Arquitectura de Proyectos en IT

Estilos arquitectónicos

Contenido:

Introducción

Un diseño de arquitectura siempre es uno de los factores críticos que determinan el éxito de un sistema de software.
Existen determinadas técnicas a seguir para estandarizar estos factores críticos llamados patrones. Los patrones son soluciones a problemas comunes dentro de un contexto determinado. Es una idea reutilizable, identificando buenas estructuras que se repiten una y otra vez en la práctica.

Hay distintas categorías de patrones en la industria del software. Principalmente:

- Patrones de arquitectura (estilos arquitectónicos)
- Patrones de diseño (soluciones a problemas de diseño de componentes)
- Patrones de codificación (específicos para un lenguaje de programación)
- Antipatrones (soluciones habitualmente usadas pero inefectivas o contraproducentes)

Estilos arquitectónicos

Los estilos arquitectónicos (o patrones de arquitectura) expresan un esquema organizativo fundamental sobre la estructura de un sistema de software. Proporcionan un conjunto de subsistemas predefinidos, especifican sus responsabilidades e incluyen reglas y recomendaciones para organizar las relaciones entre los subsistemas.
La selección de un estilo arquitectónico es una decisión de diseño fundamental cuando se desarrolla un software, ya que afectará a la mayor parte del sistema y sus componentes. Suelen elegirse en una etapa temprana del diseño, ya que cambiarlos posteriormente suele tener un alto costo.

Desde mediados de la década del 90, varios autores se encargaron de clasificar y enunciar estos patrones. Veremos una de estas clasificaciones:

- Estructuración
    - Sistemas de flujo de datos
    - Sistemas de llamada y retorno (Call & Return)
- Sistemas distribuidos
- Sistemas interactivos
- Sistemas adaptables
- Comunicación

Patrones de estructuración

Asumiendo de forma optimista que nuestros requerimientos están bien definidos y son estables, una de las primeras tareas es definir la arquitectura del sistema. En este punto, hay que encontrar una subdivisión de alto nivel del sistema en sus módulos o

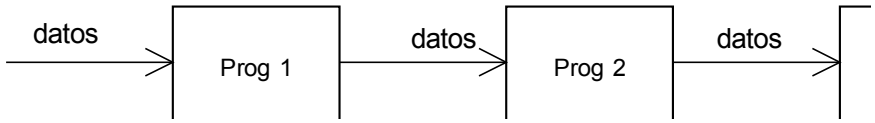
componentes constitutivos. Es decir, convertir ese estado de desorganización a una estructura definida. Los patrones de estructuración aplican en este nivel.

Sistemas de flujo de datos (Dataflow systems)

Se caracterizan en ver al sistema como una serie de transformaciones sucesivas sobre los datos de entrada.
Los datos entran en el sistema y fluyen a través de los componentes independientes uno por uno hasta que se asignan a un destino final. Los conectores tienen solamente el rol de transportar la información. La lógica se aplica en los componentes de procesamiento.

Batch Sequential (Secuencial en lote)

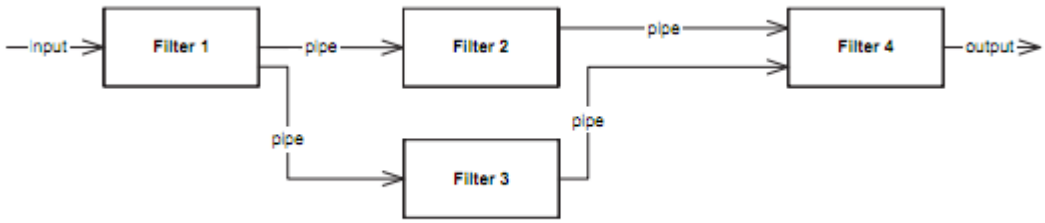
Considera una tarea compleja que puede ser sub-dividida en subtareas más pequeñas. No es conveniente asignar toda la tarea a un único componente ya que tendría una complejidad excesiva. Cada sub-tarea se termina en su completitud y se envía el conjunto total de los datos procesados a la siguiente.
Este patrón es útil para flujos de datos simples y secuenciales.



Pipe & Filters (Tuberías y filtros)

Hace hincapié en la transformación incremental de datos mediante componentes colocados sucesivamente.
Las tuberías (pipes) no tienen estado, pero a diferencia del batch sequential, tienen una mayor importancia que la de transportar datos. Se busca una flexibilidad en la configuración del dataflow, pudiendo cambiar la conexión de una tubería, modificando el flujo de datos de forma distinta.
Los filtros transforman los datos incrementalmente, no retienen información de estado. Además, y a diferencia del patrón anterior, no es necesario que cada tarea se termine completamente, ya que un filtro puede estar tomando los datos del filtro anterior a medida que se generan.
En realidad, la palabra “filtros” no debería ser la correcta, ya que da la idea de componentes que filtran la información, reteniendo una parte y enviando otra. En realidad, en Pipe & Filters, estos componentes vendrían a ser en realidad “transformadores”.

En este patrón, las divisiones y uniones del flujo están permitidas, así como los ciclos (loops) en las tuberías.



Ejemplos: programas de consola para Unix (usando Pipes), compiladores (analizador léxico, parser, analizador semántico, generador de código).

Call & Return systems (Sistemas de llamada y retorno)

Los sistemas de "Call & Return" han sido el estilo arquitectónico dominante en los últimos 30 años. Se caracterizan por ver al sistema como una serie de llamadas a procedimientos o funciones, forma en la que interactúan los componentes del mismo.

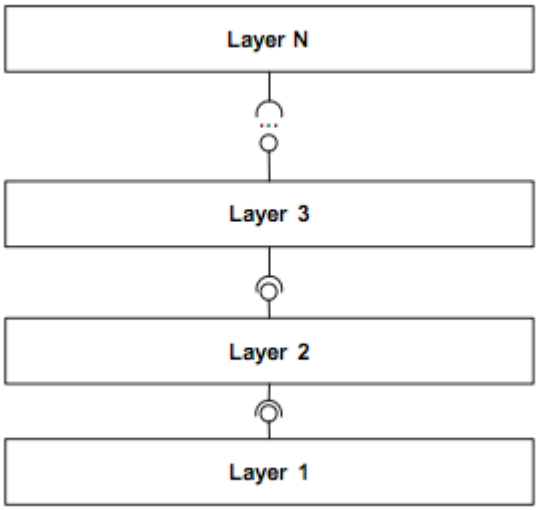
Hierarchical layers (Capas jerárquicas)

Un sistema en capas está organizado jerárquicamente, donde cada capa provee servicios a la capa superior y consume los que brinda la capa inferior.
Existen distintas variantes de esta organización, dependiendo de si una capa puede consumir y brindar servicios sólo a las capas adyacentes, o a todas las superiores/inferiores.

Es una organización muy difundida tanto para sistemas de software como para protocolos de comunicación (OSI, TCP/IP, etc), metodologías, etc.

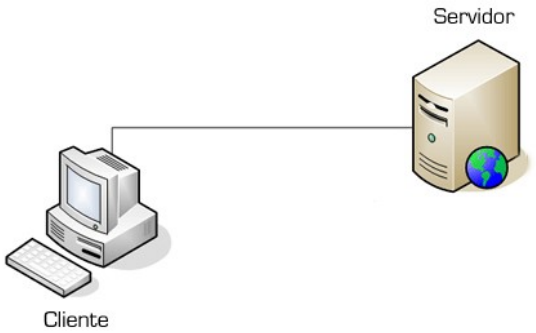
Los sistemas en capas tienen varias ventajas. Primero, soportan un diseño basado en distintos niveles de abstracción. Esto permite a los implementadotes dividir un problema complejo en una secuencia de pasos incrementales. Segundo, soporta el mejoramiento del sistema. Mejoras en una capa, afectan positivamente al resto. Tercero, suponen reusabilidad. Distintas implementaciones de una misma capa pueden ser usadas e intercambiadas de ser necesario. Esto permite definir estándares para las interfaces y luego poder utilizar cualquiera de las implementaciones de una capa. Además, y a diferencia de los Dataflow Systems, se permite una comunicación bidireccional entre capas adyacentes.

Pero esta organización también tiene desventajas. En principio, no todos los sistemas pueden (o convienen) ser estructurados en capas. Es posible que aunque puedan estructurarse así, cuestiones de performance pueden requerir el alto acoplamiento entre las funciones e implementaciones de las capas. Además, puede ser muy difícil encontrar las correctas abstracciones para las capas. Con una organización en capas, es habitual sobrediseñar un sistema en muchas capas, afectando negativamente la performance por la cantidad excesiva de llamados entre las capas. También suele pasar que un cambio en una de las capas, deba ser replicado en todas las demás, haciendo los cambios costosos.



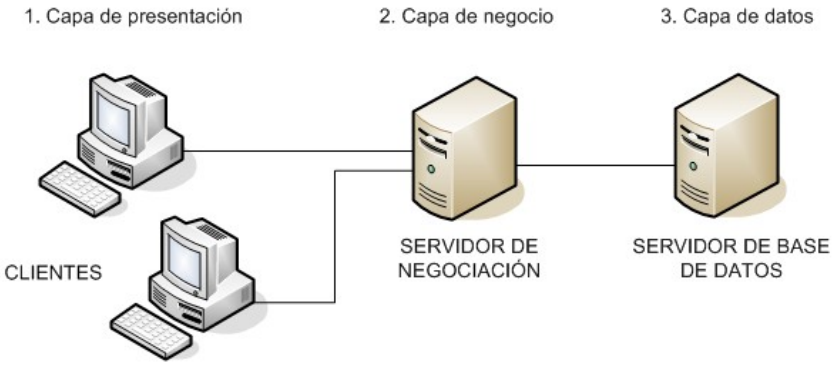
Organizaciones típicas en sistemas software:

Cliente-Servidor (2 capas):



Los servidores centralizan la información y brindan servicios a los clientes, que los consumen. Se utiliza para sistemas interactivos.

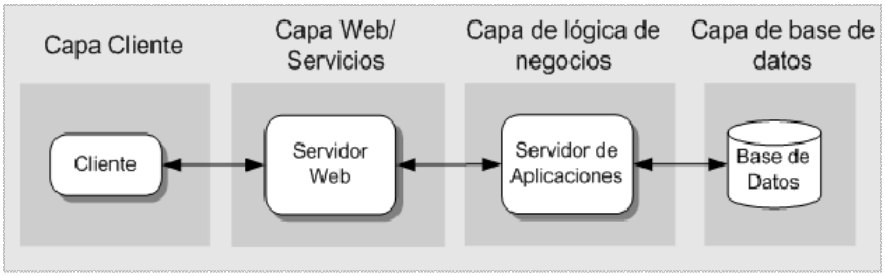
Sistemas en 3 capas:



Habitual organización de software interactivo. La capa de presentación presenta las vistas de los datos a los usuarios y la interacción con los mismos. Un mismo software puede tener distintas formas de presentación. En la capa de negocio se determina la

lógica y reglas directamente relacionadas al dominio de la aplicación. Realiza el principal procesamiento en la aplicación. La capa de datos almacena la información y permite su consulta y modificación. No se debe suponer que siempre estará constituida por una base de datos, ya que puede ser un sistema externo el que cumpla esta función.

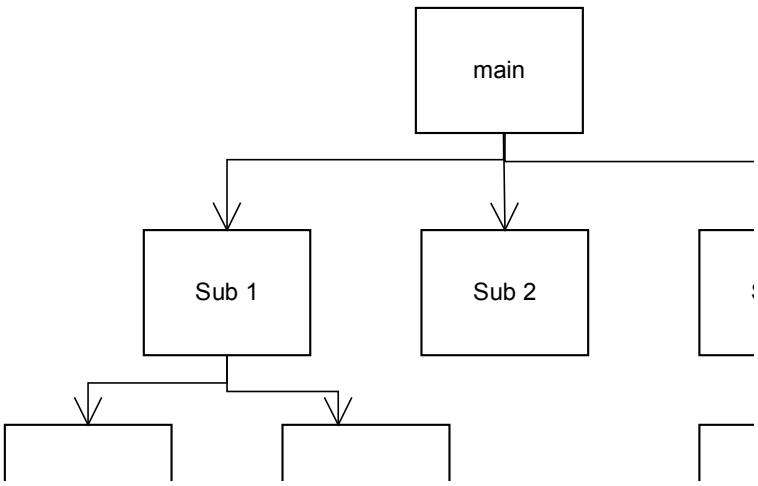
Arquitectura de 4 capas:



En los últimos años, se comenzó a utilizar una capa intermedia entre la capa de negocio y la capa de presentación denominada “capa de servicios”. La principal motivación es proveer una interfaz simple para el acceso a la capa de negocios desde distintas formas de presentación.
Muchas veces, las aplicaciones pueden accederse desde PCs, móviles, sistemas externos y otras formas de presentación (que muchas veces acceden utilizando distintos protocolos y tecnologías), y se quiere transparentar la capa de negocios con una interfaz simple.

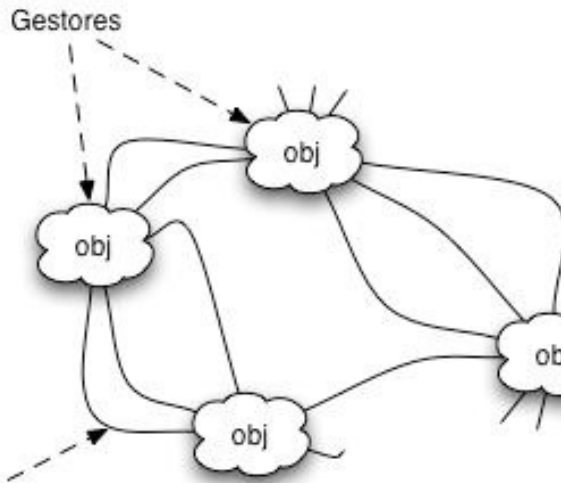
Main Program / Subroutines (Programa principal y subrutinas)

Es el paradigma de programación clásico y estructurado, que consiste en descomponer un programa jerárquicamente en piezas más pequeñas para ayudar a mejorar la modificabilidad del sistema.
Se utilizó ampliamente hasta los años 90, donde el paradigma de objetos comenzó a ganar terreno y convertirse en el paradigma dominante.



Object-Oriented Systems (Sistemas orientados a objetos)

Una evolución del paradigma de tipos abstractos de datos, se centra en la encapsulación de los datos y del conocimiento acerca de como manipular y acceder a dichos datos.
Un objeto oculta su representación de sus clientes lo que permite cambiar la implementación del mismo sin afectar a quienes lo referencian.
Se distinguen de los Tipos de Datos Abstractos por la utilización de herencia (compartiendo jerárquicamente las definiciones y código) y por el polmorfismo (determinar la semántica de una operación en tiempo de ejecución).



Patrones sobre sistemas distribuidos

Los sistemas distribuidos necesitan claramente un software distinto que los sistemas centralizados. Generalmente, las tecnologías y patrones utilizados son distintos también.
El crecimiento de las redes de información y la mejora sustancial de la comunicación entre los componentes de esas redes, dio lugar a sistemas completos de software distribuidos en la red, sin un punto único de centralización de la información.
Aunque en esta categoría solo incluimos el patrón Broker, generalmente se ven aplicados también patrones como Pipe & Filters (ver patrones de estructuración) o Microkernel (ver patrones para sistemas adaptables) en los sistemas distribuidos.

Broker (Agente intermediario)

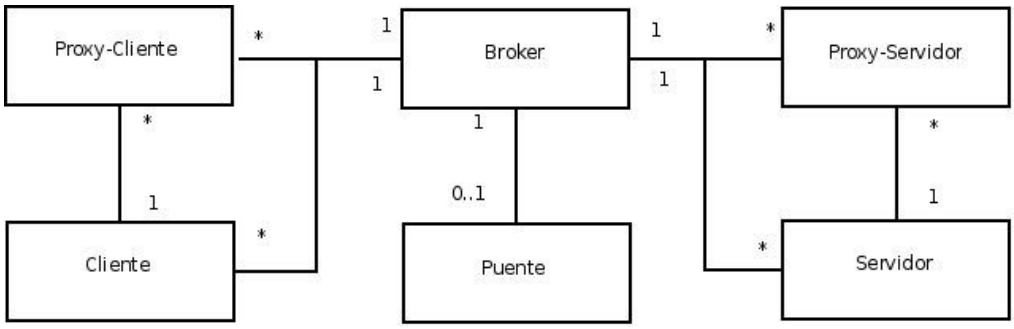
El patrón Broker puede ser usado para estructurar sistemas de software distribuidos con componentes desacoplados que interactúan mediante invocaciones a servicios remotos.
Un componente Broker es responsable de la coordinación de la comunicación, como el envío de peticiones y la transmisión de respuestas y excepciones.

El patrón Broker comprende seis componentes principales: clientes, servidores, brokers, puentes, proxies del cliente y proxies del servidor.

Los servidores se registran en el broker y hacen disponibles sus servicios a través de sus interfaces (proxies). Los clientes acceden a esas funcionalidades a través de sus proxies, enviando la solicitud via brokers. Las tareas del broker incluyen localizar el servidor apropiado, enviarle la solicitud, y transmitir resultados y excepciones, de regreso, al cliente.

Utilizando el patrón Broker, 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.

El patrón Broker reduce la complejidad en el desarrollo de aplicaciones distribuidas porque hace que la distribución sea transparente al desarrollador, mediante la introducción de un modelo de objetos en el que los servicios distribuidos se encapsulan en objetos. Por lo tanto, los sistemas Broker ofrecen una ruta para la integración de dos tecnologías: distribución y objetos. Asimismo, los sistemas Broker extienden los modelos de objetos desde aplicaciones individuales hasta aplicaciones distribuidas que constan de componentes desacoplados que pueden ejecutarse en máquinas heterogéneas, y que pueden escribirse en lenguajes de programación diferentes.



Ejemplos: CORBA, World Wide Web (Browsers web como brokers y los servidores WWW toman el rol de Service Providers).

Patrones sobre sistemas interactivos

La mayoría de los sistemas actuales tienen algún grado de interacción con el usuario, principalmente ayudados por las interfaces gráficas modernas. El objetivo de esto es incrementar la usabilidad de la aplicación.
El objetivo principal del diseño de estos sistemas es mantener el núcleo de la funcionalidad separado de la interfaz de usuario. Generalmente, las interfaces de usuario sufren muchas modificaciones y adaptaciones, por lo que esta separación es fundamental para no afectar constantemente a los componentes encargados del núcleo funcional del software.

El patrón arquitectural más significativo de esta categoría es el MVC, pero existen varios estilos que se aplican a software de estas características.

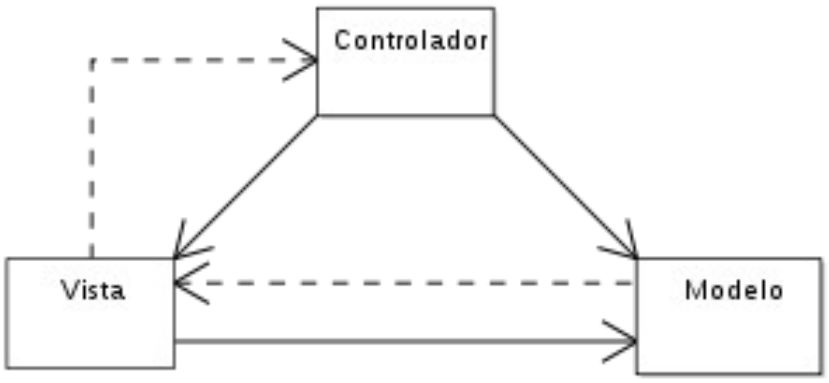
MVC – Model View Controller (Modelo - Vista - Controlador)

El patrón MVC (Modelo – Vista – Controlador) fue originalmente descrito en 1979 para una investigación en el lenguaje Smalltalk. Este patrón tuvo muchísimo éxito en el desarrollo de software interactivo, y se convirtió en el principal estilo arquitectónico de este tipo de aplicaciones en los últimos años. Prácticamente todos los frameworks estructurales actuales permiten (o directamente obligan) la modelización de las aplicaciones bajo este patrón.

El MVC divide una aplicación interactiva en 3 componentes. El modelo contiene la principal funcionalidad y la información, la vista muestra la información al usuario, y los controladores manejan las entradas del usuario. Las vistas y controladores forman la interfaz de usuario (UI: user interface).

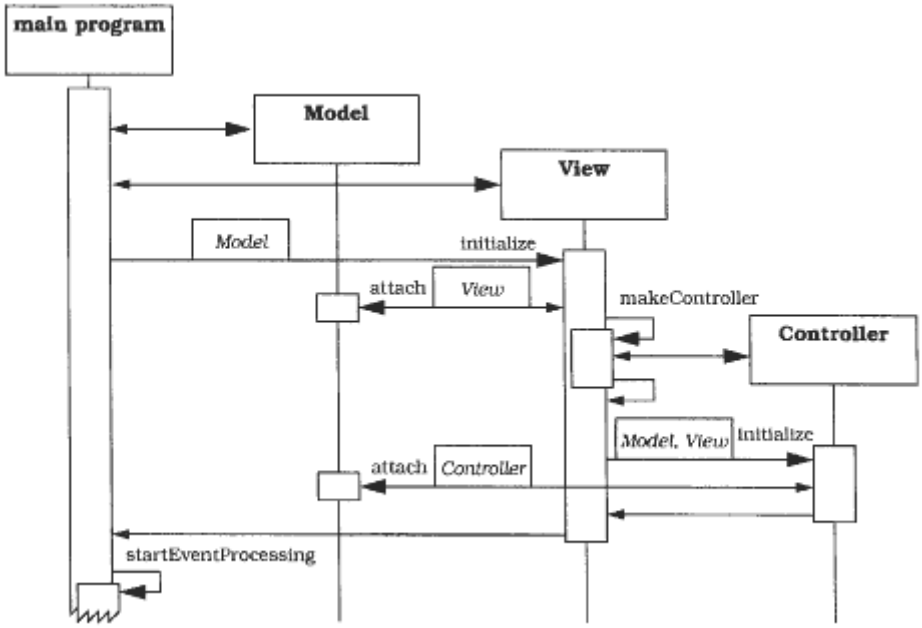
El problema que intenta resolver el MVC es el siguiente: habitualmente, una de las partes que más cambian es la vista al usuario. Además, la misma información puede mostrarse de distintas formas (tablas y listados, gráficos diversos, estadísticas, etc). La idea principal entonces es separar totalmente el modelo de datos de las vistas que puede llegar a tener, y tener los controladores como forma de integración entre ambas.

La estructura básica del patrón es:

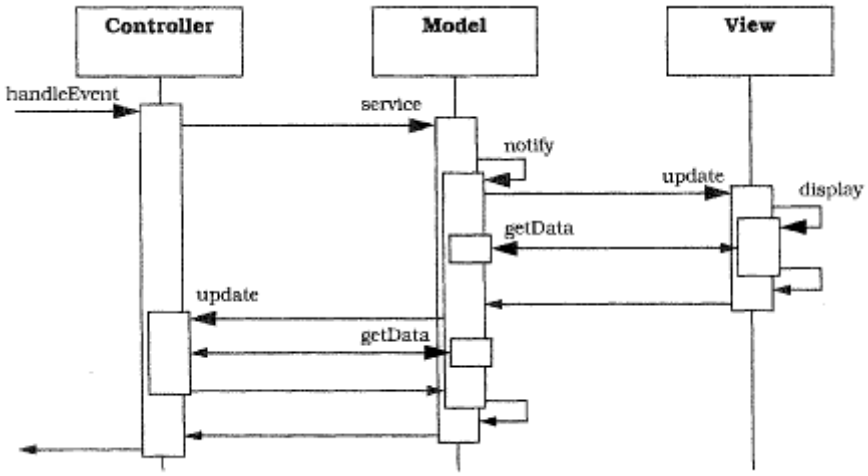


Claro está que la aplicación no debe contar únicamente con 3 componentes (por ejemplo: 3 clases). Estos 3 son tipos de componentes, y todas las clases que sean de un tipo, tienen que cumplir los requisitos y tener las responsabilidades del tipo al que pertenecen.

Este patrón generalmente es inicializado por un programa principal, que no pertenece a ninguno de los tres tipos. Este programa principal es el encargado de inicializar el modelo y las vistas, mientras que los controladores son creados por las vistas mismas. Un diagrama de secuencia especifica como suele ser esta inicialización:

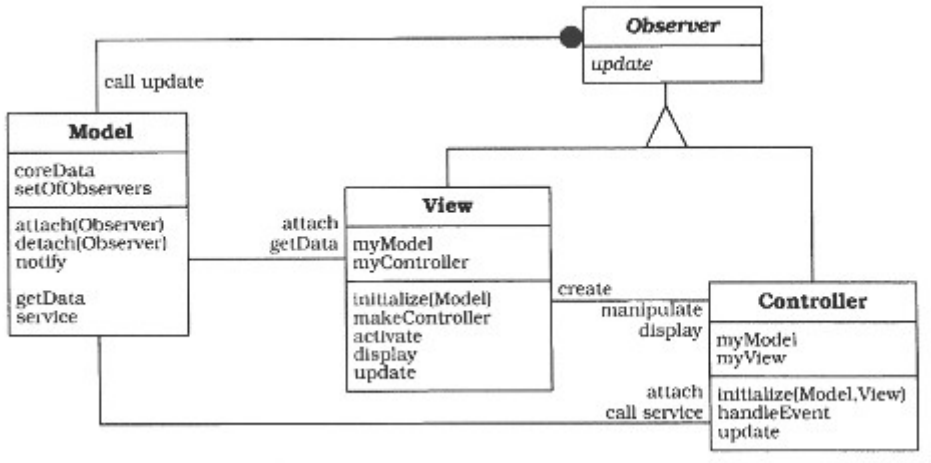


Una vez que están creados los componentes, el controlador es el responsable de recibir los comandos del usuario. El controlador modifica el modelo según esas entradas, que terminan actualizando las vistas y controladores asociados. Las vistas luego vuelven a pedir la información al modelo para redibujarse, y los controladores hacen lo propio para saber que comandos estarán disponibles y cuales no. El proceso básico sería:



Uno de los problemas visibles rápidamente con el patrón MVC, es que al existir muchas vistas activas para un único modelo, cuando este último cambia, deben modificarse todas las vistas.
Una variante del patrón MVC clásico es combinando este estilo arquitectónico con un patrón de diseño muy conocido: el Observer (observador). El Observer es análogo al patrón Publisher-Subscriber (ver sección “Comunicación”).
Este patrón de diseño permite invertir el modelo natural de avisos, en los cuáles el que avisa conoce a los objetos a los que avisará. Usando Observer, los observadores se enlistan en una estructura interna del avisador, para que cuando surja un evento, este avise a toda su lista de observadores, de forma elegante y desacoplada.

La variante del patrón MVC + Observer es muy usual actualmente y se estructura así:



Cuando una vista o un controlador son creados, se les pasa una referencia al modelo que corresponde, y estos se ocupan de alistarse en una estructura interna de observadores que tiene ese modelo.
Cuando un controlador toma un comando del usuario y al impactar en el modelo hace que sus datos cambien, el modelo envía un mensaje de "actualización" a todos los observadores que tiene en su lista. Esto desacopla al modelo de sus observadores, ya que no conoce al componente de vista o controlador específicamente, sino únicamente a su interfaz común: el Observer.
De esta forma, los cambios en el modelo se propagan elegantemente en las vistas y controladores asociados.

Ventajas del patrón:

- Múltiples vistas del mismo modelo
- Vistas sincronizadas
- Cambios de vistas y controles en tiempo de ejecución
- Cambio del aspecto externo de las aplicaciones
- Base potencial para construir un Framework

Inconvenientes posibles del patrón:

- Se incrementa la complejidad
- Número de actualizaciones potencialmente alto
- Íntima conexión entre la vista y el controlador
- Alto acoplamiento de las vistas y los controladores con respecto al modelo
- Acceso ineficiente a los datos desde la vista
- Cambio inevitable de las vistas y controladores cuando se porte a otras plataformas.

Frameworks MVC actuales:

Son muchos los frameworks actuales para aplicaciones interactivas que utilizan el patrón MVC como arquitectura básica de modelo.
Los más conocidos son:

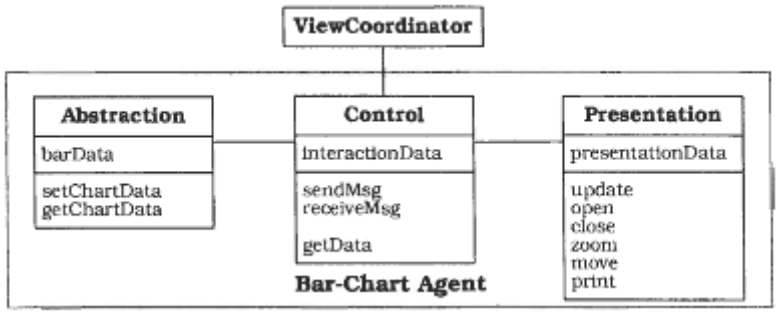
Lenguaje	Framework MVC
Java / J2EE	SpringMVC
Java / J2EE	Struts
Java / J2EE	JSF
Java / J2EE	Grails
Ruby	Ruby on Rails
PHP	Zend Framework
PHP	CakePHP
PHP	Symfony
PHP	Yii PHP
Python	Django
.NET	Spring .NET
.NET	Maverick .NET
.NET	ASP.NET MVC
Adobe Flex	Cairngorm

De todos modos, generalmente en aplicaciones web no es MVC el patrón que realmente se utiliza, sino una variante del mismo llamada MVP (Model – View – Presenter) donde el View combina los roles de Vista y Controlador del MVC, y el presenter trae la información del modelo, la persiste y la formatea para la vista. El modelo solo brinda una interfaz de la información que será presentada.

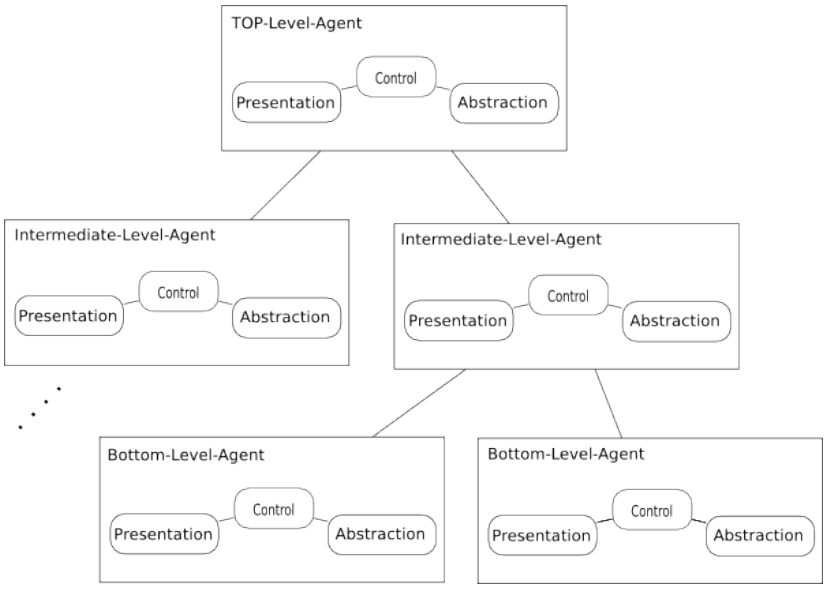
PAC – Presentation Abstraction Control

El patrón PAC (Presentación – Abstracción – Control) define una estructura para sistemas interactivos como una jerarquía de componentes cooperativos.
Cada componente (o agente) es responsable de un aspecto específico de la funcionalidad de la aplicación y consiste de 3 componentes: presentación, abstracción y control.

Esta subdivisión separa el aspecto de interacción usuario-sistema de su funcionalidad principal y de la comunicación con otros agentes.



Estructuras como éstas, para cada agente, se distribuyen en una jerarquía de agentes de habitualmente 3 ó 4 niveles.



Ejemplos de uso: administración de tráfico en red, aplicación de robots móviles.

Patrones sobre sistemas adaptables

Los sistemas evolucionan con el tiempo. Nuevas funcionalidades se agregan y los servicios existentes cambian. Deben soportar nuevas versiones de sistemas operativos, plataformas de UI o componentes y librerías de terceros. Adaptación a nuevos estándares o hardware también es necesaria. “Diseñar para el cambio” es un concepto importante al diseñar una arquitectura de software.

Explicaremos dos patrones aplicables a estos sistemas: Microkernel y Reflection.

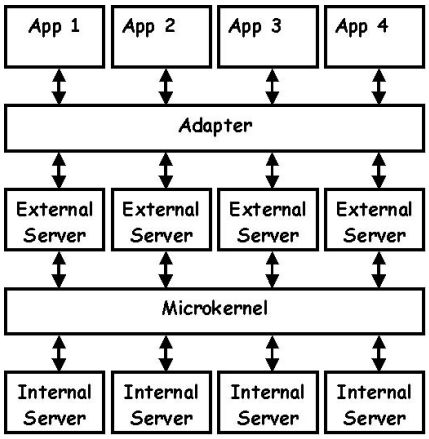
Microkernel

El patrón microkernel aplica claramente a sistemas de software que son susceptibles a cambios en el tiempo.
Separan la mínima funcionalidad principal (el core) de la funcionalidad extendida y las partes específicas del cliente. También sirve como un socket para encastrar estas extensiones y coordinar su colaboración.

Durante el diseño de un sistema aplicando el patrón microkernel, es fundamental encontrar las funcionalidades esenciales que necesitarán la gran mayoría de las extensiones. Estas funcionalidades formarán parte del “microkernel” y brindarán servicios básicos hacia el resto de los componentes. Se buscará un equilibrio en el número de funcionalidades. Es necesario que el tamaño del microkernel no sea muy grande, pero tiene que cubrir todas las necesidades básicas de las extensiones.

El microkernel provee dos tipos de componentes “servidores”. Unos internos, que extienden la funcionalidad que provee el microkernel, y unos externos que proveen

interfaces de programación a los clientes. Los “servidores” internos sólo son utilizados por los externos a través del microkernel como una extensión de este. Los externos resuelven a través del microkernel la funcionalidad invocada por los clientes. El cliente no interactúa directamente sobre los servidores externos, sino que lo hace a través de adapters.



Ejemplos de uso: Sistemas operativos y sus API

Reflection

El patrón “reflection” provee un mecanismo para cambiar la estructura y comportamiento de un software de manera dinámica. Soporta la modificación de aspectos fundamentales, como los tipos de estructura y el mecanismo de llamados a funciones.
En este patrón, la aplicación se divide en dos partes. Un metanivel provee información acerca de propiedades y comportamiento particulares del sistema y posibilita modificarlas. Un nivel de base incluye la lógica de la aplicación, pero sigue lo especificado en el metanivel. Cambios en el metanivel impactan en el nivel base.

A veces sólo se utiliza el metanivel para observar el nivel base, pero no modificarlo. Otras veces, los cambios están permitidos. Generalmente, la reflexión es dinámica (en tiempo de ejecución).

Lenguajes de programación como Java, C# o Python disponen de un metanivel para sus aplicaciones utilizando el patrón Reflection.

Comunicación

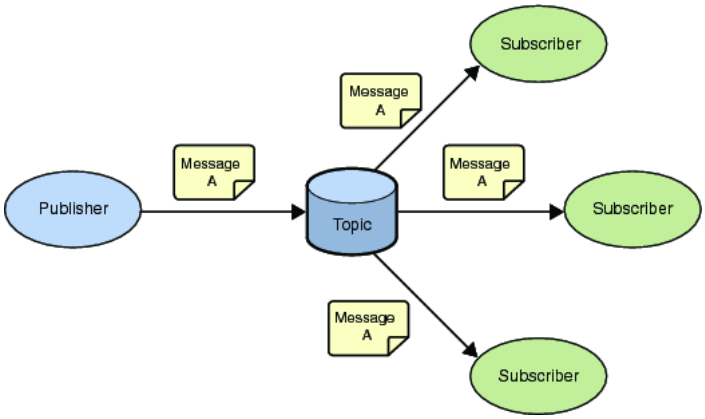
Sólo pocos sistemas de software actuales se ejecutan actualmente en una única computadora. La mayoría utiliza redes de información.
La necesidad de distribuir los componentes de un software en una red, implica el requerimiento de la comunicación entre los mismos.

Existen varios patrones que aplican a la mejora del diseño de comunicación entre estos componentes.

Publisher – Subscriber (Productor - Consumidor)

El patrón "publisher-subscriber" ayuda a mantener el estado de sincronización entre componentes cooperativos. Para lograr esto permite la propagación unidireccional de los cambios: un publicador notifica a varios suscriptores sobre un cambio en su estado.

En este patrón, un componente toma el rol de "publisher" (productor). Todos los componentes interesados en el cambio de estado del Publisher toman el rol de "subscribers" (consumidor).
El publisher mantiene un registro de sus componentes suscriptos. Cuando un componente desea ser suscriptor, utiliza una interfaz de suscripción que provee el Publisher. Cuando se produce un cambio en el Publisher, este notifica a todos los suscriptores que mantenga en su registro interno.
Este patrón es análogo al patrón de diseño "Observer".

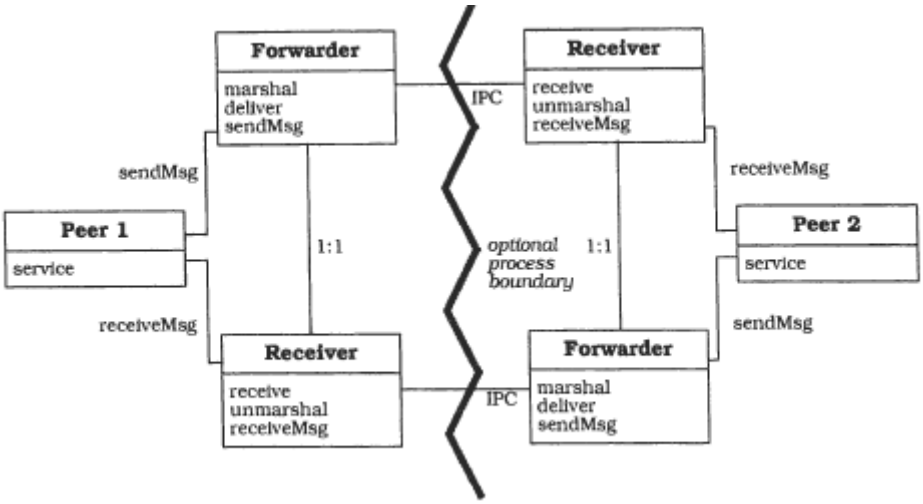


Este patrón se utiliza mucho en el diseño de componentes como patrón "Observer" y es ampliamente utilizado también en soluciones de mensajería con el concepto de "tópico".

Forwarder-Receiver (envío y recepción)

El patrón "forwarder-receiver" provee una comunicación transparente entre procesos para sistemas de software con un modelo de interacción peer-to-peer (red entre pares). Introduce forwarders y receivers para desacoplar los componentes "peers" de los mecanismos de intercomunicación.

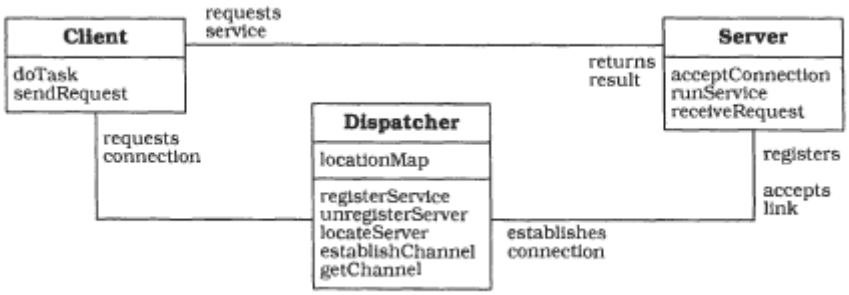
Los componentes forwarder y receiver se encargan de toda la lógica de comunicación entre los peers, como el envío y recepción de información, la comunicación con los peers, el manejo de red, el "marshalling" y "unmarshalling" de datos, etc.



Client-Dispatcher-Server (cliente, despachante y servidor)

El patrón "client-dispatcher-server" introduce una capa intermedia entre clientes y servidores: el "dispatcher". Provee transparencia en la locación de componentes remotos y esconde detalles del establecimiento de la conexión entre ambos componentes.

El dispatcher permite a los servidores registrarse. Los clientes pueden obtener canales de comunicación con los servidores mediante el dispatcher, que establece la comunicación con el servidor para ese canal. Luego, con ese canal ya creado por el dispatcher, el cliente puede interactuar con el servidor directamente.



Ejemplo de uso: Las RPC de Java (Remote Procedure Calls) se basan en este patrón, utilizando los procesos portmappers remotos como dispatcher.

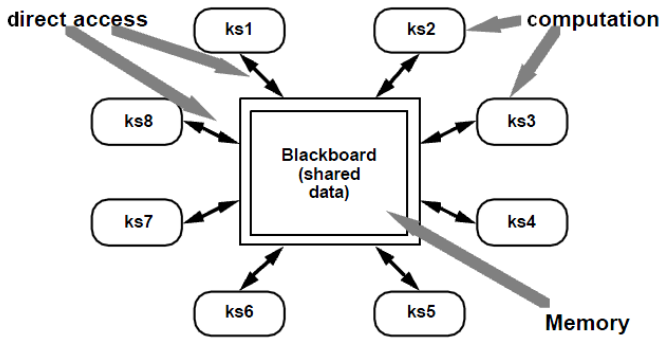
Otros estilos

Existen muchos otros estilos arquitectónicos descritos a lo largo de los años por distintos autores especializados en el tema.

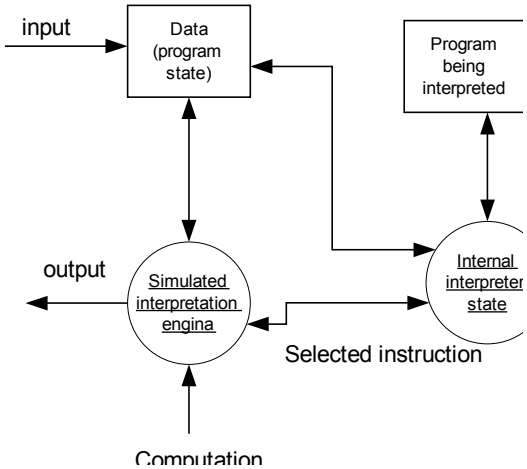
Algunos patrones bastante aceptados y reconocidos son:

- Sistemas basados en eventos
- Repositorios y blackboards
- Intérpretes y basados en tablas
- Máquinas virtuales (Virtual Machines)
- Sistemas basados en reglas (Rule Based)
- Patrones sobre Web Servers
- Patrones para concurrencia

Blackboard:



Intérpretes:



Bibliografía

- **"Pattern Oriented Software Architecture: A System of Patterns"**. John Wiley & Sons, 1996. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.
- **"Software Architecture: Perspectives on an Emerging Discipline"**. Prentice-Hall, 1996. Shaw, M., Garlan, D.
- **"Patterns of Enterprise Application architecture"**. Addison-Wesley, 2002. Fowler, M.
- **"Design Patterns. Elements of Reusable Object-Oriented Software"**. Addison-Wesley, 1995. Gamma, E., Helm, R., Johnson, R., Vlissides, J.