INSTITUCIÓN UNIVERSITARIA
SNIES 4709

Patrones de Arquitectura de Software

Alejandro Matallana
Sebastian Salazar
Miguel Tamayo

Institución universitaria EAM
Arquitectura de software
Ingeniería de Software
2018

Patrón de arquitectura modelo vista controlador (MVC)

Modelo-vista-controlador (**MVC**) es un patrón de arquitectura de software, que separa los datos y la lógica de negocio de una aplicación de su representación y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el **modelo**, la **vista** y el **controlador**, es decir, por un lado, define componentes para la representación de la información, y por otro lado para la interacción del usuario. Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

El patrón MVC divide la aplicación en tres partes diferenciadas: el Modelo, la Vista y el Controlador.

- El **Modelo**: gestiona los **datos** de la aplicación. En nuestra aplicación, el modelo se encargará de guardar la **información** de todos los marcadores que el usuario haya añadido. El modelo no conocerá nada de la vista o el controlador. Su tarea es guardar y gestionar la información.
- La **Vista**: representa el estado actual del Modelo, sin estar en contacto con él. La vista es la parte "tonta" de la aplicación, su tarea es mostrar la información al usuario.
- El **Controlador**: Es el enlace entre el modelo y la vista. Se encarga de "avisar" al modelo cuando el usuario manipule la vista. En nuestra aplicación, el controlador será responsable de gestionar los cambios que el usuario lleve a cabo, como añadir o eliminar un marcador.

Cuando utilizarlo:

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.

Diagrama:

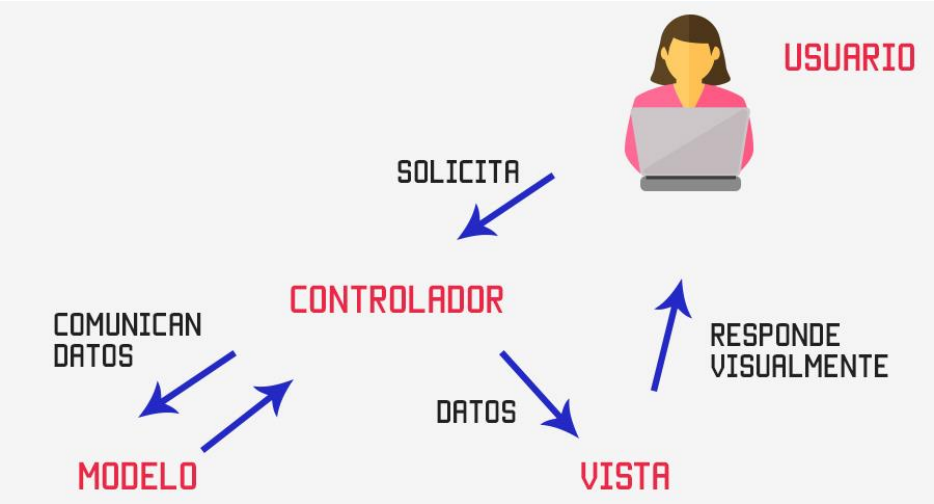
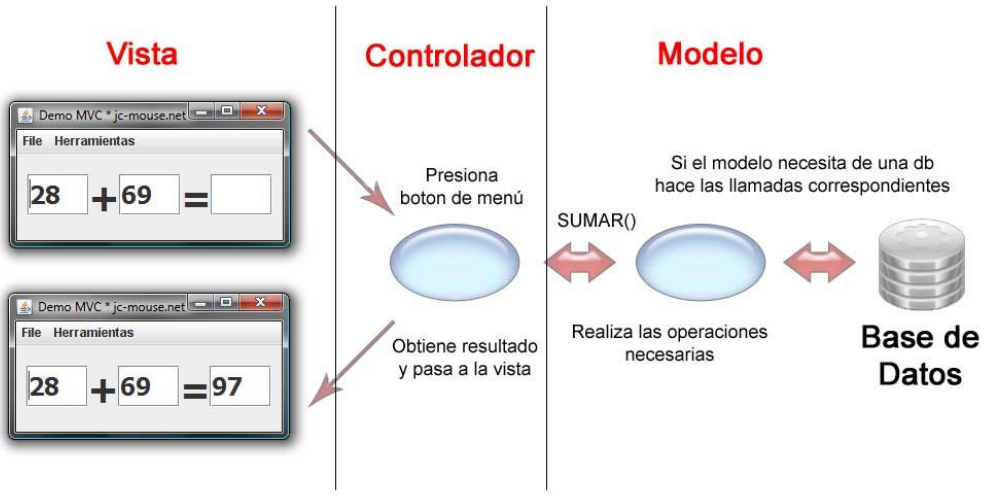


Diagrama uso real:



Análisis del patrón:

Overall agility: Alta, porque un componente no afecta al otro entonces al momento de agregar nuevas opciones no se afecta al modelo como tal.

Ease of deployment: Alta, porque cuenta con un bajo acoplamiento.

Testability: Alta, porque el patrón maneja componentes que no dependen uno del otro, entonces facilita su uso a la hora de realizar pruebas sobre ellos.

Performance: Baja, el patrón requiere de una gran cantidad de tiempo al momento de su implementación, fundamentalmente cuando se inicia, ya que al tener tres componentes individuales se requiere una configuración especial para su comunicación, esto es tiempo que se puede dedicar en otras actividades y que otros patrones no tienen.

Scalability: Alta, debido a que los componentes no cuentan con una relación entre ellos, facilita su manejo y a su vez permite una buena escalabilidad.

Ease of development: Alta, porque cuenta con un bajo acoplamiento entonces facilita en gran medida realizar un desarrollo especifico en cada uno de los componentes.

Arquitectura en pizarra:

El modelo de arquitectura en pizarra separa el sistema en una estructura de datos centra y globalmente accesible, normalmente llamada pizarra, y un conjunto de módulos o programas independientes, denominados fuentes de conocimiento (Procesos). Las fuentes de conocimiento realizan sus tareas en base al contenido de la pizarra y tienen como objetivo realizar cambios en la misma. La idea de este patrón de arquitectura es que las fuentes de conocimiento no se puedan comunicar entre si directamente o interactuar de una forma que no sea mediante cambios en la pizarra, es decir, la resolución del problema progresa a medida que el estado de la pizarra evoluciona.

La arquitectura en pizarra consta de múltiples elementos funcionales denominados agentes y un instrumento de control denominado pizarra.

Todo modelo de este tipo consiste en las siguientes tres partes:

- Fuentes de conocimiento, necesarias para resolver el problema.
- Una pizarra que representa el estado actual de la resolución del problema.
- Una estrategia, que regula el orden en que operan las fuentes.

Cuando utilizarlo:

Estos sistemas se han usado en aplicaciones que requieren complejas interpretaciones de proceso de señales (reconocimiento de patrones,

reconocimiento de habla, etc.), o en sistemas que involucran acceso compartido a datos con agentes débilmente acoplados.

También se han implementado estilos de este tipo en procesos en lotes de bases de datos y ambientes de programación organizados como colecciones de herramientas en torno a un repositorio común.

- Reconocimiento de voz
- Identificación y seguimiento de vehículos.
- Identificación de la estructura de la proteína
- Sonar señales de interpretación.

Ventajas:

- Hace posible la interacción de agentes contra el sistema
- Funciona muy bien con los problemas no deterministas (IA)
- Tenemos el conocimiento de la secuencia del proceso

Desventajas:

- Problemas de seguridad ya que la pizarra es accesible por todos.
- Problemas de sincronización al chequear y vigilar la pizarra.

Diagrama:

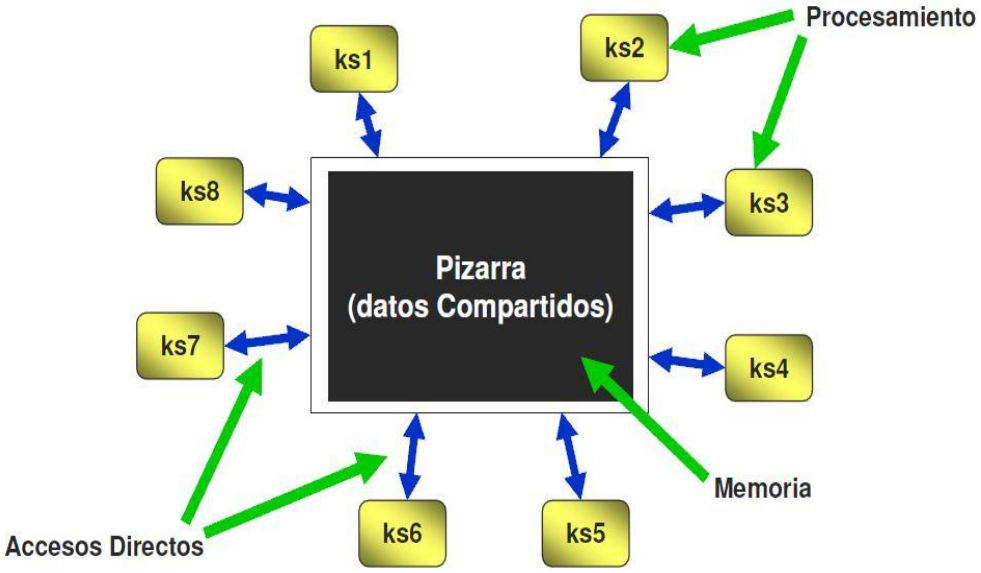
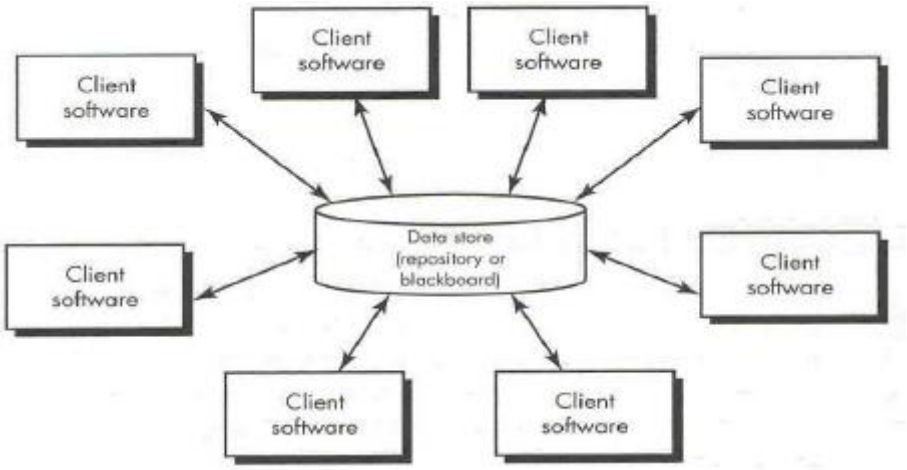


Diagrama uso real:



Análisis del patrón:

Overall Agility: Media porque si se eliminan fuentes de conocimiento la afectación que tiene sobre el patrón es baja, caso contrario a lo que pasa se cambia la estrategia que es la encargada de regular el orden en que se maneja las fuentes de estímulo, además si también se cambiara la pizarra que es un elemento fundamental en el correcto funcionamiento del patrón esta conllevaría a tener varios problemas de comunicación entre sí.

Ease of deployment: Baja, porque antes de poner a funcionar las fuentes de conocimiento se debe primero tener un orden para que se pueda ir agregando una tras otra secuencialmente.

Testability: Alta, porque entre sus componentes existe un bajo acoplamiento

Performance: Media, el patrón es términos generales es sencillo, gracias al bajo acoplamiento que este posee, pero tiene un gran problema y es que en algunos casos no se llega a la solución del problema planteado con su implementación en el proyecto, lo cual hace que tenga una gran desventaja ya que siempre se busca que por medio de los patrones lleguen a la solución del problema, además con este patrón no se puede ver la traza de procesos que este tiene.

Scalability: Alta, porque los componentes no tienen relación alguna entre ellos lo cual facilita su manejo en gran manera.

Ease of development: Alta, porque el patrón cuenta con componentes que tienen bajo acoplamiento entre si lo cual quiere decir que se puede hacer un desarrollo sencillo en términos generales.

Patrón de arquitectura en Tiempo Real

Un sistema de tiempo real es un sistema de procesamiento de información el cual tiene que responder a estímulos de entrada generados externamente en un período finito y específico.

Un sistema en tiempo real es una combinación de computadoras, dispositivos de E/S, hardware y software de propósito específico en donde:

- Existe una fuerte interacción con el ambiente.
- El ambiente cambia con el tiempo
- El sistema debe controlar y/o reaccionar a diferentes aspectos del ambiente.

Un sistema de tiempo real es un software cuyo correcto funcionamiento depende de los resultados producidos por el mismo y del instante de tiempo en el que se producen estos resultados. Un sistema de tiempo real blando (soft) es un sistema cuyo funcionamiento degrada si los resultados no se producen correctamente de acuerdo con los requerimientos temporales especificados. Un sistema de tiempo real duro (hard) es un sistema cuyo funcionamiento es incorrecto si los resultados no se producen de acuerdo a la especificación temporal.

Una forma de ver un sistema en tiempo real es como un sistema de estimo/respuesta. Dado un estímulo de entrada, el sistema debe producir la correspondiente salida. Se puede, por lo tanto, definir el comportamiento de un sistema de tiempo real haciendo una lista de estímulos recibidos por el sistema, las respuestas asociadas y el tiempo real en que dichas respuestas deben producirse.

Ventajas

- Simplicidad
- Evolución: se pueden reemplazar componentes suscriptores
- Modularidad: una sola modalidad para eventos diversos

Desventajas

- Posibilidad de desborde
- Potencial imprevisión de escalabilidad
- Pobre comprensibilidad: Puede ser difícil prever qué pasará en respuesta a una acción
- No hay garantía del lado del publicador, que el suscriptor responderá al evento
- No hay mucho soporte de recuperación en caso de falla parcial

Cuando utilizarlo:

Estos sistemas se han usado en muchas aplicaciones, como por ejemplo un sistema debe examinar un sensor cada 30 milisegundos y realizar una acción dependiendo del valor de ese sensor.

Además, cuando tengamos o se nos presente algunas de las siguientes necesidades:

- Representación de interrupciones y cambio de contexto
- Concurrencia
- Comunicación y sincronización de tareas
- Amplias variaciones en las tasas de datos y la comunicación

Diagrama:

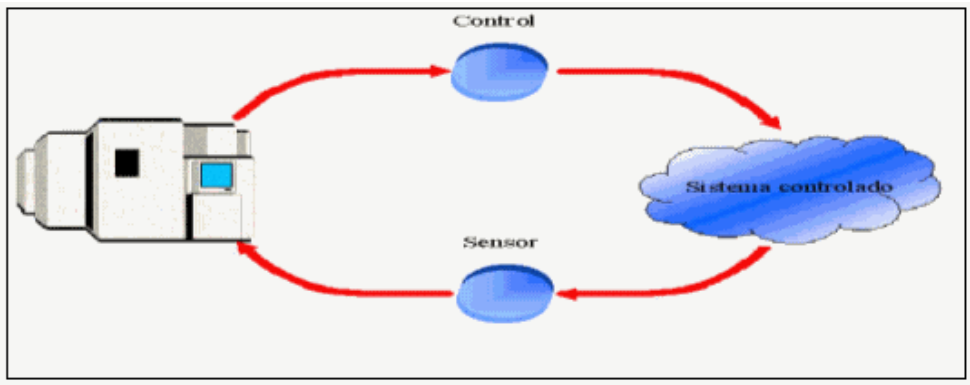
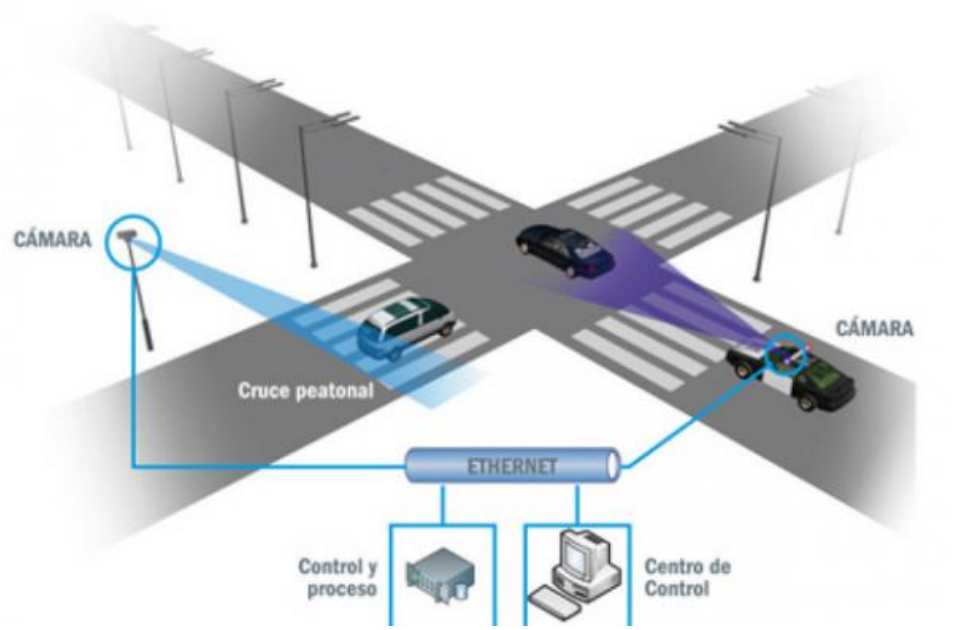


Diagrama uso real:



Análisis del patrón:

Overall agility: Alta, porque la arquitectura tiene que ser ágil ya que las aplicaciones construidas con esta, necesitan precisión y tiempos de entrada y salida muy cortos.

Ease of deployment: Media, porque esta compuesta por componentes que están unidos entre si y al ejecutarse lo hacen todos al mismo tiempo.

Testability: Media, porque para probar se necesita algo del ambiente en el cual se va a emplear, se debe saber cuál será su funcionamiento para poder hacer una acción que ejecute el procedimiento esperado y responda en el tiempo aceptado.

Performance: Alto, porque este tipo de arquitecturas manejan tiempos de respuesta que son inferiores a los segundos, lo cual hace que su rendimiento sea demasiado preciso y óptimo.

Scalability: baja, porque sus componentes están muy acoplados y hace difícil la creación de nuevos componentes para que crezca o mejore la aplicación.

Ease of development: Baja, porque para este tipo de aplicaciones es indispensable el uso de componentes externos.

Patrón CQRS (Command and Query Responsibility Segregation)

El concepto central de este patrón es que una aplicación que tenga operaciones de lectura y operaciones de escritura deben estar totalmente separadas. En las aplicaciones tradicionales se usa el mismo modelo de datos para consultar y actualizar, esto funciona bien para las operaciones CRUD básicas, pero en aplicaciones más complejas puede resultar difícil de manejar. CQRS da solución a estos problemas mediante la separación de lecturas y escrituras en modelos independientes con comandos para actualizar datos y consultas para leer datos.

Si se utilizan bases de datos de escritura y de lectura independientes, se deben mantener sincronizados, esto se consigue haciendo que el modelo de escritura publique un evento cada vez que actualiza la base de datos. Algunas implementaciones de CQRS utilizan el patrón Event Sourcing con este patrón, el estado de la aplicación se almacena como una secuencia de eventos. Cada evento representa un conjunto de cambios en los datos.

Ventajas

- CQRS permite las cargas de trabajo de lectura y escritura que se escalen de forma independiente, lo que puede dar lugar a menos contenciones de bloqueo.
- El lado de lectura puede usar un esquema que está optimizado para las consultas, mientras que el lado de escritura utiliza un esquema que está optimizado para las actualizaciones.
- Es más fácil asegurarse de que solo las entidades de dominio correctas realicen escrituras en los datos.
- La separación de los lados de lectura y escritura puede dar lugar a modelos que sean más flexibles y fáciles de mantener. La mayor parte de la lógica

de negocios compleja entra en el modelo de escritura y el modelo de lectura puede ser relativamente sencillo.

Desventajas

- La idea básica de CQRS es sencilla, pero puede generar un diseño de aplicación más complejo, especialmente si incluye el patrón Event Sourcing.
- Aunque CQRS no requiere mensajería, es normal usarla para procesar comandos y publicar eventos de actualización. En ese caso, la aplicación debe gestionar errores de mensaje o mensajes duplicados.
- Si separa las bases de datos de lectura y escritura, los datos de lectura pueden estar obsoletos.

Cuando utilizarlo

Para los dominios de colaboración en los que varios usuarios tienen acceso a los mismos datos, especialmente cuando las cargas de trabajo de lectura y escritura son asimétricas.

Aplique CQRS solo a aquellos subsistemas en los que se obtiene un valor evidente de la separación de lecturas y escrituras, en caso contrario está creando una complejidad adicional, pero sin obtener ventaja alguna.

Diagrama:

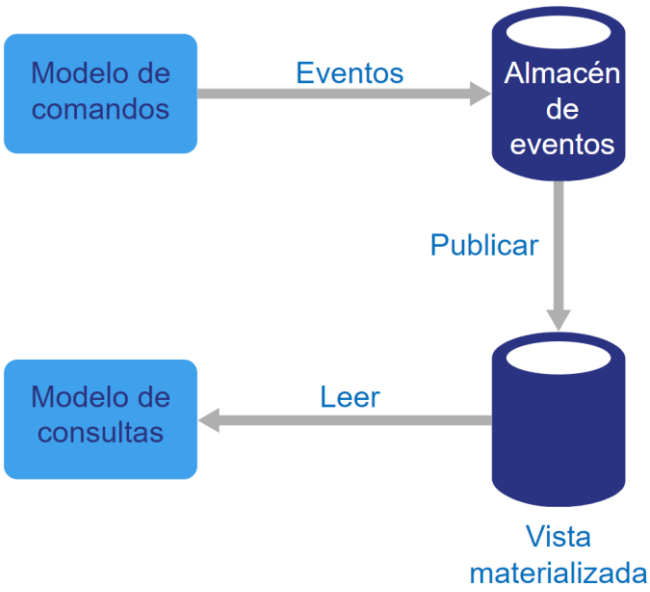
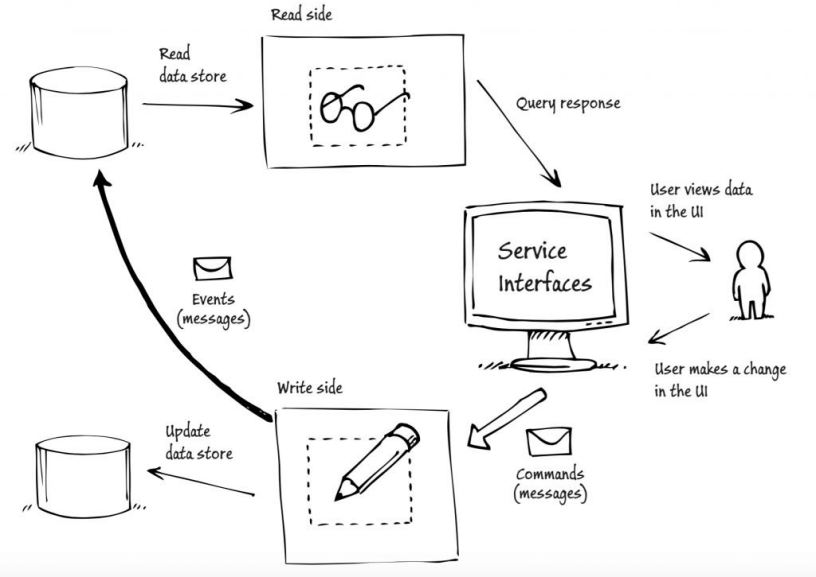


Diagrama uso real:



Análisis del patrón

Overall agility: Alta, porque su estructura hace que sea fácil de adaptarse a algún cambio de manera rápida.

Ease of deployment: Baja/Media, porque dependiendo de como hagamos el desarrollo podríamos tener las bases de datos desacopladas, una para

operaciones de lectura y otra para operaciones de escritura y poder tener facilidad a la hora de hacer un despliegue, pero también podemos tener una sola base de datos y separar estas operaciones de escritura y lectura, pero sería la misma base de datos acoplaríamos más el modelo.

Testability: Alta, al tener las operaciones de escritura y lectura separadas podemos hacer de manera mas adecuada y mas sencilla las pruebas de estos.

Performance: Medio/Alto, aunque las operaciones estén desacopladas o sean dos bases de datos una para la operación de escritura y otra para la operación de lectura para la comunicación entre estas se puede llegar a necesitar un servicio de mensajería y este puede llegar a bajar un poco el rendimiento por el manejo que se le debe hacer.

Scalability: Baja/Media, porque al tener las operaciones de lectura y escritura separadas permite que podamos identificar de una mejor manera que se debería escalar, pero la facilidad dependería mucho de si tenemos separadas las bases de datos por operación o si es una misma base de datos en la cual separamos estas operaciones.

Ease of development: Media/Alta, al separar las operaciones de escritura y lectura tendríamos una mejor visión a la hora de hacer el desarrollo, pero si usamos el patrón Event Sourcing haríamos el desarrollo mas complejo igualmente si usamos servicio de mensajeria.

Patrón Event Sourcing

En lugar de almacenar simplemente el estado actual de los datos de un dominio, se usa un almacén de eventos para registrar toda la serie de acciones realizadas en los datos. El almacén actúa como el sistema de registro esto puede simplificar las tareas en los dominios complejos, al evitar la necesidad de sincronizar el modelo de datos y el dominio empresarial, al tiempo que mejora el rendimiento, la escalabilidad y la capacidad de respuesta. También proporciona coherencia a los datos transaccionales y conserva trazas de auditoría e historiales completos.

Ventajas

- Cada evento representa una manipulación de los datos en un cierto punto de tiempo.
- Este patrón de arquitectura de software puede proporcionar un registro de auditoria.

Desventajas

- Esto requiere algo de disciplina porque no puede solo arreglar datos malos con un simple editor en base de datos.
- No es una tarea trivial cambiar la estructura de un evento. Por ejemplo, si agrega una propiedad, la base de datos aún contiene eventos sin esos datos su código tendrá que manejar estos datos faltantes amablemente.

Cuando utilizarlo:

Lo podemos usar en los siguientes posibles escenarios:

- Si desea capturar la intención, el propósito o el motivo en los datos. Por ejemplo, los cambios en una entidad de cliente se pueden capturar como una serie de tipos de evento específicos como Cambio de domicilio, Cierre de cuenta o Fallecimiento.
- Cuando el uso de eventos es una característica natural de la operación de la aplicación y requiere poco esfuerzo de implementación o desarrollo adicional.

Diagrama:

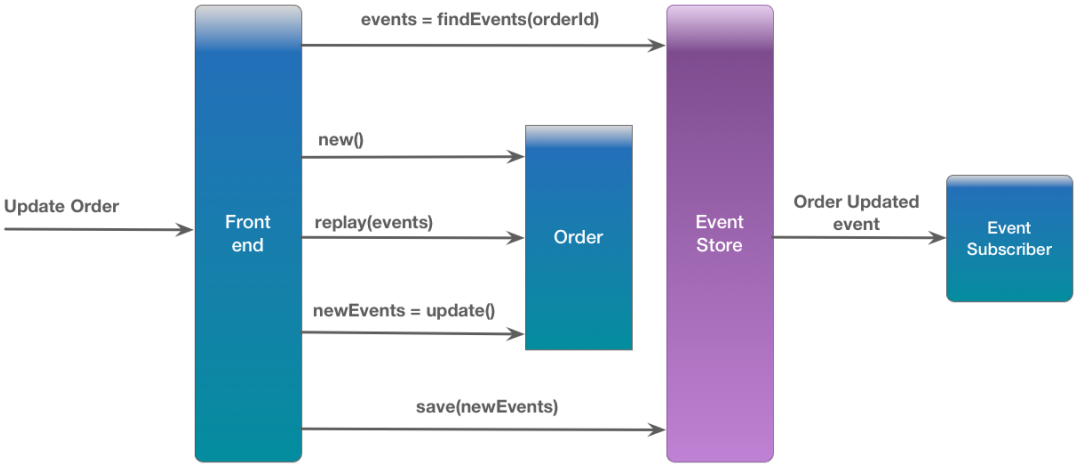
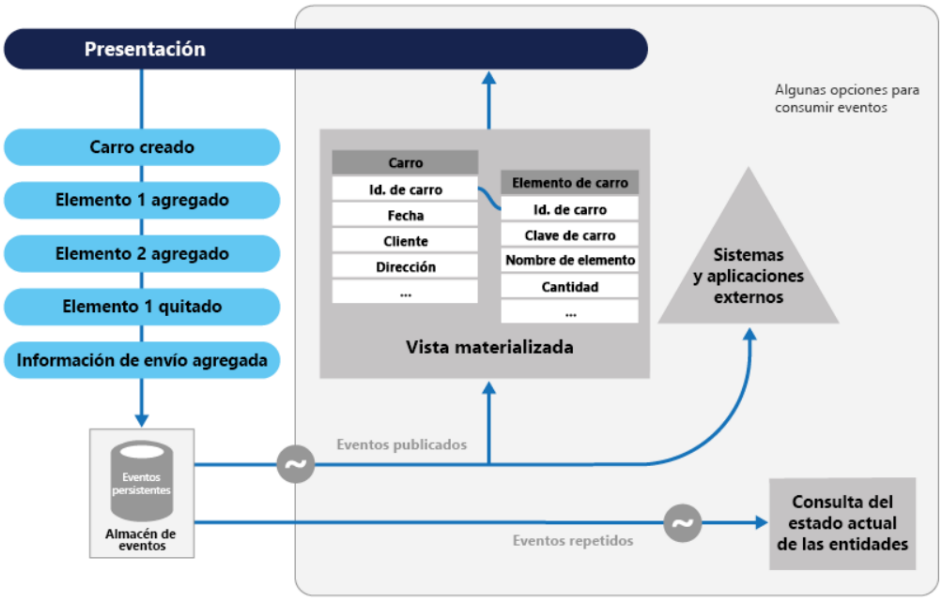


Diagrama uso real:



Análisis del Patrón

Overall agility: Media, porque al cambiar un dato en los eventos se vuelve algo mas complejo pero aún así el patrón esta muy estructurado permitiendo una visión detallada de su funcionamiento y así saber en donde se deben realizar cambios si son necesarios.

Ease of deployment: Baja, porque no es que se tenga mucho desacople en general y por ese motivo se hace difícil hacer cambios sin afectar la disponibilidad.

Testability: Alta, porque gracias al almacén de eventos cualquier cambio se ve reflejado como un nuevo registro y sería mas fácil a la hora de hacer pruebas.

Performance: Alto, al almacenar eventos se mejora mucho el rendimiento ya que se evitan muchos procesos largos y complejos, mejorando mucho el rendimiento.

Scalability: Media/Alta, porque con el almacén de eventos se puede simplificar muchas tareas complejas haciendo que se mas sencillo de escalar si se requiere.

Ease of development: Baja/Media, porque se debe hacer un tipo desarrollo donde se pueda manejar los cambios de datos en los eventos de una manera amable y óptima.

Patron SOA (Service Oriented Architecture):

La Arquitectura Orientada a Servicios permite la creación de sistemas altamente escalables que reflejan el negocio de la organización, a su vez brinda una forma bien definida de exposición e invocación de servicios, lo cual facilita la interacción entre diferentes sistemas propios o de terceros.

SOA proporciona una metodología y un marco de trabajo para documentar las capacidades de negocio y puede dar soporte a las actividades de integración y consolidación.

Cuando utilizarlo:

Cuando se necesite hacer la integración de la información dentro de un conjunto de aplicaciones dinámicas compuestas, por lo que los directivos de las organizaciones podrán tener acceso a información no solo más exacta y actualizada, sino también en menor tiempo, lo que les otorgará la posibilidad de reaccionar manera rápida y ágil ante posibles problemas o cambios dentro de la organización.

Ventajas

- SOA permite el desarrollo de aplicaciones manejables y más seguras, ya que proporciona una infraestructura y documentación común para desarrollar servicios con la posibilidad de añadir nuevas funcionalidades.
- La arquitectura SOA permite el desarrollo de aplicación en menor tiempo y más económicas, gracias a la integración de todos los datos de manera flexible.

Desventajas

- SOA depende de la implementación de estándares. Sin estándares, la comunicación entre aplicaciones requiere de mucho tiempo y código.
- Implica conocer los procesos del negocio, clasificarlos, extraer las funciones que son comunes a ellos, estandarizarlas y formar con ellas capas de servicios que serán requeridas por cualquier proceso de negocio.

Diagrama

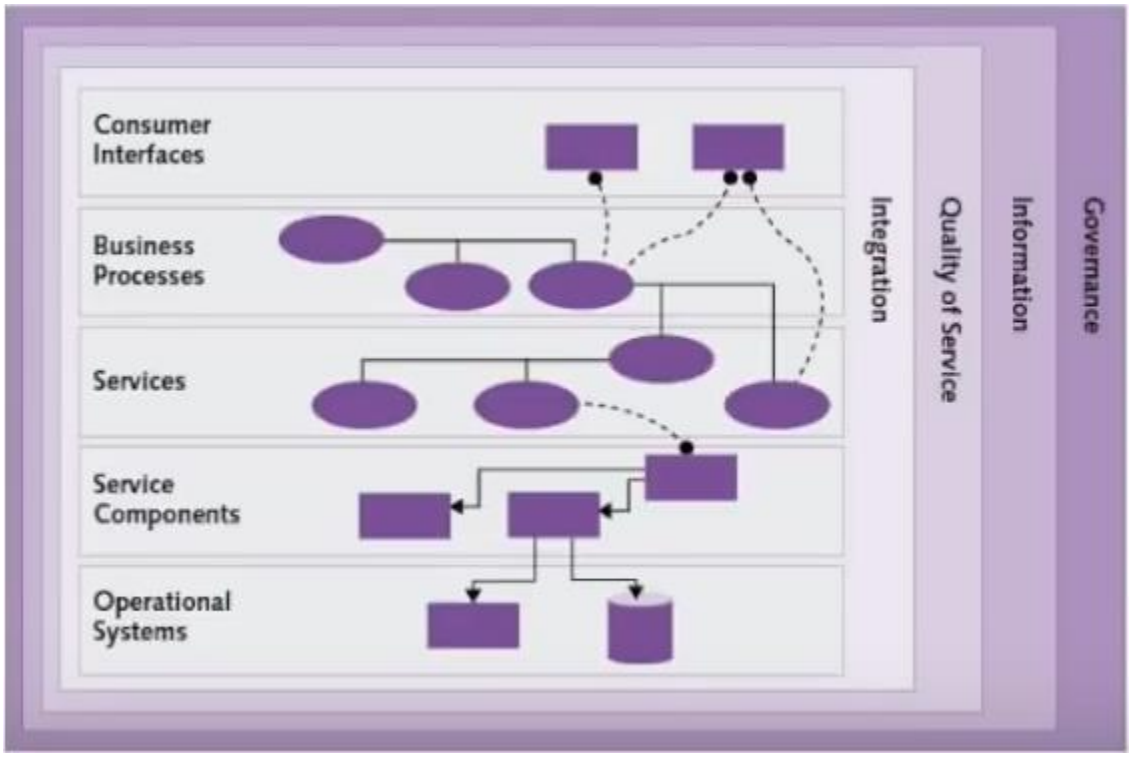
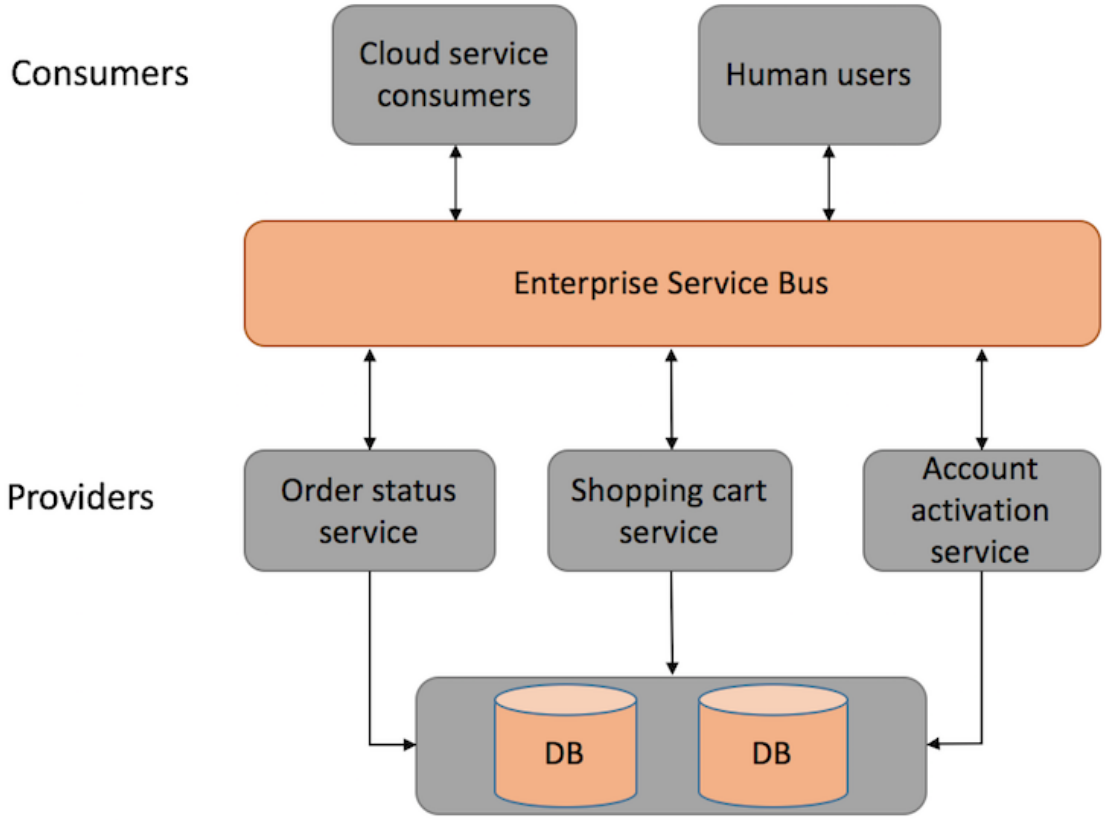


Diagrama uso real



Análisis del patrón

Overall Agility: Alto, por su conjunto de atributos que se aplican para manejar a los proveedores del servicio o a los consumidores.

Ease of Deployment: Bajo, debido a muchos tipos de servicios diferentes, la implementación y la complejidad operativa son una preocupación en esta arquitectura.

Testability: Medio, permite realizar un análisis estadístico de los flujos de negocio reales en base a indicadores clave de negocio, permitiendo la identificación de puntos de mejora a optimizar.

Performance: Bajo, por lo que para garantizar su rendimiento requiere administrar el rendimiento de los componentes, aplicaciones y sistemas que se encuentran debajo de la abstracción de servicios.

Scalability: Alto, ya que la composición de servicios, lo cual permite que los usuarios los utilicen sin importar la implementación o cuestiones no funcionales.

Ease of Development: Alto, porque se ocupa del diseño y desarrollo de sistemas distribuidos para llevar a cabo grandes volúmenes de datos.

Patron Broker:

Este patrón permite desarrollar sistemas distribuidos mediante componentes desacoplados que responden a invocaciones remotas. Un componente bróker es responsable de las comunicaciones, tanto de enviar/reenviar las peticiones, asi como de transmitir los resultados y las excepciones. El patrón Broker comprende seis componentes participantes: clientes, servidores, brokers, puentes, proxies del lado del cliente y proxies del lado del servidor.

Cuando utilizarlo:

Cuando una aplicación puede acceder a los servicios distribuidos enviando mensajes al objeto apropiado, en vez de enfocarse en la comunicación entre procesos de bajo nivel. Además, el patrón Broker es flexible porque permite cambiar, añadir, quitar y reubicar objetos dinámicamente.

Ventajas:

- Como el broker tiene la función de localizar un determinado servidor utilizando un identificador único, los clientes no necesitan conocer la ubicación de los servidores.
- Si los servidores cambian pero sus interfaces permanecen sin cambio, no se tiene impacto funcional sobre los clientes. Modificando la implementación interna del broker, pero no los APIs que éstos proveen, no tiene efecto en clientes y servidores más que cambios en el desempeño.

- Sistemas Broker diferentes pueden interoperar si entienden un protocolo común para el intercambio de mensajes. Este protocolo es implementado y manejado por puentes, los cuales son responsables de traducir el protocolo específico del broker en un protocolo común, y viceversa.

Desventajas:

- Las aplicaciones que utilizan la implementación Broker usualmente son más lentas que las aplicaciones cuya distribución de componentes es estática y conocida.
- El sistema Broker puede ofrecer baja tolerancia a fallos.
- La prueba y debbuging de un sistema Broker es tedioso debido a que el número de componentes implicados es grande.

Diagrama

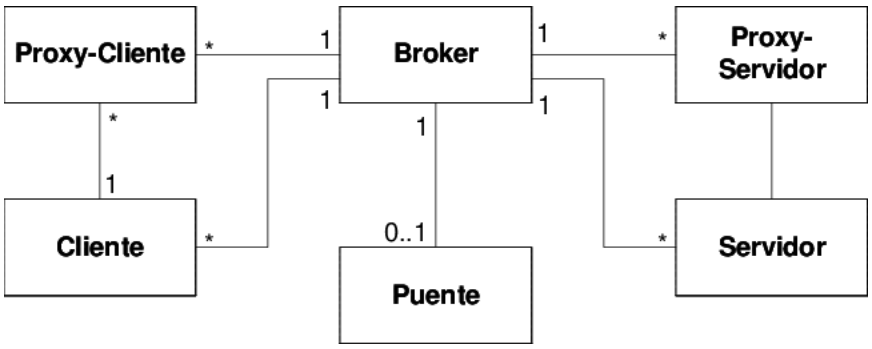
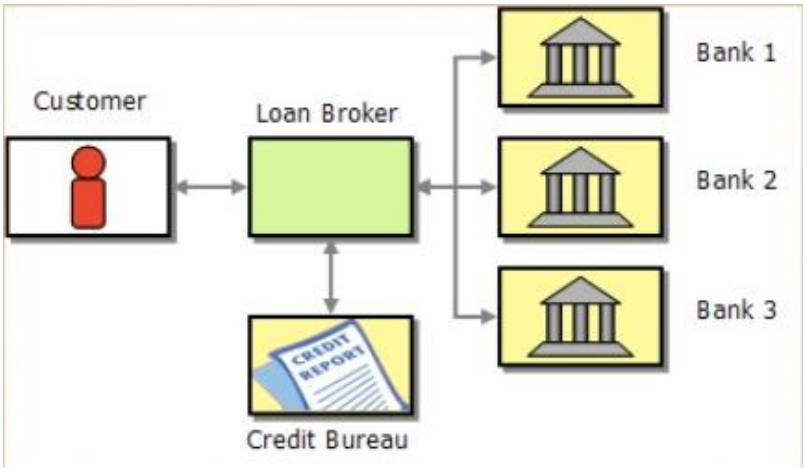


Diagrama uso real



Analisis del patrón:

Overall Agility: Bajo, no es necesario enfocar ningún otro componente que el bróker en la comunicación entre procesos de bajo nivel.

Performance: Medio, pueden convertirse potencialmente en un cuello de botella, especialmente si deben atender altos volúmenes de mensajes y ejecutar la lógica de transformación compleja

Scalability: Alto, porque la agrupación de instancias de Broker hace posible construir sistemas a escala para manejar una alta solicitud cargas.

Ease of deployment: Medio porque una aplicación que utiliza objetos sólo debe ver las interfaces ofrecidas por éstos, sin necesidad de conocer los detalles de su implementación o su ubicación física.

Testability: Alto, por ejemplo la cooperación entre un cliente y un servidor puede fallar por dos razones posibles: el servidor ha entrado en un estado de error, o hay un problema en algún lugar de la ruta de comunicación entre el cliente y el servidor.

Web Grafía

MVC

https://openclassrooms.com/en/courses/4309566-descubre-la-arquitectura-mvc/4942546-el-patron-modelo-vista-controlador-mvc

https://si.ua.es/es/documentacion/asp-net-mvc-3/1-dia/modelo-vista-controlador-mvc.html

https://desarrolloweb.com/articulos/que-es-mvc.html

https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador

http://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/122

https://codigofacilito.com/articulos/mvc-model-view-controller-explicado

Pizarra

http://www.authorstream.com/Presentation/frank4477-1862120-estilos-arquitectonicos-pizarra/

http://sistemascentradosendatos.blogspot.com/2013/10/arquitectura-de-pizarra.html

https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013

https://es.slideshare.net/rehoscript/arquitecturas-de-pizarra-o-repositrio

Tiempo Real

http://magdalyithunid3.blogspot.com/

http://arquitectura-de-software.blogspot.es/1464570537/diseno-de-software-de-arquitectura-de-tiempo-real/

https://ittgweb.wordpress.com/2016/05/29/3-7-diseno-de-software-de-arquitectura-de-tiempo-real/

https://es.wikipedia.org/wiki/Arquitectura_dirigida_por_eventos

https://www.megapractical.com/blog-de-arquitectura-soa-y-desarrollo-de-software/patrones-y-respuesta-en-tiempo-real-en-la-asistencia-medica

https://www.marcoteorico.com/curso/45/ingenieria-de-software/276/diseno-de-software-de-arquitectura-de-tiempo-real

CQRS

https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/cqrs

Event Sourcing

https://docs.microsoft.com/es-mx/azure/architecture/patterns/event-sourcing

https://dzone.com/articles/software-architecture-the-5-patterns-you-need-to-k

SOA

https://www.service-architecture.com/

http://www.opengroup.org/soa/source-book/soa/p4.htm

Broker

https://es.scribd.com/document/143875290/El-Patron-Broker