz de usuario. La adquisición de tensión eléctrica y el almacenamiento a disco se encuentran en el bucle esclavo, mientras que la adquisición de la línea de transmisión y su despliegue en pantalla suceden en otro bucle.

Aplicaciones que involucran control automático también se benefician del patrón de diseño Maestro/Esclavo. Un ejemplo es la interacción de un usuario con un brazo robótico de movimiento libre. Este tipo de aplicación está extremadamente orientada a control automático por el daño físico del brazo o sus alrededores que puede ocurrir si el control no se aplica adecuadamente. Por ejemplo, si el usuario le indica al brazo que detenga su movimiento hacia abajo, pero el programa está ocupado con el control de giro del brazo, el brazo robótico podría colisionar con la plataforma de soporte. Esta situación se puede evitar utilizando el patrón de diseño Maestro/Esclavo en la aplicación. En este caso, la interfaz de usuario será gestionada por el bucle maestro, y cada sección controlable del brazo robótico tendrá su propio bucle esclavo.

Utilizando este método, cada sección controlable del brazo tendrá su propio bucle, y por lo tanto, su debida cantidad de tiempo de procesamiento.

El patrón de diseño Maestro/Esclavo es muy ventajoso cuando creamos aplicaciones multi-tarea. Le da un enfoque más modular al desarrollo de la aplicación debido a su funcionalidad multi-bucle, pero más importante, le da un mejor control de la gestión de tiempo en su aplicación. En LabVIEW, cada bucle paralelo es tratado como una tarea o hilo separado. Un hilo se define como la parte de un programa que se puede ejecutar independientemente de las otras partes. Si tiene una aplicación que no utiliza hilos separados, esa aplicación se interpreta por el sistema como un hilo. Cuando separa su aplicación en múltiples hilos, cada uno comparte el tiempo de procesamiento por igual entre ellos.

Esto le da un mejor control de cómo se temporiza su aplicación y le da al usuario más control sobre su aplicación. La naturaleza paralela de LabVIEW se presta para la implementación de un patrón de diseño Maestro/Esclavo.

Para el ejemplo de adquisición de datos anterior, podríamos poner la medición de tensión eléctrica y la adquisición de forma de onda en el mismo bucle, y sólo realizar la medición de tensión cada quincuagésima iteración del bucle. Sin embargo, la medición de tensión y la archivación a disco de datos puede tardar más en completarse que una única adquisición y despliegue de la forma de onda. Si este es el caso, entonces la siguiente iteración de la forma de onda estará retrasada, puesto que no puede iniciar antes de que todo el código de la iteración anterior se complete. Adicionalmente, esta arquitectura haría difícil cambiar la tasa a la que se adquirió la forma de onda sin cambiar la tasa de almacenamiento a disco de la tensión eléctrica.

El enfoque común siguiendo un patrón de diseño Maestro/Esclavo en esta aplicación sería colocar los procesos de adquisición en dos bucles separados (bucles esclavos) ambos controlados por un bucle maestro que revisa los controles de la interfaz gráfica de usuario (GUI) para ver si algún parámetro ha cambiado. Para comunicarse con los bucles esclavos, el bucle maestro escribe a variables locales. Esto asegura que cada proceso de adquisición no afecte al otro, y que cualquier retardo causado por la interfaz de usuario (por ejemplo, abrir una ventana de diálogo) no retrasará ninguna iteración de los procesos de adquisición.

Patrón Tuberías y Filtros (Pipe and Filter)

En la ingeniería de software, un Pipe and Filter consta de una cadena de elementos de procesos (procesos, hilos, co-rutinas, etc.) dispuestos de manera que la salida de cada elemento es la entrada de la siguiente; el nombre es por analogía a una tubería física. Usualmente, una cierta cantidad de almacenamiento en bufer está dispuesto entre elementos consecutivos. La información que fluye en estas tuberías es a menudo una corriente de registros, bytes o bits; los elementos de un Pipe pueden ser llamados Filter. De ahí es que se conoce con el nombre de Pipe and Filter a este patrón de diseño.

Estos elementos se implementan a menudo en multitareas con el lanzamientos de todos los elementos al mismo tiempo como procesos, el mantenimiento de los datos leídos automáticamente de las peticiones de cada proceso con los datos escritos por el proceso encima - esto puede ser llamado un Pipe multiprocesos - de esta manera el CPU se cambiará naturalmente entre los procesos por el planificador con el fin de minimizar su tiempo de inactividad. En otros procesos comunes, los elementos se implementan con hilos ligeros o como co-rutinas para reducir la sobrecarga del sistema operativo a menudo involucrado en los procesos. Dependiendo del sistema operativo las discusiones pueden ser programadas directamente por el sistema operativo o por un hilo responsable. Co-rutinas siempre son programadas por un administrador.

Cuando usar este patrón

- El procesamiento requerido por una aplicación se puede dividir fácilmente en un conjunto de pasos independientes.
- Los pasos de procesamiento realizados por una aplicación tienen diferentes requisitos de escalabilidad.
- Es posible agrupar filtros que deberían escalarse juntos en el mismo proceso.
- Se requiere flexibilidad para permitir la reordenación de los pasos de procesamiento realizados por una aplicación, o la capacidad de agregar y quitar pasos.
- El sistema puede beneficiarse de distribuir el procesamiento para los pasos en diferentes servidores.
- Se requiere una solución confiable que minimice los efectos de la falla en un paso mientras se procesan los datos.

Este patrón puede no ser útil cuando:

- Los pasos de procesamiento realizados por una aplicación no son independientes, o deben realizarse juntos como parte de la misma transacción.
- La cantidad de información de contexto o estado requerida por un paso hace que este enfoque sea ineficiente. En su lugar, es posible conservar la información de estado en una base de datos, pero no use esta estrategia si la carga adicional en la base de datos causa una contención excesiva.

Patrón de Igual a Igual (Peer to Peer)

Se trata de una de arquitectura de red descentralizada y distribuida en la que nodos individuales en la red (llamados “pares”) actúan como proveedores y consumidores de recursos, en contraste con el modelo de cliente-servidor centralizado donde los nodos del cliente solicitan acceso a los recursos proporcionados por los servidores centrales.

Las tecnologías ‘peer to peer’ (P2P) hacen referencia a un tipo de arquitectura para la comunicación entre aplicaciones que permite a individuos comunicarse y compartir información con otros individuos sin necesidad de un servidor central que facilite la comunicación.

Este patrón no es aplicable si la aplicación o la información no se pueden distribuir en varios servidores. La escalabilidad de las aplicaciones y el acceso a la información debe ser examinada y gestionada, ya que esto puede ser un problema si se espera crecimiento, especialmente el número de pares que se comunican simultáneamente.

Es importante destacar que el término “P2P” se refiere a un tipo de arquitectura de aplicaciones y no a la funcionalidad específica de una aplicación final; es decir, la tecnología P2P es un medio para alcanzar un fin superior. Sin embargo, a menudo se utiliza el término “P2P” como sinónimo de “intercambio de archivos”, ya que éste es uno de los usos más populares de dicha tecnología. No obstante, existen muchos otros usos de la tecnología P2P, por ejemplo Skype utiliza una arquitectura P2P híbrida para ofrecer servicios VoIP, mientras que Tor utiliza una arquitectura P2P para ofrecer una funcionalidad de enrutamiento anónimo.

Ventajas

- Alta capacidad de almacenamiento: La información no se encuentra concentrada en un solo punto sino que está distribuida.
- Alta disponibilidad: Dado que la información se encuentra repartida, hay muchos sitios para poder descargarla, por lo que existe una gran probabilidad de conseguir lo que se quiera.
- Fiabilidad: Si falla un nodo, podemos seguir descargando.
- Distribución del tráfico en la red: La información no se solicita toda a un mismo punto, si no que se encuentra distribuida, por lo que los servidores no se saturarán.

Desventajas

- Los pares de nodos se desconectan después de descargar: De esta forma reciben datos pero no ceden contenidos por lo que la arquitectura fallará, ya que no hay de donde descargar.
- Presencia de los Firewalls: En ocasiones estos evitan que los usuarios reciban contenidos y que a su vez los cedan.
- Problemas de confianza: Debido que no hay seguridad centralizada, cada computadora debe utilizar medidas de seguridad individuales para la protección de los datos.

Características

- **Escalabilidad**. Las redes P2P tienen un alcance mundial con cientos de millones de usuarios potenciales. En general, lo deseable es que cuantos más nodos estén conectados a una red P2P mejor será su funcionamiento. Así, cuando los nodos llegan y comparten sus propios recursos, los recursos totales del sistema aumentan. Esto es diferente en una arquitectura del modo servidor-cliente con un sistema fijo de servidores, en los cuales la adición de más clientes podría significar una transferencia de datos más lenta para todos los usuarios.
- **Robustez**. La naturaleza distribuida de las redes peer-to-peer también incrementa la robustez en caso de haber fallos en la réplica excesiva de los datos hacia múltiples destinos, permitiendo a los Peers encontrar la información sin hacer peticiones a ningún servidor centralizado de indexado.
- **Descentralización**. Estas redes por definición son descentralizadas y todos los nodos son iguales. No existen nodos con funciones especiales, y por tanto ningún nodo es imprescindible para el funcionamiento de la red. En realidad, algunas redes

comúnmente llamadas P2P no cumplen esta característica, como Napster, EDonkey o BitTorrent.

- **Los costos están repartidos entre los usuarios**. Se comparten o donan recursos a cambio de recursos. Según la aplicación de la red, los recursos pueden ser Archivos, Ancho de banda, Ciclos de proceso o Almacenamiento de disco.
- **Anonimato**. Es deseable que en estas redes quede anónimo el autor de un contenido, el editor, el lector, el servidor que lo alberga y la petición para encontrarlo siempre que así lo necesiten los usuarios. Muchas veces el derecho al anonimato y los derechos de autor son incompatibles entre sí, y la industria propone mecanismos como el DRM para limitar ambos.
- **Seguridad**. Es una de las características deseables de las redes P2P menos implementada. Los objetivos de un P2P seguro serían identificar y evitar los nodos maliciosos, evitar el contenido infectado, evitar el espionaje de las comunicaciones entre nodos, creación de grupos seguros de nodos dentro de la red, protección de los recursos de la red… En su mayoría aún están bajo investigación, pero los mecanismos más prometedores son: Cifrado multiclave, cajas de arena, gestión de derechos de autor (la industria define qué puede hacer el usuario, por ejemplo la segunda vez que se oye la canción se apaga), reputación (sólo permitir acceso a los conocidos), comunicaciones seguras, comentarios sobre los ficheros…

Modelo-Vista-Controlador (MVC)

MVC (por sus siglas en inglés) es un patrón de diseño de arquitectura de software usado principalmente en aplicaciones que manejan gran cantidad de datos y transacciones complejas donde se requiere una mejor separación de conceptos para que el desarrollo este estructurado de una mejor manera, facilitando la programación en diferentes capas de manera paralela e independiente. MVC sugiere la separación del software en tres estratos: Modelo, Vista y Controlador.

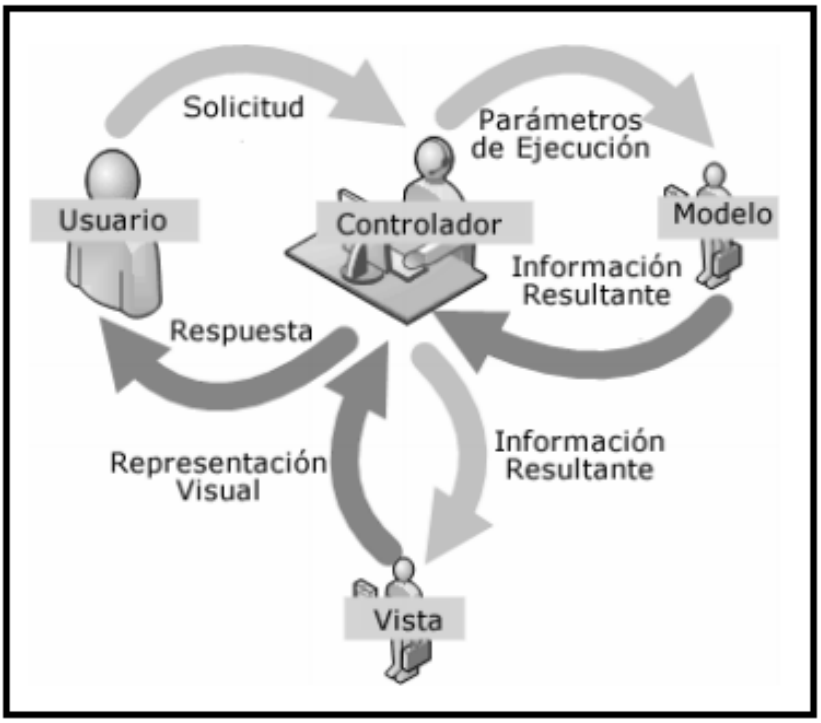
Modelo: es la representación de la información que maneja la aplicación. El modelo en si son los datos puros que puestos en contexto del sistema proveen de información al usuario o a la aplicación misma.

Vista: es la representación del modelo en forma gráfica disponible para la interacción con el usuario. En el caso de una aplicación Web, la “vista” es una página HTML con contenido dinámico sobre el cual el usuario puede realizar operaciones.

Controlador: es la capa encargada de manejar y responder las solicitudes del usuario, procesando la información necesaria y modificando el modelo en caso de ser necesario

Ciclo de vida de MVC

El ciclo de vida de MVC es normalmente representado por las tres capas presentadas anteriormente y el cliente (también conocido como usuario). El siguiente diagrama representa el ciclo de vida de manera sencilla:



En el primer paso en el ciclo de vida empieza cuando el usuario hace una solicitud al controlador con información sobre lo que el usuario desea realizar. Entonces el controlador decide a quien debe delegar la tarea y es aquí donde el modelo comienza su trabajo. En esta etapa, el modelo se encarga de realizar operaciones sobre la información que maneja para cumplir con lo que le solicita el controlador. Una vez que termina su labor, le regresa al controlador la información resultante de sus operaciones, el cual a su vez redirige a la vista. La vista se encarga de transformar los datos en información visualmente entendible para el usuario. Finalmente, la representación gráfica es transmitida de regreso al controlador y este se encarga de transmitírsela al usuario. El ciclo entero puede empezar nuevamente si el usuario así lo requiere.

Ventajas

- La separación del modelo de la vista, es decir, separar los datos de la representación visual de los mismos.
- Es mucho más sencillo agregar múltiples representaciones de los mismos datos o información.
- Facilita agregar nuevos tipos e datos según sea requerido por la aplicación ya que son independientes del funcionamiento de las otras capas
- Crea independencia de funcionamiento
- Facilita el mantenimiento en caso de errores.
- Ofrece maneras más sencillas para probar el correcto funcionamiento del sistema.
- Permite el escalamiento de la aplicación en caso de ser requerido.

Desventajas

- La separación de conceptos en capas agrega complejidad al sistema.
- La cantidad de archivos a mantener y desarrollar se incrementa considerablemente.
- La curva de aprendizaje del patrón de diseño es más alta que cuando usamos otros modelos mas sencillos

Interacción de los componentes

Se pueden encontrar muchas implementaciones de MVC, pero generalmente el flujo de datos se describe así:

1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace, etc.).

2. El controlador recibe (por parte de los objetos de la interfaz-vista) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega, frecuentemente a través de un gestor de eventos (handler o callback).

3. El controlador accede al modelo, actualizándolo, posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario. Los controladores complejos están a menudo estructurados usando un patrón de comando que encapsula las acciones y simplifica su extensión.

4. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se

reflejan los cambios en el modelo. El modelo no debe tener conocimiento directo sobre la vista. Sin embargo, se podría utilizar el patrón Observador para proveer cierta indirección entre el modelo y la vista, permitiendo al modelo notificar a los interesados de cualquier cambio.

5. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.

MVC y bases de datos

Muchos sistemas informáticos utilizan un Sistema de Gestión de Base de Datos para gestionar los datos que debe utilizar la aplicación; en líneas generales del MVC dicha gestión corresponde al modelo. La unión entre capa de presentación y capa de negocio conocido en el paradigma de la Programación por capas representaría la integración entre la Vista y su correspondiente Controlador de eventos y acceso a datos, MVC no pretende discriminar entre capa de negocio y capa de presentación pero si pretende separar la capa visual gráfica de su correspondiente programación y acceso a datos, algo que mejora el desarrollo y mantenimiento de la Vista y el Controlador en paralelo, ya que ambos cumplen ciclos de vida muy distintos entre sí.

Uso en aplicaciones Web

Aunque originalmente MVC fue desarrollado para aplicaciones de escritorio, ha sido ampliamente adaptado como arquitectura para diseñar e implementar aplicaciones web en los principales lenguajes de programación. Se han desarrollado multitud de frameworks, comerciales y no comerciales, que implementan este patrón; estos frameworks se diferencian básicamente en la interpretación de como las funciones MVC se dividen entre cliente y servidor.

Los primeros frameworks MVC para desarrollo web planteaban un enfoque de cliente ligero en el que casi todas las funciones, tanto de la vista, el modelo y el controlador recaían en el servidor. En este enfoque, el cliente manda la petición de cualquier hiperenlace o formulario al controlador y después recibe de la vista una página completa y actualizada (u otro documento); tanto el modelo como el controlador (y buena parte de la vista) están completamente alojados en el servidor.

- **Vista:** la página HTML.

- **Controlador**: código que obtiene los datos dinámicamente y genera el contenido HTML.

- **Modelo**: la información almacenada en base de datos o en XML. 10

Frameworks

Es un término utilizado en la computación en general, para referirse a un conjunto de bibliotecas utilizadas para implementar la estructura estándar de una aplicación. Todo esto es realizado con el propósito de promover la reutilización de código, con el fin de ahorrarse trabajo al desarrollador al no tener que rescribir ese código para cada nueva aplicación que se desea crear. Existen varios FrameWorks para diferentes fines, algunos son orientados para aplicaciones web, otros para aplicaciones multiplataforma, sistemas operativos, etc.

Este determina la arquitectura de una aplicación, se encarga de definir la estructura general, sus particiones en clases y objetos, responsabilidades clave, así como la colaboración entre las clases objetos, esto evita que el usuario tenga que definirlo y se pueda enfocar en cosas específicas de su aplicación.

Los Frameworks utilizan un variado número de patrones de diseño, ya que así logran soportar aplicaciones de más alto nivel y que reutilizan una mayor cantidad de código, que uno que no utiliza dichos patrones. "Los patrones ayudan hacer la arquitectura de los FrameWorks más adecuada para muchas y diferentes aplicaciones sin necesidad de rediseño". Por esta razón es importante que se documenten que patrones utiliza el FrameWorks para que los que se encuentren familiarizados con dichos patrones puedan tener una mejor visión y poder adentrarse en el FrameWorks mas fácilmente.

El patrón MVC es utilizado en múltiples Frameworks como:

- Java Swing, Java Enterprise Edition

- XForms (formato XML estándar para la especificación de un modelo de proceso de datos XML e interfaces de usuario como formularios web).

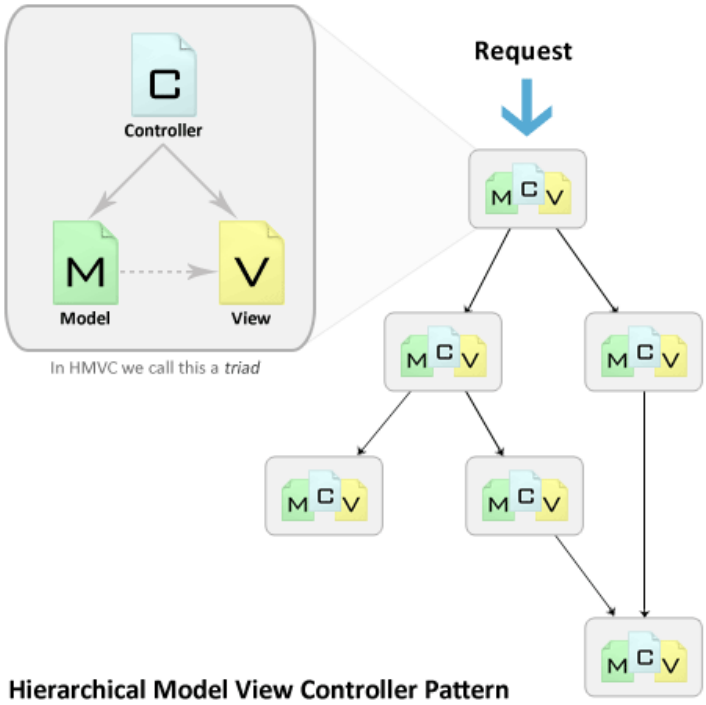
- GTK+ (escrito en C, toolkit creado por Gnome para construir aplicaciones gráficas, inicialmente para el sistema X Window)

- Google Web Toolkit, Apache Struts, Ruby On Rails, entre otros.

Evolución de MVC

HMVC (MVC Jerárquico)

HMVC es una evolución del patrón MVC utilizado para la mayoría de las aplicaciones web en la actualidad. Surgió como una respuesta a los problemas de viabilidad aparentes dentro de las aplicaciones que usaban MVC. La solución presentada en el sitio web de JavaWorld, julio de 2000, propuso que el modelo estándar, la vista y la tríada del controlador se superpongan en una "jerarquía de capas MCV padre-hijo". La imagen a continuación ilustra cómo funciona esto:



Hierarchical Model View Controller Pattern

Cada tríada funciona independientemente una de la otra. Una tríada puede solicitar acceso a otra tríada a través de sus controladores. Ambos puntos permiten que la aplicación se distribuya en varias ubicaciones, si es necesario. Además, la estratificación de triadas de MVC permite un desarrollo de aplicaciones más profundo y robusto. Esto nos lleva a varias ventajas que nos llevan a nuestro siguiente punto.

¿Por qué usar HMVC?

Principales ventajas de implementar el patrón HMVC en tu ciclo de desarrollo:

- **Modularizacion**: Reducción de dependencias entre las partes dispares de la aplicación.
- **Organización**: Tener una carpeta para cada una de las triadas relevantes hace que la carga de trabajo sea más ligera.

- **Reutilización**: Por la naturaleza del diseño, es fácil reutilizar casi cada pieza de código.
- **Extensibilidad**: Hace que la aplicación sea más extensible sin sacrificar la facilidad de mantenimiento.

MVA (Modelo, Vista, Adaptador)

El modelo-vista-adaptador (MVA) o controlador de mediación MVC es un patrón arquitectónico de software y una arquitectura multinivel. En aplicaciones informáticas complejas que presentan grandes cantidades de datos a los usuarios, los desarrolladores a menudo desean separar los datos (modelo) y las preocupaciones de la interfaz de usuario (vista) para que los cambios en la interfaz de usuario no afecten el manejo de datos y que los datos se puedan reorganizar sin cambiar La interfaz de usuario. Tanto MVA como MVC tradicional intentan resolver este mismo problema, pero con dos estilos diferentes de solución. MVC tradicional organiza el modelo (por Ej., Estructuras de datos y almacenamiento), vista (por Ej., Interfaz de usuario) y controlador (por Ej., Lógica de negocios) en un triángulo, con el modelo, la vista y el controlador como vértices, de modo que parte de la información fluya entre modelo y vistas fuera del control directo del controlador. El modelo-vista-adaptador resuelve esto de manera bastante diferente del modelo-vista-controlador organizando el modelo, adaptador o controlador de mediación y visualiza linealmente sin ninguna conexión directa entre el modelo y la vista.

MVP (Modelo, Vista, Presentador)

El modelo-vista-presentar (MVP) es una derivación del patrón arquitectónico modelo-vista-controlador (MVC), y es utilizado mayoritariamente para construir interfaces de usuario. En MVP el presentador asume la funcionalidad del “medio- hombre”. En MVP, toda lógica de presentación es colocada al presentador.

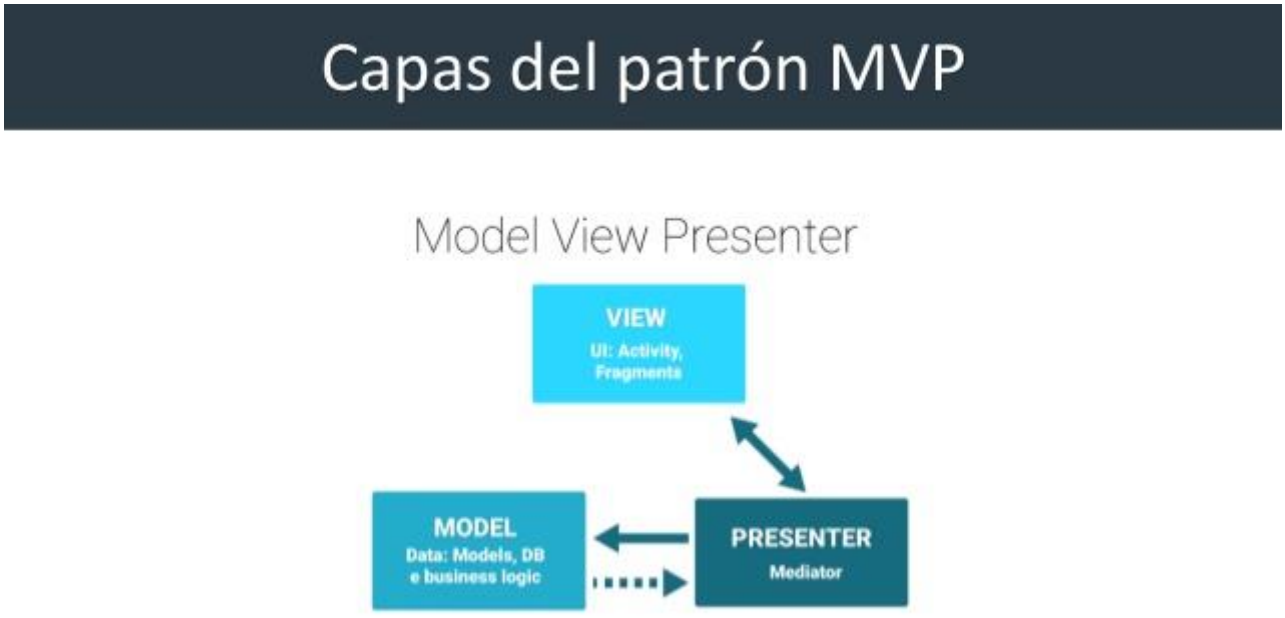
En el MVC, el modelo notifica a la vista cualquier cambio que sufra el estado del modelo. La información puede pasarse en la propia notificación, o después de la notificación, la vista puede consultar el modelo directamente para obtener los datos actualizados. Por el contrario, en el MVP, la vista no sabe nada sobre el modelo y la función del presentador es la de mediar entre ambos, enlazando los datos con la vista. 2. En el modelo MVC, la vista tiende a tener más lógica porque es responsable de manejar las notificaciones del modelo y de procesar los datos. En el modelo MVP, esa lógica se encuentra en el presentador, haciendo a la vista “estúpida”. Su única función es representar la información que el presentador le ha proporcionado. 3. En MVC, el modelo tiene lógica extra para interactuar con la vista. En el MVP, esta lógica se encontraría en el presentador.

Ejemplo • El usuario introduce su usuario y clave.

• La Vista ejecuta el método del presentador a hacer Login.

• El Presentador hacer la llamada al ApiCliente.

• Si todo ha ido bien, el Presentador notifica a la vista que el login fue exitoso. En Caso de que se haya producido algún error, muestra una alerta con un mensaje de error.



MVVM (Modelo, Vista, Vista-Modelo)

El patrón Model-View-ViewModel es muy popular con los Frameworks que soportan la vinculación de datos (data-binding), tal como Xamarin.Forms. Fue popularizado por SDKs habilitados para XAML como Windows Presentation Foundation (WPF) y Silverlight donde el ViewModel actúa como un intermediario entre los datos (Model) y la interfaz de usuario (View) a través de enlace de datos y comandos.

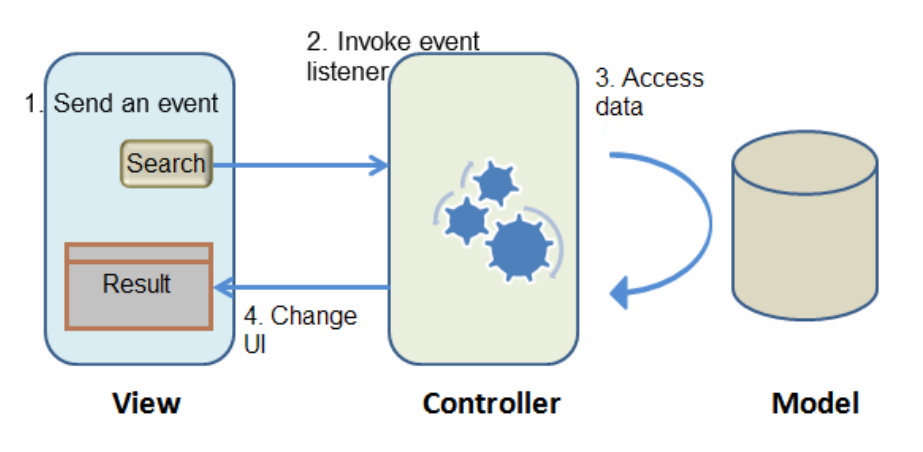
MVVM significa Modelo Vista VistaModelo, porque en este patrón de diseño se separan los datos de la aplicación, la interfaz de usuario pero en vez de controlar manualmente los cambios en la vista o en los datos, estos se actualizan directamente cuando sucede un cambio en ellos, por ejemplo si la vista actualiza un dato que está presentando se actualiza el modelo automáticamente y viceversa.

Comparación entre MVC y MVVM

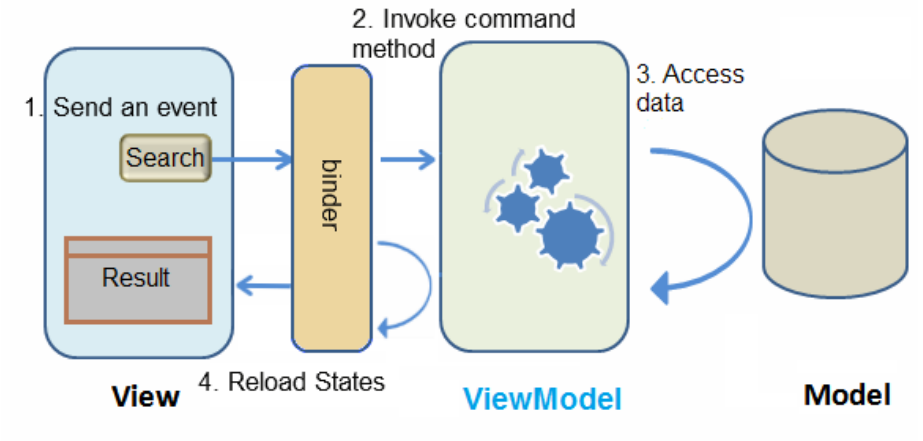
La primera imagen describe la interacción que se produce con MVC en ZK, y la siguiente es la correspondiente al enfoque MVVM que hace ZK.

Las principales diferencias entre MVC y MVVM son que en MVVM el «**Controller**» cambia a «**ViewModel**» y hay un «**binder**» que sincroniza la información en vez de hacerlo un controlador «**Controller**» como sucede en MVC.

MVC



MVVM



Los dos enfoques tienen muchas cosas en común, pero también hay claras diferencias entre ellos. Cada uno de los 2 enfoques tiene su razón de ser/existir.

¿Cuál debo usar?

Construir una aplicación mediante **MVC** puede resultar más intuitivo, porque directamente controlas lo que ves en la vista y su comportamiento, es decir manualmente. MVC se caracteriza porque tienes control total de los componentes, por lo tanto puedes crear componentes hijo dinámicamente («child»), controlar componentes propios personalizados, o realizar cualquier cosa sobre el componente que este pueda hacer por sí mismo.

En el patrón **MVVM**, puesto que la capa «ViewModel» esta débilmente acoplada con la vista (no está referenciada a los componentes de la vista), podemos usarla con múltiples vistas sin tener que modificarla. Por lo tanto los diseñadores y programadores pueden trabajar en paralelo. Si la información y comportamiento no cambian, un cambio en la vista no provoca que se tenga que modificar la capa «ViewModel». Además, como la capa «ViewModel» es un POJO (Plain Old Java Object, uso de clases simples y que no dependen de un framework en especial), es fácil realizar tests unitarios sobre ella sin ninguna configuración ni entorno especial. Lo que significa que la capa «ViewModel» tiene una mejor reusabilidad, testabilidad, y los cambios en la vista le afectan menos.

Para terminar, comparamos los 2 patrones de diseño en una tabla:

	MVC	**MVVM**
Acoplamiento con la vista	Muy poco con plantilla	Muy poco
Acoplamiento con el componente	Un poco	Muy poco
Codificar en la vista	Mediante el ID del componente	A través de una expresión Data binding
Implementación de un controlador	Extendemos ZK's Composer	Es un POJO
Acceso a la información de la UI	Acceso directo	Automático

Acceso a la información desde el backend	Acceso directo	Acceso directo
Actualización de la interfaz de usuario	Manipulamos directamente los componentes	Automático (@NotifyChange)
Nivel de control del componente	Elevado, control total	Normal
Rendimiento	Alto	Normal

PATRONES DE DISEÑO

Una cosa que los diseñadores expertos no hacen es resolver cada problema desde el principio. A menudo reutilizan las soluciones que ellos han obtenido en el pasado. Cuando encuentran una buena solución, utilizan esta solución una y otra vez. Consecuentemente podrías encontrar patrones de clases y comunicaciones entre objetos en muchos sistemas orientados a objetos. Estos patrones solucionan problemas específicos del diseño y hacen los diseños orientados a objetos más flexibles, elegantes y por último reutilizables. Los patrones de diseño ayudan a los diseñadores a reutilizar con éxito diseños para obtener nuevos diseños. Un diseñador que conoce algunos patrones puede aplicarlos inmediatamente a problemas de diseño sin tener que descubrirlos.

El valor de la experiencia en los diseños es muy importante. Cuando tienes que hacer un diseño que crees que ya has solucionado antes pero no sabes exactamente cuándo o como lo hiciste. Si pudieras recordar los detalles de los problemas anteriores y como los solucionaste, entonces podrías reutilizar la experiencia en vez de tener que volver a pensarlo.

El objetivo de los patrones de diseño es guardar la experiencia en diseños de programas orientados a objetos. Cada patrón de diseño nombra, explica y evalúa un importante diseño en los sistemas orientados a objetos. Es decir se trata de agrupar la experiencia en diseño de una forma que la gente pueda utilizarlos con efectividad. Por eso se han documentado los más importantes patrones de diseño y presentado en catálogos.

Los patrones de diseño hacen más fácil reutilizar con éxito los diseños y arquitecturas. Expresando estas técnicas verificadas como patrones de diseño se hacen más accesibles para los

diseñadores de nuevos sistemas. Los patrones de diseño te ayudan a elegir diseños alternativos que hacen un sistema reutilizable y evitan alternativas que comprometan la reutilización. Los patrones de diseño pueden incluso mejorar la documentación y mantenimiento de sistemas existentes. Es decir, los patrones de diseño ayudan a un diseñador a conseguir un diseño correcto rápidamente.

¿Qué es un Patrón de Diseño?

Christopher Alexander dice: *"Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces, sin hacerlo ni siquiera dos veces de la misma forma"*.

Los trabajos de Alexander intentan identificar y resolver, en un marco descriptivo formal aunque no exacto, problemas esenciales en el dominio de la arquitectura. A los diseñadores de software les ha parecido trasladar muchas de las ideas de Alexander al dominio software. En software las soluciones son expresadas en términos de objetos e interfaces en lugar de paredes y puertas, pero el núcleo de ambos tipos de patrones es una solución a un problema en un contexto.

Alexander ha servido, en realidad, de catalizador de ciertas tendencias "constructivas" utilizadas en el diseño de sistemas software.

En general, un patrón tiene cuatro elementos esenciales:

1. El nombre del patrón se utiliza para describir un problema de diseño, su solución, y consecuencias en una o dos palabras. Nombrar un patrón incrementa inmediatamente nuestro vocabulario de diseño. Esto nos permite diseños a un alto nivel de abstracción. Tener un vocabulario de patrones nos permite hablar sobre ellos con nuestros amigos, en nuestra documentación, e incluso a nosotros mismos.
2. El problema describe cuando aplicar el patrón. Se explica el problema y su contexto. Esto podría describir problemas de diseño específicos tales como algoritmos como objetos. Podría describir estructuras de clases u objetos que son sintomáticas de un diseño inflexible. Algunas veces el problema incluirá una lista de condiciones que deben cumplirse para poder aplicar el patrón.
3. La solución describe los elementos que forma el diseño, sus relaciones, responsabilidades y colaboraciones. La solución no describe un diseño particular o

implementación, porque un patrón es como una plantilla que puede ser aplicada en diferentes situaciones. En cambio, los patrones proveen una descripción abstracta de un problema de diseño y como una disposición general de los elementos (clases y objetos en nuestro caso) lo soluciona.

4. Las consecuencias son los resultados de aplicar el patrón. Estas son muy importantes para la evaluación de diseños alternativos y para comprender los costes y beneficios de la aplicación del patrón.

Los patrones de diseño tienen un cierto nivel de abstracción. Los patrones de diseño no son diseños tales como la realización de listas y tablas hash que pueden ser codificadas en clases y reutilizadas. Un algoritmo puede ser un ejemplo de implementación de un patrón, pero es demasiado incompleto, específico y rígido para ser un patrón. Una regla o heurística puede participar en los efectos de un patrón, pero un patrón es mucho más. Los patrones de diseño son descripciones de las comunicaciones de objetos y clases que son personalizadas para resolver un problema general de diseño en un contexto particular.

Un patrón de diseño nombra, abstrae e identifica los aspectos clave de un diseño estructurado, común, que lo hace útil para la creación de diseños orientados a objetos reutilizables. Los patrones de diseño identifican las clases participantes y las instancias, sus papeles y colaboraciones, y la distribución de responsabilidades. Cada patrón de diseño se enfoca sobre un particular diseño orientado a objetos. Se describe cuando se aplica, las características de otros diseños y las consecuencias y ventajas de su uso.

Los patrones de diseño se pueden utilizar en cualquier lenguaje de programación orientado a objetos, adaptando los diseños generales a las características de la implementación particular.

Descripción de Patrones de Diseño

¿Cómo describimos los patrones de diseño? Las notaciones gráficas, aunque importantes y útiles, no son suficientes. Estás solo recogen el producto final del proceso de diseño como las relaciones entre clases y objetos. Para reutilizar el diseño, nosotros debemos también guardar las decisiones, alternativas y ventajas que nos llevaron a ese diseño. Los ejemplos concretos son también muy importantes, porque ellos nos ayudan a ver el funcionamiento del diseño.

Para describir los patrones de diseño se utiliza un formato consistente. Cada patrón es dividido en secciones de acuerdo con la siguiente plantilla. La plantilla nos muestra una estructura uniforme para la información, de tal forma que los patrones de diseño sean fáciles de aprender, comparar y utilizar.

1. Nombre del patrón

Esta sección consiste de un nombre del patrón y una referencia bibliografía que indica de donde procede el patrón. El nombre es significativo y corto, fácil de recordar y asociar a la información que sigue.

2. Objetivo

Esta sección contiene unas pocas frases describiendo el patrón. El objetivo aporta la esencia de la solución que es proporcionada por el patrón. El objetivo está dirigido a programadores con experiencia que pueden reconocer el patrón como uno que ellos ya conocen, pero para el cual ellos no le han dado un nombre. Después de reconocer el patrón por su nombre y objetivo, esto podría ser suficiente para comprender el resto de la descripción del patrón.

3. Contexto

La sección de Contexto describe el problema que el patrón soluciona. Este problema suele ser introducido en términos de un ejemplo concreto. Después de presentar el problema en el ejemplo, la sección de Contexto sugiere una solución de diseño a ese problema.

4. Aplicabilidad

La sección Aplicabilidad resume las consideraciones que guían a la solución general presentada en la sección Solución. En que situaciones es aplicable el patrón.

5. Solución

La sección Solución es el núcleo del patrón. Se describe una solución general al problema que el patrón soluciona. Esta descripción puede incluir, diagramas y texto que identifique la estructura del patrón, sus participantes y sus colaboraciones para mostrar cómo se soluciona el problema. Debe describir tanto la estructura dinámica como el comportamiento estático.

6. Consecuencias

La sección Consecuencias explica las implicaciones, buenas y malas, del uso de la solución.

7. Implementación

La sección de Implementación describe las consideraciones importantes que se han de tener en cuenta cuando se codifica la solución. También puede contener algunas variaciones o simplificaciones de la solución.

8. Patrones relacionados

Esta sección contiene una lista de los patrones que están relacionados con el patrón que se describe.

Cualidades de un Patrón de Diseño

En adición a los elementos mencionados anteriormente, un buen escritor de patrones expuso varias cualidades deseables. Doug Lea, en su libro *"Christopher Alexander: an Introduction for Object-Oriented Designers"* muestra una descripción detallada de estas cualidades, las cuales son resumidas a continuación:

- **Encapsulación y abstracción:** cada patrón encapsula un problema bien definido y su solución en un dominio particular. Los patrones deberían de proporcionar límites claros que ayuden a cristalizar el entorno del problema y el entorno de la solución empaquetados en un entramado distinto, con fragmentos interconectados. Los patrones también sirven como abstracciones las cuales contienen dominios conocidos y experiencia, y podrían ocurrir en distintos niveles jerárquicos de granularidad conceptual.
- **Extensión y variabilidad:** cada patrón debería ser abierto por extensión o parametrización por otros patrones, de tal forma que pueden aplicarse juntos para solucionar un gran problema. Un patrón solución debería ser también capaz de realizar una variedad infinita de implementaciones (de forma individual, y también en conjunción con otros patrones).
- **Generatividad y composición:** cada patrón, una vez aplicado, genera un contexto resultante, el cual concuerda con el contexto inicial de uno o más de uno de los patrones del catálogo. Está subsecuencia de patrones podría luego ser aplicada progresivamente para conseguir el objetivo final de generación de un "todo" o solución completa. Los patrones son aplicados por el principio de evolución fragmentada. Pero los patrones no son simplemente de naturaleza lineal, más bien esos patrones en un nivel particular de abstracción y granularidad podrían guiar hacia o ser compuestos con otros patrones para modificar niveles de escala.
- **Equilibrio:** cada patrón debe realizar algún tipo de balance entre sus efectos y restricciones. Esto podría ser debido a uno o más de un heurístico que son utilizados para minimizar el conflicto sin el contexto de la solución. Las invariaciones representadas en un problema subyacente solucionan el principio o filosofía para el

dominio particular, y proveen una razón fundamental para cada paso o regla en el patrón.

El objetivo es este, si está bien escrito, cada patrón describe un todo que es más grande que la suma de sus partes, debido a la acertada coreografía de sus elementos trabajando juntos para satisfacer todas sus variaciones reclamadas.

Clasificación de los Patrones de Diseño

Dado que hay muchos patrones de diseño necesitamos un modo de organizarlos. En esta sección clasificamos los patrones de diseño de tal forma que podamos referirnos a familias de patrones relacionados. La clasificación nos ayuda a saber lo que hace un patrón. Según el libro *"Patterns in Java (Volume 1)"* existen seis categorías:

Patrones de Diseño Fundamentales

Los patrones de esta categoría son los más fundamentales e importantes patrones de diseño conocidos. Estos patrones son utilizados extensivamente en otros patrones de diseño.

Patrones de Creación

Los patrones de creación muestran la guía de cómo crear objetos cuando sus creaciones requieren tomar decisiones. Estas decisiones normalmente serán resueltas dinámicamente decidiendo que clases instanciar o sobre que objetos un objeto delegará responsabilidades. La valía de los patrones de creación nos dice como estructurar y encapsular estas decisiones.

A menudo hay varios patrones de creación que puedes aplicar en una situación. Algunas veces se pueden combinar múltiples patrones ventajosamente. En otros casos se debe elegir entre los patrones que compiten. Por estas razones es importante conocer los seis patrones descritos en esta categoría.

Patrones de Partición

En la etapa de análisis, tu problema para identificar los actores, casos de uso, requerimientos y las relaciones que constituyen el problema. Los patrones de esta categoría proveen la guía sobre como dividir actores complejos y casos de uso en múltiples clases.

Patrones Estructurales

Los patrones de esta categoría describen las formas comunes en que diferentes tipos de objetos pueden ser organizados para trabajar unos con otros.

Patrones de Comportamiento

Los patrones de este tipo son utilizados para organizar, manejar y combinar comportamientos.

Patrones de Concurrencia

Los patrones de esta categoría permiten coordinar las operaciones concurrentes. Estos patrones se dirigen principalmente a dos tipos diferentes de problemas:

1. **Recursos compartidos:** Cuando las operaciones concurrentes acceden a los mismos datos u otros tipos de recursos compartidos, podría darse la posibilidad de que las operaciones interfirieran unas con otras si ellas acceden a los recursos al mismo tiempo. Para garantizar que cada operación se ejecuta correctamente, la operación debe ser protegida para acceder a los recursos compartidos en solitario. Sin embargo, si las operaciones están completamente protegidas, entonces podrían bloquearse y no ser capaces de finalizar su ejecución.

El bloqueo es una situación en la cual una operación espera por otra para realizar algo antes de que esta proceda. Porque cada operación esta esperando por la otra para hacer algo, entonces ambas esperan para siempre y nunca hacen nada.

2. **Secuencia de operaciones:** Si las operaciones son protegidas para acceder a un recurso compartido una cada vez, entonces podría ser necesario garantizar que ellas acceden a los recursos compartidos en un orden particular. Por ejemplo, un objeto nunca será borrado de una estructura de datos antes de que esté sea añadido a la estructura de datos.

<table>
<tr><th>Fundamentales</th><th>De Creacion</th><th>De Particion</th><th>Estructurales</th><th>De Comportamiento</th><th>De Concurrencia</th></tr>
<tr><td>Delegation</td><td>Factory Method</td><td>Layered Initialization</td><td>Adapter</td><td>Chain of Responsability</td><td>Single Threaded Execution</td></tr>
<tr><td>Interface</td><td>Abstract Factory</td><td>Filter</td><td>Iterator</td><td>Command</td><td>Guarded Suspension</td></tr>
<tr><td>Immutable</td><td>Builder</td><td rowspan="8">Composite</td><td>Bridge</td><td>Little Language</td><td>Balking</td></tr>
<tr><td>Marker Interface</td><td>Prototype</td><td>Fecade</td><td>Mediator</td><td>Scheduler</td></tr>
<tr><td rowspan="7">Proxy</td><td>Singleton</td><td>Flyweight</td><td>Snapshot</td><td>Read/Write Lock</td></tr>
<tr><td rowspan="6">Object Pool</td><td>Dynamic Linkage</td><td>Observer</td><td>Producer-Consumer</td></tr>
<tr><td>Virtual Proxy</td><td>State</td><td rowspan="6">Two-Phase Termination</td></tr>
<tr><td>Decorator</td><td>Null Object</td></tr>
<tr><td rowspan="3">Cache Management</td><td>Strategy</td></tr>
<tr><td>Template Method</td></tr>
<tr><td>Visitor</td></tr>
</table>