Unidad 6 – POSA y MVC

Introducción y orientaciones para el estudio

En el desarrollo de este módulo abordaremos:

- Introducción al concepto de Patrones de Arquitectura
- Presentación de la Arquitectura Orientada a Patrones de Software (POSA)
- Desarrollo del Patrón de Arquitectura MVC (Modelo Vista Controlador).

Objetivos

Pretendemos que al finalizar de estudiar esta unidad el alumno logre:

- Comprender qué son los Patrones de Arquitectura.
- Entender las generalidades de cada uno, sus ventajas y desventajas
- Entender el patrón MVC, sus variantes y beneficios.

Aclaraciones previas al estudio

En este módulo, el alumno encontrará:

- Contenidos
- Conceptualizaciones centrales

Se debe tener presente que los contenidos presentados en el módulo no ahondan profundamente en el tema, sino que pretenden ser un recurso motivador para que, a través de la lectura del material, la bibliografía sugerida y el desarrollo de las actividades propuestas se alcancen finalmente los objetivos planteados.

Cada módulo constituye una guía cuya finalidad es facilitar el aprendizaje.

Introducción

Antes de que comience un importante desarrollo de software, debemos elegir una arquitectura adecuada que nos proporcione funcionalidad deseada de acuerdo a los atributos de calidad (que son los requisitos no funcionales que el cliente prioriza o simplemente los que la solución necesita).

Los atributos de calidad aparecen durante la etapa de análisis (en la recolección de requerimientos) y en las etapas tempranas del diseño. Por lo tanto, debemos entender diferentes las arquitecturas, antes de aplicarlas al diseño de nuestra aplicación.

A continuación hablaremos resumidamente de estilos arquitectónicos y/o de POSA (o Pattern Oriented Software Arquitect, por sus siglas en inglés) como introducción necesaria para explicar el patrón MVC (tema más central de esta unidad y apunte).

Arquitecturas Orientadas en Patrones (POSA)

Los **patrones de arquitectura**, (otros les dicen estilos arquitectónicos) ofrecen soluciones a problemas de arquitectura de software dentro de un estilo arquitectónico determinado en la ingeniería de software, mas recientemente se hace mención a la sigla POSA en Inglés.

Dan una **descripción y describen elementos**, tienen **restricciones** sobre cómo pueden ser usados y de cómo se comunican o relacionan entre sí. Un patrón de arquitectura **expresa un esquema u organización esencial** para un sistema de software, que consta de subsistemas, responsabilidades e interrelaciones entre sí.

En comparación con los patrones de diseño, los patrones de arquitectura **tienen un nivel de abstracción mayor,** definiendo muchas veces un estilo. Aunque un patrón arquitectónico comunica una imagen de un sistema**, no siempre representa una arquitectura en sí misma**. Un patrón de arquitectura es más un concepto que captura elementos esenciales de una arquitectura o estilo arquitectónico.

Muchas arquitecturas, o estilos diferentes pueden implementar el mismo patrón y por lo tanto compartir características. Además, los patrones son a menudo definidos como una cosa **estrictamente descrita** y **comúnmente disponible**.

¿Qué es un Patrón de Arquitectura?

Es una solución general y reutilizable a un problema común (como los patrones de diseño que ya hemos visto) pero para una arquitectura de software en un contexto dado. Los patrones de arquitectura son similares a los de diseño pero tienen un alcance más amplio. En el listado de abajo, los diez patrones más usados. En el anexo un detalle sobre cada uno. Seguidamente el desarrollo del patrón MVC (Modelo Vista Controlador).

Patrones de Arquitectura comúnmente usados

1. Patrón de capas
2. Patrón cliente-servidor
3. Patrón maestro-esclavo
4. Patrón de filtro de tubería
5. Patrón de intermediario
6. Patrón de igual a igual
7. Patrón de bus de evento
8. Modelo-vista-controlador
9. Patrón de pizarra
10. Patrón de intérprete

El patrón MVC (Modelo Vista Controlador)

El patrón MVC (por Model-View-Controler por su acrónimo en inglés) fue una de las primeras ideas formalizadas en el campo del manejo de las interfaces gráficas de usuario. Este patrón fue progresando desde las primeras propuestas organizativas de componentes, transformándose en un patrón de arquitectura o un estilo arquitectónico en sí mismo.

El patrón promueve una organización de componentes en 3 capas, fomentando una economía para el mantenimiento, mejorando la cambiabilidad y permitiendo la reutilización de artefactos dentro de una aplicación o entre aplicaciones de la misma tecnología.

Al ser anterior al concepto de patrones, el concepto se ha ido reinterpretando sin perder su filosofía o propósito de origen. Algunos autores o proveedores de tecnología han propuesto o evolucionado el mismo en variantes, las que mencionaremos (las más difundidas) más adelante en este apunte.

En las primeras definiciones de MVC, se decía que un controlador (o controler) era “el que se ocupa de manejar la entrada”, y de forma similar se decía que la vista o los componentes de la vista (o view) era “la que se ocupa de lo que se mostraba en la salida”, quedando medio el modelo (o model) que se ocupaba de las lógicas de negocio (y/o acceso a datos) el cual era accedido o consultado por las anteriores.

Esas primeras definiciones han quedado parcialmente obsoletas, o se han ido adecuando dado que en aplicaciones modernas éstas funciones son asumidas por una combinación de componentes en cada capa dentro de algún framework de desarrollo propio de los lenguajes o externo a ellos según la tecnología usada.

En aplicaciones modernas (2000 en adelante), el 'controlador' representa una sección intermedia de código, que intermedia la comunicación entre el 'modelo' y la 'vista', permitiendo validaciones, utilizando llamadas directas a otros componentes.

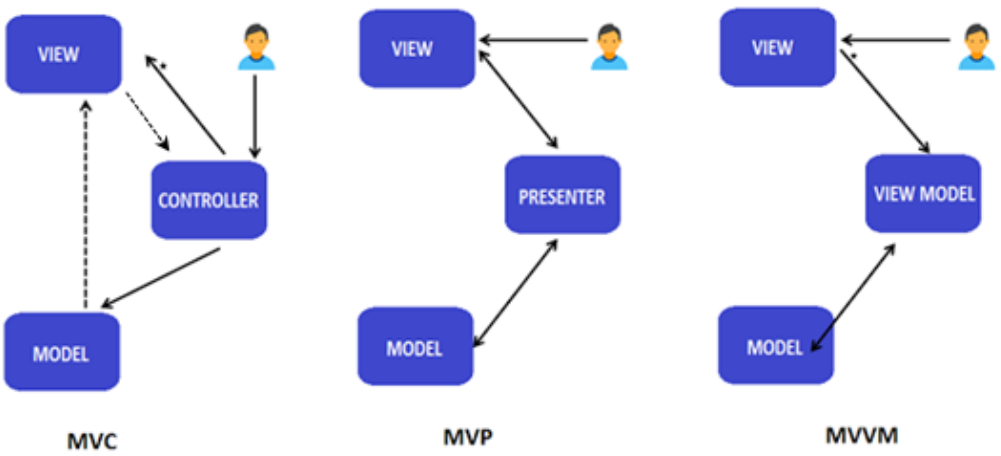
En todo el patrón se implementa el patrón “Observer” (en más de un sentido y entre los distintos componentes de cada capa) para desacoplar el 'modelo' de la 'vista' para aumentar la cambiabilidad y/o la seguridad de todo el conjunto.

MVC y variantes principales (MVP y MVVM) y sus mejoras

En la medida que se fue reinterpretando y mejorando el MVC, se presentaron las variantes. MVP (Model View Presenter) y MVVM (Model-View View-Model) son las más difundidas[1] siempre relacionadas con las interfaces de usuarios.

Las vamos a explorar en forma conjunta para entender que implica cada una y cómo usarlas. Las tres versiones ayudan a desarrollar aplicaciones y se suelen combinar de forma flexible.

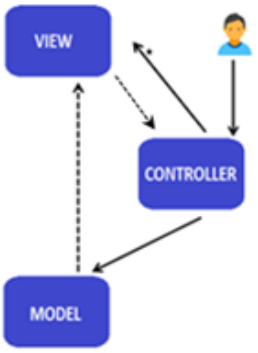
Sin entrar aún en detalles, el gráfico de abajo muestra un esquema de las interacciones entre los componentes de cada capa en el patrón original y estas variantes, indicando las distintas formas en las que fluye la información y se comunican.



[1] Existen otras variantes tales como **HMVC** (MVC Jerárquico) o **MVA** (Modelo-Vista-Adaptador), que escapan al alcance de ésta materia o que quizás se consideren de momento menos difundidas o popularizadas.

MVC Original

En su versión original, divide a los componentes en tres capas, y define las funciones de cada uno de la siguiente manera:



MVC

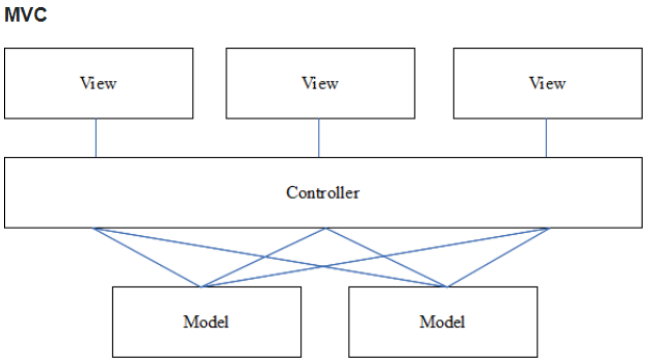
Vista: representa componentes de la IU (como XML, HTML, etc). Muestra datos que se reciben del controlador como resultado. La vista supervisa al modelo para detectar cualquier cambio de estado y accede al modelo actualizado. Modelo y Vista interactúan entre sí utilizando el patrón Observer (el modelo es sujeto de observación de la vista).

Controlador: es responsable de recibir, procesar y enviar al modelo las solicitudes entrantes de datos. El Controlador supervisa a la Vista. Cuando en la Vista ocurre algún evento el controlador la escucha y atiende sus eventos. Controler y la View también interactúan entre sí con un patrón observer (la Vista es sujeto de observación del controlador).

De ese modo, envía los datos del usuario al Modelo y devuelve los resultados a la Vista. Se dice que actúa como receptor de eventos, administrador del flujo de datos, pudiendo redirigir la respuesta de una Vista a otra.

Modelo: significa que sus componentes manejan los datos que se requieren para mostrar en la Vista. Representa una colección de clases que describe la lógica de negocio (modelo de negocio y el modelo de datos).

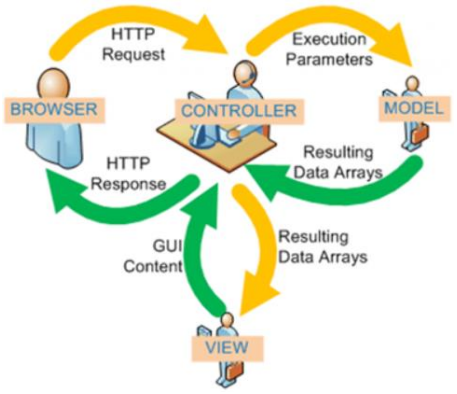
También define las reglas de negocios para los medios de datos como la forma en que se pueden cambiar y manipular los datos.



En esta versión tradicional del patrón, la información circula entre las tres capas de forma casi unidireccional completando un ciclo entre los componentes de las tres capas.

Esta circulación sería en alguna medida “asincrónica”, ya que un componente envía un mensaje a otro de otra capa y no espera la respuesta de ese mismo componente.

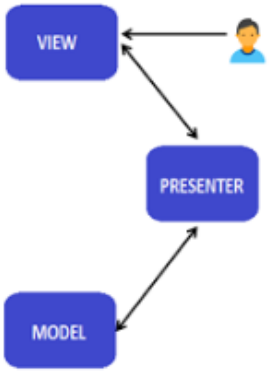
Otros aspectos de la visión y/o implementación original del patrón son los mostrados en los gráficos de la derecha, en la cual se ve al controlador como un componente único o muy compartido (con baja cohesión) entre todas las vistas, mostrando menor nivel de mantenibilidad y reutilización.



Modelo Vista Presentador (MVP)

El patrón MVP es similar al patrón MVC, se deriva del mismo y es más moderno.

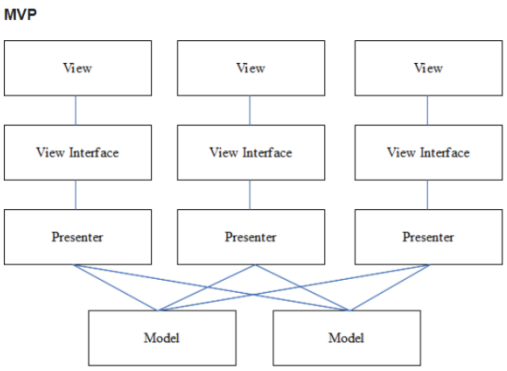
En este caso el controlador o los controladores es o son reemplazados o renombrados como un Presentador/es (debido al cambio en sus funciones). Las funciones en cada capa quedan de la siguiente manera:



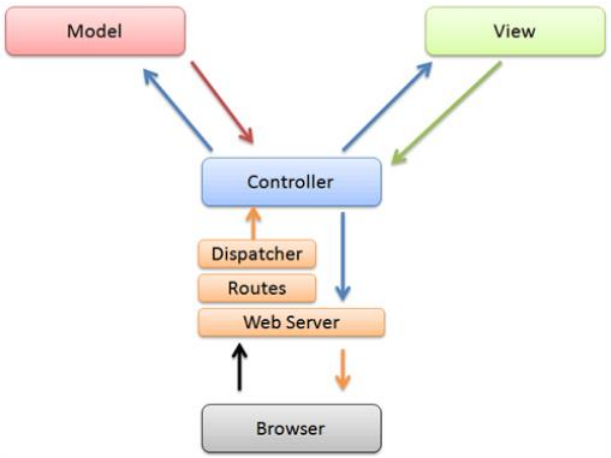
MVP

Vista: el o los componentes de esta capa interactúan directamente con el usuario mediante componentes del tipo HTML, XML, Actividad (“action” según algunos frameworks), etc. En esta versión, se busca que los componentes de la Vista, no contengan ninguna lógica implementada.

Presentador/es: recibe y procesa los eventos de entrada de los usuarios a través de la Vista, luego procesa (validando según el caso y la implementación) los datos del usuario, y con la ayuda del Modelo, devuelve los resultados a la Vista/s. El Presentador se comunica con la Vista a través de un interfaz. La interfaz se define en la clase de presentador, a la que pasa los datos requeridos. La actividad / fragmento o cualquier otro componente de la Vista implementa esta interfaz y representa los datos de la manera que ellos quieran.



Modelo: de igual modo que antes representa un conjunto de clases que describe la lógica de negocios. También pueden definir las reglas de negocios para los datos significa cómo se pueden cambiar y manipular los datos.



En este patrón, el Presentador manipula el modelo y también actualiza la Vista. En MVP Vista y Presentador están completamente desacoplados entre sí y se comunican entre sí mediante la interfaz. Este desacoplamiento del mockup de la vista es más fácil para las pruebas unitarias respecto del MVC original. Algunos autores consideran a este derivado como una versión (además de más moderna o simple), más segura ya que el modelo está más resguardado debido a la mediación del Presentador.

Una representación más real de esta variante se muestra en el siguiente gráfico. En el mismo se ve con claridad el conjunto de actividades que realiza el controlador.

En este caso también, la Vista y el Modelo están completamente desacoplados, ni siquiera se conocen, haciendo que toda la implementación sea de alguna manera más segura, y permitiendo que el Modelo pueda ser reutilizado para otros conjuntos de Presentadores y Vistas.

En este caso, los componentes de la Vista son sujeto de observación (o son observados) por parte del Presentador (los observadores). Por otro lado el Presentador esta también observador del Modelo, de este modo cuando se producen cambios en los dos extremos (Vista y Modelo), el Presentador se ocupa de actualizarlos.

Por lo anterior, se dice en este caso que el Presentador actúa como si fuera un “patrón mediador” (que es otro patrón de diseño del GoF), cuya motivación es vincular a dos objetos que ambos no se conocen y que no fueron desarrollados para comunicarse.

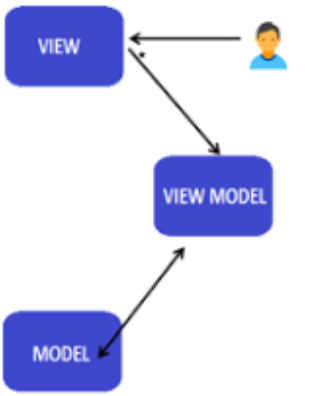
Esta variante hace que las comunicaciones o las secuencias de mensajes enviados entre los componentes, al ir y venir a través del Presentador, sean completamente sincrónicas. Esto permite que todo el arreglo sea más claro de entender, seguir, validar y probar.

Modelo Vista Vista-Modelo (MVVM)

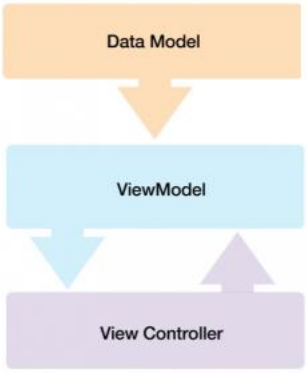
Esta variante admite el enlace de datos bidireccional entre Vista y Vista-Modelo (otra forma de llamar a la capa Controlador). Permite la propagación automática de cambios, dentro del estado de Vista-Modelo a la Vista.

En general, la Vista-Modelo también utiliza un patrón de observador para informar los cambios en la Vista Modelo al Modelo. Este derivado es más usado en lenguajes de plataformas móviles.

Vista-Modelo: responsable de exponer métodos, comandos y otras propiedades que ayudan a mantener el estado de la Vista, manipular el Modelo como resultado de las acciones en la Vista y desencadenar eventos en la Vista misma.



MVVM



La Vista tiene una referencia a Vista-Modelo pero Vista-Modelo no tiene información sobre Vista. Existe una relación de muchos a uno entre Vistas y Vista-Modelo, lo que significa que muchas Vistas se pueden mapear a un View-Model.

El enlace de datos bidireccional o el enlace de datos bidireccional entre la Vista y el Vista-Modelo garantizan que los Modelos y las propiedades del Modelo de Vista estén también sincronizados con la Vista.

El patrón de diseño de MVVM es muy adecuado en aplicaciones que necesitan soporte para enlace de datos bidireccional.

Conclusión de MVC y derivados

- Podríamos haber profundizado en esta discusión, hablando de otras variaciones de MVC (como las mencionadas por Martin Fowler), o incorporar otros patrones como el "Front Controler". Pero, lo comentado nos da una buena idea de las principales diferencias entre los tres patrones.
- No todos los autores, proveedores y comunidad consensúa las formas del patrón y/o sus derivados. Es natural ver discrepancias de interpretación y/o implementación en cualquiera de sus formas.

Uso

- Ampliamente usado en aplicaciones World Wide Web en los principales lenguajes de programación a través de los frameworks:

(.NET) - MonoRail
(.NET) - Spring.NET
(.NET) - Maverick.NET
(.NET) - ASP.NET MVC
(.NET) - User Interface Process (UIP) App Block
(C++) - treefrog
(Objective C) - Cocoa
(Ruby) - Ruby on Rails
(Ruby) - Merb
(Ruby) - Ramaze
(Ruby) - Rhodes
(Java) - Grails
(Java) - Interface Java Obj.
(Java) - Framework Dinámica
(Java) - Struts
(Java) - Brutos framework
(Java) - Beehive
(Java) - Spring
(Java) - Tapestry
(Java) - Aurora
(Java) - JavaServerFaces
(Java) - PrimeFaces
(Java) - Vaadin
(JavaScript) - Sails.JS
(JavaScript) - ExtJS 4
(JavaScript) - AngularJS
(JavaScript) - Nest
(Perl) - Mojolicious
(Perl) - Catalyst
(Perl) - CGI::Application
(Perl) - Gantry Framework
(Perl) - Jifty
(Perl) - Maypole
(Perl) - OpenInteract2
(Perl) - PageKit
(Perl) - Cyclone 3
(Perl) - CGI::Builder
(PHP) - BitPHP
(PHP) - Yii
(PHP) - Laravel
(PHP) - Self FrmWrk (php5, MVC/ORM, I18N, + DB)
(PHP) - ZanPHP
(PHP) - Tlalokes
(PHP) - SiaMVC
(PHP) - Agavi
(PHP) - Zend Framework
(PHP) - CakePHP
(PHP) - KumbiaPHP
(PHP) - Symfony
(PHP) - QCodo
(PHP) - CodeIgniter
(PHP) - Polka-PHP
(PHP) - Kohana
(PHP) - PHP4ECore
(PHP) - Practico
(PHP) - FlavorPHP
(PHP) - Yupp PHP
(PHP) - Yii PHP Framework
(PHP) - Osezno PHP FrmWrk
(PHP) - (sPHPf) Simple PHP FrmWrk
(PHP) - gvHidra
(Python) - Zope3
(Python) - Turbogears
(Python) - Web2py
(Python) - Pylons
(Python) - Django
(AS3) - Cairngorm
(AS3 y Flex) – Cycle Frame Work

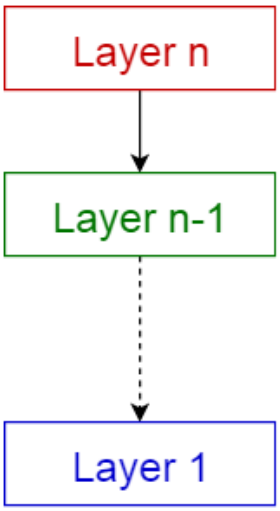
Anexo – Los diez Patrones de Arquitectura más usados

1. Patrón de capas (Layered)

Este patrón se puede utilizar para estructurar programas que se pueden descomponer en grupos de sub tareas, cada una de las cuales se encuentra en un nivel particular de abstracción. Cada capa proporciona servicios a la siguiente capa superior.

Las 4 capas más comúnmente encontradas de un sistema de información general son las siguientes.

- Capa de presentación (también conocida como capa UI)
- Capa de aplicación (también conocida como capa de servicio)
- Capa de lógica de negocios (también conocida como capa de dominio)
- Capa de acceso a datos (también conocida como capa de persistencia)

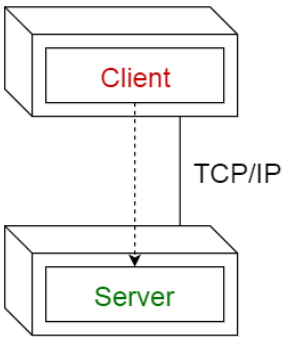


Patrón de capas

Uso

- Aplicaciones de escritorio generales.
- Aplicaciones web de comercio electrónico.

2. Patrón Cliente Servidor (Client – Server)



Patrón cliente-servidor

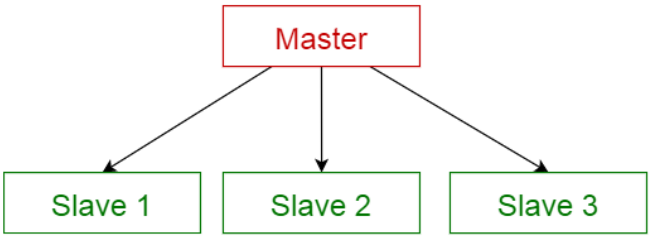
Este patrón consiste en dos partes; un **servidor** y múltiples **clientes**. El componente del servidor proporcionará servicios a múltiples componentes del cliente.

Los clientes solicitan servicios del servidor y el servidor proporciona servicios relevantes a esos clientes. Además, el servidor sigue escuchando las solicitudes de los clientes.

Uso

- Aplicaciones en línea como correo electrónico, uso compartido de documentos y banca.

3. Patrón Maestro-Esclavo (Master-Slave)



Patrón maestro-esclavo

Este patrón consiste en dos partes; **maestro** y **esclavos**.

El componente maestro distribuye el trabajo entre componentes esclavos idénticos y calcula el resultado final de los resultados que devuelven los esclavos.

Uso

- En la replicación de la base de datos, la base de datos maestra se considera como la fuente autorizada y las bases de datos esclavas se sincronizan con ella.
- Periféricos conectados a un bus en un sistema informático (unidades maestra y esclava).

4. Patrón de Filtro y Tubería (Pipe & Filter)

Este patrón se puede usar para estructurar sistemas que producen y procesan una secuencia de datos. Cada paso de procesamiento se incluye dentro de un componente de **filtro**.

Los datos que se procesarán se pasan a través de las **tuberías**. Estas tuberías se pueden utilizar para el almacenamiento en búfer o con fines de sincronización.

Uso

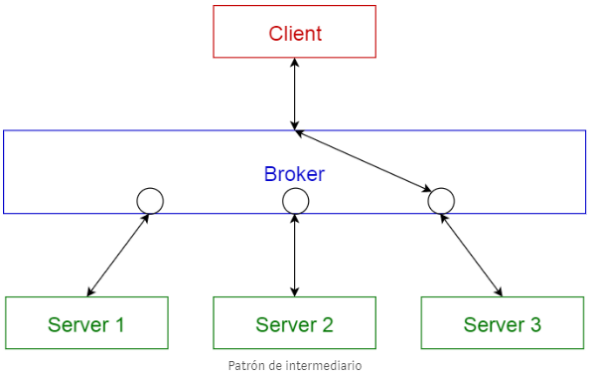
- Compiladores Los filtros consecutivos realizan análisis léxico, análisis sintáctico y generación de código.
- Flujos de trabajo en bioinformática.

Patrón de filtro de tubería

5. Patrón del Agente (Broker)

Este patrón se usa para estructurar sistemas distribuidos con componentes desacoplados.

Estos componentes pueden interactuar entre sí mediante invocaciones de servicios remotos.



Patrón de intermediario

Un componente de **intermediario** es responsable de la coordinación de la comunicación entre los **componentes**.

Los servidores publican sus capacidades (servicios y características) a un intermediario.

Los clientes solicitan un servicio del intermediario y el intermediario redirección al cliente a un servicio adecuado desde su registro.

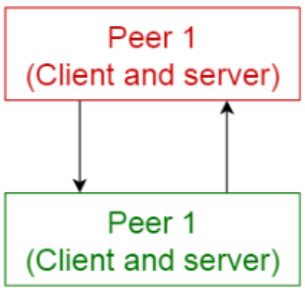
Uso

- Software de Message Broker cómo **Apache ActiveMQ**, **Apache Kafka**, **RabbitMQ** y **JBoss Messaging** .

6. Patrón de igual a igual (Peer to Peer)

En este patrón, los componentes individuales se conocen como **pares**. Los pares pueden funcionar tanto como un **cliente**, solicitando servicios de otros pares, y como un **servidor**, proporcionando servicios a otros pares.

Un par puede actuar como un cliente o como un servidor o como ambos, y puede cambiar su rol dinámicamente con el tiempo.



Patrón de igual a igual

Uso

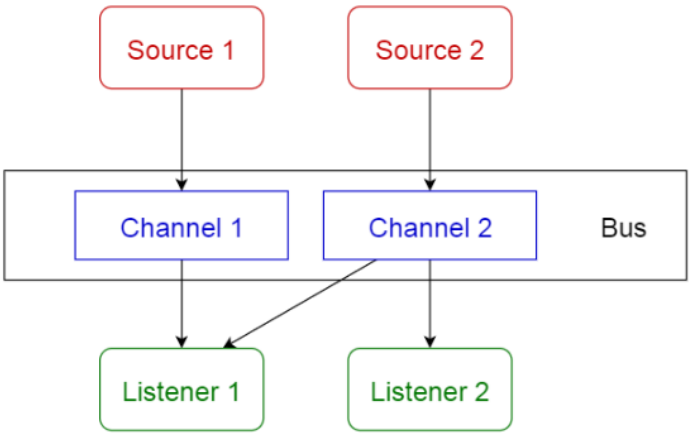
- Redes de intercambio de archivos como **Gnutella** y **G2**)
- Protocolos multimedia como **P2PTV** y **PDTP** .

7. Patrón de bus de evento (Event Bus)

Este patrón trata principalmente con eventos y tiene 4 componentes principales; **fuente de evento**, **escucha de evento**, **canal** y **bus de evento**.

Las fuentes publican mensajes en canales particulares en un bus de eventos. Los oyentes se suscriben a canales particulares.

Los oyentes son notificados de los mensajes que se publican en un canal al que se han suscrito anteriormente.



Patrón de bus de evento

Uso

- Todo tipo de aplicaciones Corporativas.
- Desarrollos en Android.
- Servicios de notificación en general multiplataforma.

8. Patrón de modelo-vista-controlador (MVC)

Como hemos visto anteriormente, divide una aplicación interactiva en 3 capas, con las funciones que hemos comentado. Resumiendo:

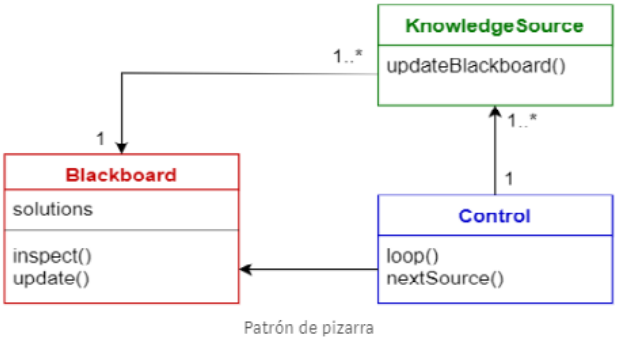
1. **Vista**: muestra la información al usuario (puede y suele haber más de una).
2. **Controlador**: maneja los eventos de la vista y el control de flujo.
3. **Modelo:** contiene la funcionalidad y datos básicos.

9. Patrón de pizarra (Dashboard)

Este patrón es útil para problemas para los que no se conocen estrategias de solución deterministas.

El patrón de pizarra consta de 3 componentes principales.

- **pizarra** : una memoria global estructurada que contiene objetos del espacio de solución
- **fuente de conocimiento** : módulos especializados con su propia representación
- **componente de control**: selecciona, configura y ejecuta módulos.



Patrón de pizarra

Todos los componentes tienen acceso a la pizarra. Los componentes pueden producir nuevos objetos de datos que se agregan a la pizarra.

Los componentes buscan tipos particulares de datos en la pizarra, y pueden encontrarlos por coincidencia de patrones con la fuente de conocimiento existente.

Uso

- Reconocimiento de voz
- Identificación y seguimiento del vehículo
- Identificación de la estructura proteica
- Sonar señala la interpretación.

10. Patrón Intérprete (Interpreter)

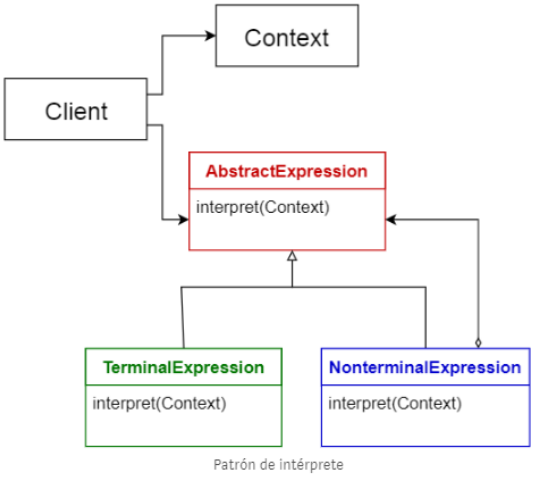
Este patrón se usa para diseñar un componente que interpreta programas escritos en un lenguaje dedicado.

Especifica principalmente cómo evaluar las líneas de programas, conocidas como oraciones o expresiones escritas en un idioma particular.

La idea básica es tener una clase para cada símbolo del idioma.

Uso

- Lenguajes de consulta de base de datos como SQL.
- Idiomas utilizados para describir los protocolos de comunicación.



Patrón de intérprete

Conclusión de POSA

Ventajas y Desventajas de los Patrones de Arquitectura

Patrón	Ventaja	Desventaja
Patrón de capas	Una capa de bajo nivel poder ser usada por una diferente de más alto nivel. Las capas hace la estandarización más fácil, en la medida en la que definimos niveles funcionales. Los cambios se pueden hacer en cada capa, sin necesariamente afectar a las restantes.	No es aplicable universalmente (o en todos los casos). Algunas capas pueden llegar a ser salteadas o sobrepasadas en ciertas situaciones.
Patrón cliente-servidor	Bueno para modelar un set de servicios donde los clientes puedan luego requerirlos.	Los requerimientos son típicamente manejados en hilos de ejecución separados en servidores. Los procesos de intercomunicación de estos hilos van en detrimento del rendimiento dado lo requerido lo requerido por los distintos clientes.
Patrón maestro-esclavo	Provee precisión. La ejecución de un servicio puede ser delegada a diferentes esclavos con diferentes implementaciones.	El componente esclavo está asilado: no hay estado compartido. La latencia en una comunicación maestro-esclavo, puede ser un problema en ciertos tipos de aplicaciones (por ejemplo aplicaciones en tiempo real). Este patrón solo puede ser aplicado en un problema que puede ser descompuesto (en varios esclavos).
Patrón de filtro de tubería	Exhibe el procesamiento concurrente, cuando la entrada y la salida suceden en ráfagas, y los filtros comienzan a procesar. Es fácil de agregar filtros. El sistema puede ser fácilmente extensible. Los filtros son reusables. Se pueden construir distintas tuberías (de procesos y filtros), recombinando un determinado conjunto de filtros.	La eficiencia es limitada por el tiempo de procesamiento del filtro más lento. La transformación de los datos va en detrimento del rendimiento cuando los datos se mueven de un filtro a otro.
Patrón de intermediario	Permite cambios dinámicos relacionados con la agregación, borrado, reubicación de objetos y los hace transparente al desarrollador.	Implica una estandarización de las descripciones de los servicios.
Patrón de igual a igual	Soporta el procesamiento descentralizado. Es altamente robusto ante la posibilidad de fallo de algún nodo. Es altamente escalable en términos de recursos de procesamiento.	No hay garantía acerca de la calidad de servicio, debido a que los nodos cooperan voluntariamente. Se dificulta garantizar la seguridad. El rendimiento depende de la cantidad de nodos.
Patrón de bus de evento	Las nuevas publicaciones, suscripciones y conexiones se pueden agregar fácilmente. Es efectivo en aplicaciones altamente distribuidas.	La escalabilidad puede ser un problema, dado que todos los mensajes viajan a través del mismo bus de eventos.
Modelo-vista-controlador	Hace sencillo tener múltiples vistas de un mismo modelo, lo cual también permite que cada una se pueda conectar y desconectar independientemente en tiempo de ejecución.	Incrementa la complejidad. Puede generar actualizaciones innecesarias para las acciones disparadas por los usuarios.
Patrón de pizarra	Es fácil de agregar nuevas aplicaciones que necesiten compartir los datos de la pizarra. Extender la estructura de los datos de la pizarra es también fácil.	Modificar la estructura de datos es complejo, debido a que muchas aplicaciones son afectadas. Suele necesitar sincronización en el acceso de datos.
Patrón de intérprete	Permite un comportamiento altamente dinámico. Resulta beneficioso para la programación de usuario final. Mejora la flexibilidad debido a que reemplazar un programa interpretado es fácil.	Debido a que un lenguaje interpretado es generalmente mas lento que un compilado, el rendimiento puede ser un problema.