élogos

LA ARQUITECTURA MODELO VISTA CONTROLADOR

élogos

LA ARQUITECTURA MODELO VISTA CONTROLADOR

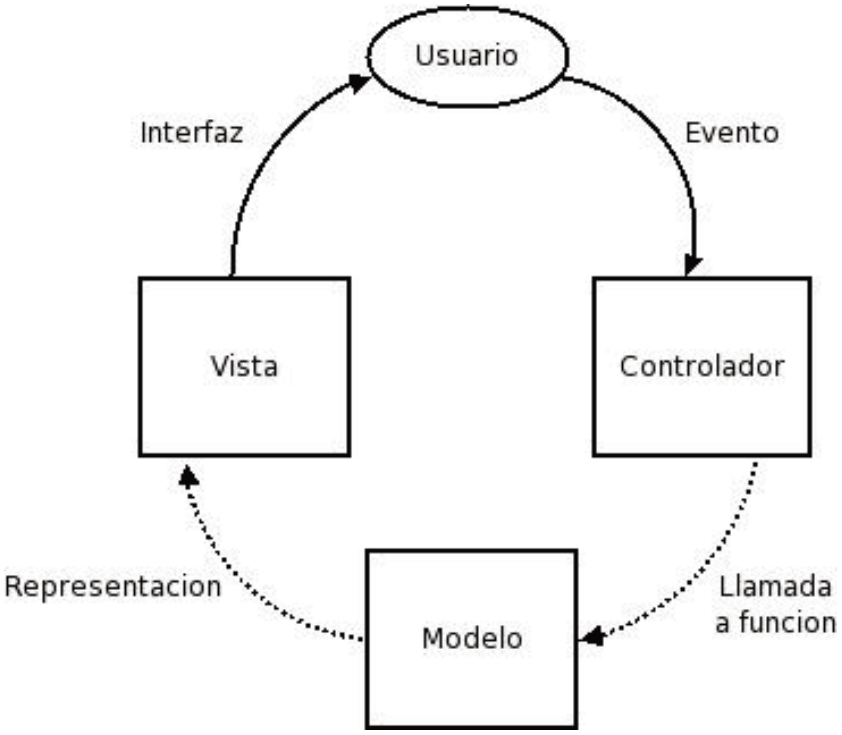
Patrones de diseño Java EE

Vamos a ver en este pdf el funcionamiento de los principales patrones de diseño divididos por capas, para ello nos hemos basado en el Catálogo Oficial de Patrones de J2EE presentado en el Sitio Web Oficial de Sun Microsystems .

Veremos patrones divididos según las 3 capas , la de presentación , la de la lógica d enegocio y la finalmente la capa de datos.

Los patrones de la capa de presentación contienen toda la información relacionada con los servlets y tecnología JSP .

Los patrones de la capa de Lógica de Negocios, contienen toda la información relacionada con los Enterprise Java Beans. Por ultimo, los patrones de la capa de Integración contienen toda la información relacionada con el Java Message Service y la tecnología para conexión con base de datos de java JDBC.



LA ARQUITECTURA MODELO VISTA CONTROLADOR

El catálogo de patrones de J2EE, esta basado en las capas numeradas , esta separación permite que los patrones diseñados para cada componente puedan enfocarse en aspectos específicos del desarrollo de la aplicación, además de proveer una mayor modularidad y la posibilidad de realizar cambios en un componente determinado sin afectar a los otros componentes.

A continuación se describirá el funcionamiento de cada uno de los patrones y posteriormente se planteará una posible adaptación de los que mas se adecuen a la aplicación que se pretende desarrollar.

Patrones de la Capa de Presentación

Patrón Decorating Filter

La mayoría de los sistemas existentes en Internet manejan solicitudes web. El procesamiento de dichas solicitudes involucra un número variable de servicios en el sistema.

El problema central en el cual se enfoca este patrón es el pre-procesamiento y post-procesamiento de peticiones, por lo cual se debe proveer un mecanismo que permita añadir y remover componentes que sirvan para realizar el procesamiento de las peticiones que llegan al sistema.

Centralización de la lógica de servicios comunes.

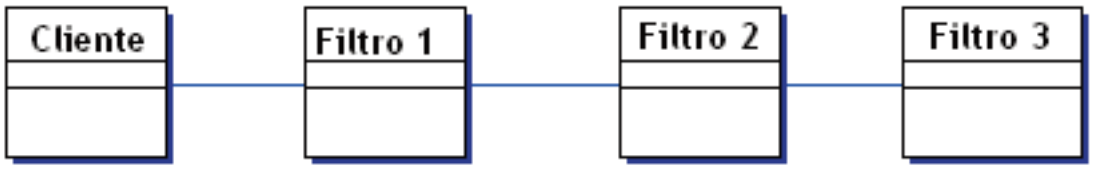
Facilidad al momento de añadir o remover servicios al sistema lo cual permite que estos puedan ser usados en una gran variedad de combinaciones de forma tal que no afecten a otros componentes como el de autenticación o registro de actividades (logging).

Procesamiento completo por solicitudes, por ejemplo, el servicio de seguridad del sistema debe chequear todo lo relacionado con la autenticación del sistema cada vez que esto sea necesario.

Una vez mencionados los puntos centrales que deben ser cubiertos por el patrón se procederá a describir la solución que propone el patrón.

La idea principal de esta solución es la de crear una serie de filtros que se conecten entre sí para realizar el procesamiento de servicios comunes del sistema en una forma estándar sin la necesidad de cambiar el código que se encuentra en el núcleo de la aplicación para el procesamiento de solicitudes, con esto se pretende “decorar” el procesamiento principal con una variedad de servicios comunes, tales como seguridad, registro de logs, depuración, etc. Estos filtros son componentes independientes del código central de la aplicación, lo cual permite que puedan ser removidos o añadidos fácilmente. Por ejemplo, un archivo de configuración de despliegue o publicación puede ser modificado con la finalidad de configurar una serie de filtros. El mismo archivo de configuración podría incluir un mapeo de URL’s específicos para el conjunto de filtros mencionados anteriormente. Cuando un cliente solicita un recurso que coincide con los URL’s que se han configurado, cada uno de los filtros es procesado antes que se retorne el recurso solicitado.

Esquema del funcionamiento del patrón “Decorating Filter”:



Este patrón provee un lugar central para manejar los servicios y lógica de negocios a lo largo de múltiples solicitudes. Existe otro patrón denominado Front Controller[i] que también permite obtener un control centralizado pero tiene la desventaja de no permitir la separación de servicios comunes no relacionados entre sí, tales como autenticación, registro de logs, cifrado de datos, etc.

Reusabilidad

Aumenta la facilidad de particionar aplicaciones e incrementa la reusabilidad. Los filtros “decoradores” son fáciles de añadir o remover del código de manejo de solicitudes existente. Pueden ser usados en cualquier combinación y fácilmente reutilizados para múltiples presentaciones.

Numerosos servicios pueden ser combinados de muchas maneras sin necesidad de cambiar el código principal de la aplicación.

Aumento de la cohesión y disminución de la dependencia entre componentes

Esto es debido a que cada componente es escrito independientemente uno del otro.

La información compartida entre filtros puede ser ineficiente, debido a que cada filtro es independiente uno del otro. Si grandes cantidades de información deben ser compartidas entre filtros puede resultar muy costoso el uso de este esquema.

Patrón Front Controller

Este patrón surge con la finalidad de proveer un mecanismo que permita a la capa de presentación controlar y coordinar el procesamiento de cada usuario a lo largo de múltiples solicitudes, además debe permitir administrar dichos mecanismos de una forma centralizada o distribuida.

LA ARQUITECTURA MODELO VISTA CONTROLADOR

El problema principal que debe resolver este patrón es que el sistema requiere un punto de acceso centralizado para el manejo de solicitudes de la capa de presentación para con ello, ayudar a la integración de los servicios del sistema, recuperación de contenido, manejo de vistas y navegación. Cuando el usuario accede a una vista directamente sin pasar a través de un mecanismo centralizado, pueden ocurrir dos tipos de problemas:

Cada vista es requerida para proveer sus propios servicios de sistema, lo cual termina resultando a menudo en duplicación de código.

La navegación visual es delegada a las vistas.

Adicionalmente, el control distribuido es más difícil de mantener que el centralizado, ya que por lo general es necesario realizar cambios en múltiples lugares con el control distribuido.

La lógica es manejada de una mejor manera en un módulo central en lugar de replicar el código de lógica de negocios en cada vista.

Los puntos de decisión funcionan de acuerdo a la frecuencia de recuperación y manipulación de los datos.

Se utilizan múltiples vistas para responder a solicitudes similares.

Un punto de contacto centralizado para el manejo de una solicitud permite mejorar la facilidad de controlar procesos tales como el control y registro del progreso de un usuario en un site.

Los servicios del sistema y la lógica de manejo de vistas son relativamente sofisticados.

La solución propuesta es la de usar un controlador como el punto inicial de contacto para el manejo de una solicitud. El controlador administra el manejo de una solicitud, incluyendo los servicios de seguridad tales como autenticación y autorización, delega procesos de lógica de negocios así como también la selección de una vista apropiada, manejo de errores y estrategias de creación de contenidos.

El controlador provee un punto de entrada centralizado que controla y administra el manejo de solicitudes Web, por medio de controles y puntos de decisión centralizados, el controlador además ayuda a reducir la cantidad de código Java, que se encuentra contenido dentro de una página JSP.

La centralización del control provista por el controlador y la reducción de la lógica de negocios en las vistas incrementa la reusabilidad de código a lo largo de las solicitudes, lo cual es una aproximación preferible a la alternativa de la inclusión de código en múltiples vistas debido a que esto puede ocasionar que la aplicación este mas propensa a errores.

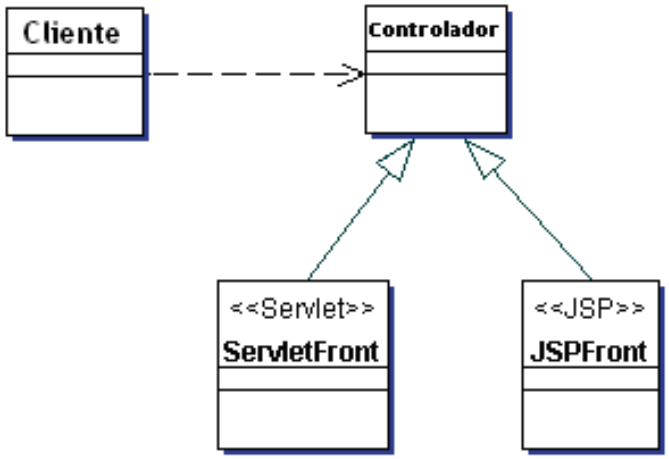
LA ARQUITECTURA MODELO VISTA CONTROLADOR

Típicamente, un controlador se coordina con un componente despachador. Los despachadores son responsables del manejo de las vistas y de la navegación. Así, un despachador escoge la próxima vista para el usuario y los vectores de control para el recurso. Los despachadores pueden estar encapsulados dentro del controlado directamente o pueden ser colocados en un componente diferente.

Aunque el patrón "Front Controller" propone la centralización del manejo de todas las solicitudes, esto no limita el número de manejadores en el sistema, tal y como lo haría un patrón de tipo Singleton.

Una aplicación puede usar múltiples controladores en un sistema, cada unos de estos mapea un conjunto de servicios diferentes.

El siguiente diagrama de clases ilustra la estructura del patrón "Front Controller":



Un controlador provee un lugar central para manejar los servicios del sistema y la lógica de negocios a lo largo de múltiples solicitudes. El acceso centralizado a una aplicación implica el hecho de que las solicitudes sean fácilmente seguidas y registradas. La desventaja principal del esquema centralizado es la poca tolerancia a fallos que posee.

Patrón Helper View

Los Sistemas manejan solicitudes Web. Los procesos de la capa de presentación requieren la generación de una vista basada en una plantilla y un modelo dinámico.

Los cambios de la capa de presentación ocurren a menudo y son difíciles de desarrollar y mantener, debido a la combinación de la lógica de negocio para el acceso a los datos y la lógica de formateo de la presentación, esto hace que el sistema sea menos flexible, menos reutilizable, y menos adaptable a los cambios.

LA ARQUITECTURA MODELO VISTA CONTROLADOR

Mezclar la lógica de negocio con el procesamiento de las vistas también reduce la modularidad y disminuye la separación de roles a la hora de asignar tareas a los miembros de un equipo de desarrollo de software o de aplicaciones web.

Incluir lógica de negocio en las vistas promueve el tipo de reuso estilo “copiar y pegar”. Esto puede ocasionar problemas y errores debido a que un trozo de la lógica de negocios es reutilizado en la misma o en diferentes vistas simplemente colocando al código duplicado en lugares diferentes.

Es deseable incrementar una separación transparente para que cada individuo del equipo de desarrollo pueda cumplir a cabalidad las labores relacionadas con su rol dentro del equipo de trabajo.

Una vista es comúnmente usada para responder a una solicitud particular.

Las vistas delegan el manejo de contenido a sus ayudantes (helpers), los cuales sirve como modelo de datos y adaptadores para los datos. La lógica de presentación es encapsulada en el ayudante y se sitúa entre la vista y la capa de negocios.

Existen múltiples estrategias para la implementación del componente de vistas. Una de las estrategias más comunes es la de usar una página JSP como el componente de vistas. Otra estrategia muy utilizada es la denominada ServletViewStrategy, la cual utiliza un servlet como vista.

La encapsulación de la lógica de negocios en un ayudante en lugar de en una vista hace que las aplicaciones sean mas modulares y facilita la reutilización de componentes.

Múltiples clientes, tales como controladores y vistas, pueden apoyarse en el mismo ayudante para recuperar y adaptar un modelo de estado similar para realizar presentaciones de distintas formas.

Una señal que puede indicar la necesidad de usar de este patrón con el código existente, es que un scriplet sobrecargue o desordene una vista JSP. El objetivo principal de aplicar este patrón es realizar la partición de la lógica de negocios fuera de la vista. Aunque algo del código de la lógica de negocios puede ser más útil si se encuentra encapsulado dentro de ayudantes, pueden existir porciones de código que deben estar en un componente centralizado que los situé en frente de las vistas y los ayudantes (tales como chequeos de autenticación o servicios de registro de actividades “logging”).[ii]

Este patrón se enfoca en la recomendación de múltiples formas de particionar las responsabilidades en una aplicación.

LA ARQUITECTURA MODELO VISTA CONTROLADOR

Estructura del patrón “Helper View”

Una vista representa y muestra información al cliente. La información que es usada en un display es obtenida de un modelo. Los ayudantes dan soporte a las vistas por medio de la encapsulación y adaptación de un modelo para su uso en un display.

Un ayudante es responsable de contribuir a completar el procesamiento que realiza una vista o un Controlador. Así, los ayudantes tienen numerosas responsabilidades, incluyendo la recolección de datos requeridos por una vista y adaptarlos al modelo de datos usado por la vista. Los ayudantes pueden atender las solicitudes de datos de una vista con el simple hecho de proveerle a la misma acceso a los metadatos o formateando los datos como contenido Web.

Una vista puede trabajar con cualquier número de ayudantes, los cuales son implementados típicamente como JavaBeans y etiquetas personalizadas. Adicionalmente, un ayudante puede representar un objeto de tipo Command, Delegate o también un objeto de tipo XSLT, el cual es usado en combinación con una hoja de estilo para adaptar y convertir el modelo a una forma apropiada.

Mejoras en el particionamiento de aplicaciones

El uso de ayudantes proporciona una separación clara de las vistas con respecto a la lógica de procesamiento de la aplicación. Los ayudantes, en la forma de JavaBeans y Etiquetas Personalizadas, proveen un lugar para colocar la lógica de procesamiento fuera de las páginas JSP.

Separación de Roles

Separando la lógica de formateo de la lógica de negocios de la aplicación, reduce las dependencias que pueden llegar a tener los diferentes roles en los mismos recursos.

Reusabilidad

La lógica de negocios que se encuentra fuera de un JSP y dentro de JavaBeans o Etiquetas personalizadas puede ser reutilizada a lo largo de múltiples JSPs. El código no es duplicado en varios JSPs, lo cual hace que la aplicación sea más fácil de mantener y depurar. Adicionalmente, debido a que la lógica de negocios se ha

removido de las vistas, la misma lógica de negocios puede ser reutilizada, sin modificación potencial, y ser utilizada en otro tipo de interfaz totalmente diferente tal como Swing.

Patrón Composite View

Las páginas Web sofisticadas presentan contenidos de numerosas Fuentes de datos, usando varias sub-vistas que se encuentran contenidas dentro de una página. Adicionalmente, una variedad de individuos con diferentes habilidades contribuyen al desarrollo y mantenimiento de estas páginas web.

En lugar de proveer un mecanismo para combinar módulos o porciones atómicas de una vista para generar una composición completa, las páginas son construidas al incrustar código de formateo directamente dentro de cada vista.

La Modificación de la disposición de múltiples vistas es difícil y propensa a errores, debido a la duplicación de código.

Porciones atómicas del contenido de una vista pueden cambiar frecuentemente.

Múltiples vistas compuestas utilizan sub-vistas similares, tales como una tabla de inventarios. Estas porciones atómicas son decoradas con diferentes plantillas de texto, o aparecen en un lugar diferente dentro de la página.

La disposición de los elementos es más difícil de manejar y el código es mas difícil de mantener cuando las sub-vistas están incluidas directamente o duplicadas en diferentes vistas

La inclusión frecuente de nuevos elementos cambiando porciones de las plantillas de texto directamente a vistas puede afectar potencialmente la disponibilidad y administración del sistema.

Usar vistas compuestas que a su vez se encuentren compuestas de sub-vistas atómicas. Cada componente de la plantilla puede ser incluido dinámicamente en un todo y la apariencia de la página puede ser manejada independientemente del contenido.

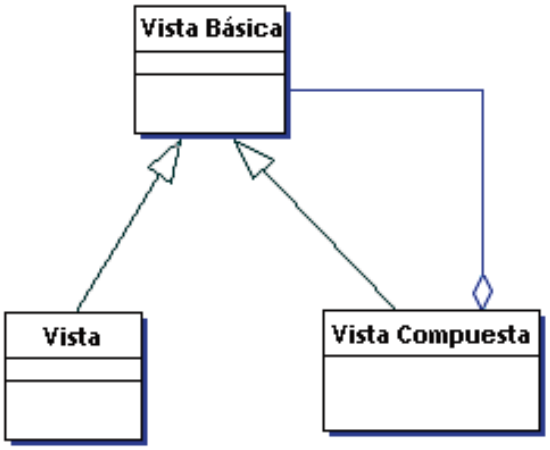
Esta solución facilita la creación de vistas compuestas basada en la inclusión y sustitución de módulos y fragmentos de plantillas dinámicamente. Esto promueve la reutilización de porciones atómicas de las vistas al facilitar el diseño modular. Las vistas compuestas pueden ser usadas para generar páginas que contienen componentes de visualización que pueden ser combinados en una gran variedad de formas, un ejemplo de esto puede ser, un portal web que incluye un gran número de sub-vistas independientes, tales como recepción de nuevas noticias, información del clima, e información de inventario todo en una página simple, para este caso la apariencia de la página es manejada y modificada independientemente del contenido de las sub-vistas.

LA ARQUITECTURA MODELO VISTA CONTROLADOR

Otro beneficio de este patrón es que los diseñadores Web pueden hacer prototipos de la apariencia de un site, colocando contenido estático en cada una de las regiones de las plantillas. A medida que el desarrollo del site avanza, el contenido puede ser modificado por sus elaboradores.

Este patrón posee desventajas, y una de las mas evidentes es la de la posible sobrecarga de ejecución producida por el incremento de flexibilidad que este patrón provee. Además, el uso de una distribución de elementos sofisticada puede generar problemas de desarrollo, debido al gran número de artefactos que se deben mantener.

Estructura del patrón patrón Composite View:



Mejora la Modularidad y la Reutilización

Este patrón mejora el diseño modular. Esto hace posible que se puedan reutilizar porciones atómicas de una plantilla, tal como una tabla de inventario, en numerosas vistas, adicionalmente estas vistas pueden ser rellenadas con información diferente. Este patrón permite que una tabla sea movida a su propio módulo y que sea usada solo cuando sea necesario. Este tipo de distribución dinámica de elementos reduce la duplicación innecesaria de código y mejora la facilidad de mantenimiento de la aplicación.

Flexibilidad mejorada

Una implementación sofisticada puede incluir fragmentos de una plantilla basándose en decisiones tomadas en tiempo de ejecución tales como el rol de un usuario o una política de seguridad.

LA ARQUITECTURA MODELO VISTA CONTROLADOR

Generar una página que contenga numerosas sub-vistas puede disminuir el rendimiento de una aplicación. La inclusión de sub-vistas en tiempo de ejecución puede convertirse en un retraso que se produce cada vez que la página es solicitada por el cliente, lo cual podría afectar drásticamente el desempeño de la aplicación en ambientes con Niveles de Servicio muy estrictos. Una alternativa es que la inclusión de la sub-vistas se realice en el momento en el que el motor JSP se encuentra procesando una página, aunque esto limita a que los cambios que se realicen sobre una sub-vista solo se muestren cuando esta sea nuevamente procesada.

Patrón Dispatcher View

Los Sistemas manejan solicitudes Web. Los procesos de la capa de presentación requieren la generación de una vista basada en una plantilla y un modelo dinámico.

Los cambios de la capa de Presentación ocurren a menudo y son difíciles de desarrollar y mantener, debido a la combinación de la lógica de negocio para el acceso a los datos y la lógica de formateo de la presentación, esto hace que el sistema sea menos flexible, menos reutilizable, y menos adaptable a los cambios.

Las porciones de lógica de negocio que son mezcladas con las vistas deben ser adaptadas al modelo intermedio para su visualización.

Mezclar la lógica de negocio con el procesamiento de las vistas también reduce la modularidad y disminuye la separación de roles a la hora de asignar tareas a los miembros de un equipo de desarrollo de software o de aplicaciones web.

Ventajas

- El contenido requiere que los datos de negocio sean extraídos dinámicamente.

- Encapsular la lógica de negocio en otros componentes distintos de las Vistas.

Las Decisiones que afectan la recuperación de contenido y la adaptación de éstos a un modelo para ser mostrados en las vistas son a menudo retrasadas hasta el momento en el que las vistas son procesadas.

Los servicios del sistema y la lógica de manejo de vistas son básicos y limitados en complejidad.

El número de vistas que puede ser mapeado a una solicitud particular es reducido.

Combinar un Despachador con Vistas y Ayudantes para manejar las solicitudes de los clientes y preparar una presentación dinámica como respuesta. Un despachador es responsable por el manejo de vistas y la navegación, y puede ser o no encapsulado dentro de un controlador, o puede ser un componente independiente que trabaje de forma sincronizada con los componentes internos.

LA ARQUITECTURA MODELO VISTA CONTROLADOR

Este patrón y el Service to Worker poseen una estructura similar, aunque cada patrón sugiere una división diferente de las labores que debe realizar cada componente. El Controlador y el Despachador tienen responsabilidades limitadas (como se mencionó en el patrón anterior), debido a que el procesamiento principal y el manejo de vistas es muy básico. Mas aún, si el control centralizado de los recursos subyacentes es considerado innecesario, entonces el Controlador es removido y el Despachador puede ser movido a una Vista.

Los patrones “Service to Worker” y “Dispatcher View” representan una combinación común de otros patrones, por lo cual cada uno garantiza a su propia manera una forma mas eficiente de comunicación entre desarrolladores.

En el patrón “Dispatcher View”, el Despachador posee un rol limitado en cuanto al manejo de Vistas. En el patrón “Service to Worker”, el Despachador posee un rol mucho mas relevante en cuanto al manejo de vistas.

Un rol limitado para el Despachador ocurre cuando no se utilizan recursos externos para seleccionar una vista. La información encapsulada en la solicitud es suficiente para determinar las Vistas a utilizar para resolver una petición. Por ejemplo en este caso:

http://some.server.com/servlet/Controller?next=login.jsp

La responsabilidad del Despachador es la de proporcionar acceso a la vista ‘login.jsp‘

Un ejemplo del Despachador con rol moderado es el caso en el cual el cliente envía una petición directamente a un Controlador con un parámetro de una consulta que describe una acción que debe ser completada:

http://some.server.com/servlet/Controller?action=login

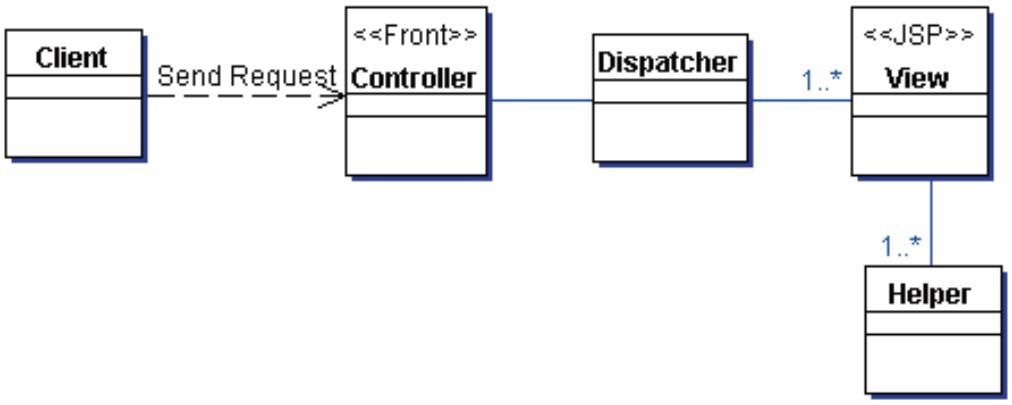
La responsabilidad del Despachador en este caso es la de traducir el nombre lógico ‘login‘ en el nombre de un recurso de una vista apropiada tal como lo es ´login.jsp´ y así proveer el acceso a dicha vista. Para llevar a cabo su traducción, el Despachador puede acceder a recursos como un archivo XML de configuración que especifique la vista que debe mostrarse.

Por otro lado, en el patrón “Service to Worker”, el Despachador debe ser más sofisticado, ya que este debe invocar las funciones de negocio necesarias para determinar las vistas que se van a mostrar.

La estructura compartida de estos dos patrones que se mencionó anteriormente esta conformada por un Controlador que trabaja con un Despachador, Vistas y Ayudantes.

LA ARQUITECTURA MODELO VISTA CONTROLADOR

Estructura del patrón Dispatcher View:



Los componentes del diagrama de clases se especifican a continuación:

Controlador

El Controlador es el punto de contacto inicial para el manejo de una solicitud. Este trabaja con un Despachador para completar el manejo de vistas y la navegación.

Despachador

Un Despachador es responsable del manejo de Vistas y navegación, ya que este se encarga de seleccionar la próxima vista a ser presentada al Usuario, además de proveer un mecanismo para el control vectorizado de un recurso.

Vista

Una vista representa y visualiza información para los clientes. La información que es usada en una vista es tomada de un modelo.

Ayudante

Un ayudante es responsable de contribuir a completar el procesamiento que realiza una vista o un Controlador. Así, los ayudantes tienen numerosas responsabilidades, incluyendo la recolección de datos requeridos por una vista y adaptarlos al modelo de datos usado por la vista. Los ayudantes pueden atender las solicitudes de datos de una vista con el simple hecho de proveerle a la misma acceso a los metadatos o formateando los datos como contenido Web.

LA ARQUITECTURA MODELO VISTA CONTROLADOR

Una vista puede trabajar con cualquier número de ayudantes, los cuales son implementados típicamente como JavaBeans y etiquetas personalizadas. Adicionalmente, un ayudante puede representar un objeto de tipo Command, Delegate o también un objeto de tipo XSLT, el cual es usado en combinación con una hoja de estilo para adaptar y convertir el modelo a una forma apropiada.

Aunque las responsabilidades del Controlador están limitadas a los servicios que posee el sistema, tales como autenticación y autorización, a menudo resulta beneficioso centralizar estos aspectos del sistema. A diferencia del patrón “Service to Worker”, el Despachador no realiza invocaciones a métodos de negocio para llevar a cabo su procesamiento y manejo de las vistas.

Las funcionalidades del Despachador pueden ser encapsuladas dentro de su propio componente. Al mismo tiempo, cuando las responsabilidades del Despachador son limitadas, pueden ser almacenadas en otro componente

De hecho, las funcionalidades del Despachador podrían ser incluso realizadas por el contenedor, en el caso de que no se necesite lógica extra a nivel de aplicación. Un ejemplo de esto, es una vista llamada main.jsp la cual tiene como alias el nombre first. El contenedor puede procesar esta petición, traducir el alias al nombre fisico del recurso y entregar directamente dicho recurso:

http://some.server.com/first

De esta manera, el patrón “Dispatcher View” describe una serie de escenarios relacionados, moviéndose desde un escenario que es muy similar estructuralmente al patrón “Service to Worker” , hasta un escenario que es similar al patrón “View Helper”

Control Centralizado

El control centralizado provee un lugar central para manejar los servicios del sistema a los largo de múltiples solicitudes. Aunque decisiones tales como seleccionar que JSP debe mostrarse se encuentran incluidas dentro de la petición, el control centralizado permite al controlador mejorar el manejo de los servicios de seguridad y despacho.

Reusabilidad

La lógica de negocios que se encuentra fuera de un JSP y dentro de JavaBeans o Etiquetas personalizadas puede ser reutilizada a lo largo de múltiples JSPs. El código no es duplicado en varios JSPs, lo cual hace que la aplicación sea más fácil de mantener y depurar. Adicionalmente, debido a que la lógica de negocios se ha removido de las vistas, la misma lógica de negocios puede ser reutilizada, sin modificación potencial, y ser utilizada en otro tipo de interfaz totalmente diferente tal como Swing.

Seguimiento de Usuarios y Registro de Sucesos

Permite el seguimiento de un usuario particular y el registro de sucesos relacionados con una actividad.

Validación y Manejo de Errores

El controlador es capaz de manejar la validación de datos y el Manejo de Errores, ya que estas operaciones deben ser realizadas por solicitud.

Mejoras en el particionamiento de aplicaciones

El uso de ayudantes proporciona una separación clara de las vistas con respecto a la lógica de procesamiento de la aplicación. Los ayudantes, en la forma de JavaBeans y Etiquetas Personalizadas, proveen un lugar para colocar la lógica de procesamiento fuera de las páginas JSP.

Patrones de la Capa de la Lógica de Negocios

Patrón Value Object

La aplicación cliente necesita intercambiar datos con algún Enterprise Java Bean (EJB).

Los aplicaciones J2EE implementan componentes de negocios del lado de servidor como beans de sesión (session beans) y beans de entidad (entity beans). Los beans de sesión representan los servicio de la lógica de negocios y permiten mantener una relación uno a uno con el cliente. Los beans de entidad, por su parte, son multiusuarios y representan la data persistente.

Un bean de sesión provee métodos de servicio de grano grueso cuando es implementado por medio del patrón Session Facade.

Algunos de éstos métodos pueden retornar datos al cliente que realiza alguna invocación a los mismos.

Un bean de entidad implementa los componentes de negocio persistente. Permite exponer los valores de sus atributos mediante los métodos de acceso (métodos get) para cada atributo. Cuando un cliente necesita obtener múltiples valores de un entity bean, debe invocar a lo métodos get respectivos hasta obtener la data correspondiente a cada atributo que requiera.

Ventajas

Las aplicaciones J2EE implementan componentes de negocio como beans de entidad. Todos los accesos hacia un bean de entidad se realizan por medio de las interfaces remotas que provee el bean. Cada llamada a un enterprise bean es potencialmente una llamada a un método remoto, lo cual recae en una sobrecarga para la red.

LA ARQUITECTURA MODELO VISTA CONTROLADOR

Por lo general, las aplicaciones utilizan con mayor frecuencia de operaciones de lectura que de actualización. La data se transfiere desde la lógica de negocio hacia la capa presentación, despliegue y posteriormente al cliente.

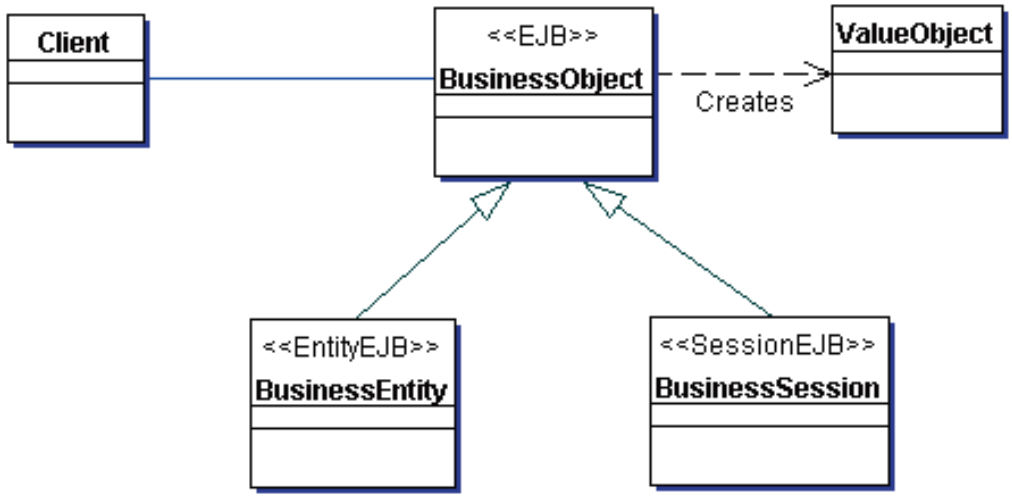
El número de llamadas que realiza el cliente hacia los EJB tiene un gran impacto sobre el rendimiento de la red, debido a la sobrecarga que se genera por cada petición.

Se puede utilizar un Value Object para encapsular la data del negocio. La llamada a un método sencillo puede ser utilizado para enviar y recibir el Value Object. Cuando el cliente le solicita al enterprise bean los datos de negocio, el bean puede construir el Value Object, rellenarlo con los valores de los atributos, y retornarlo al cliente.

Los clientes por lo general requieren más de un valor de un enterprise bean. Para reducir el numero de llamadas remotas y evitar por consiguiente la sobrecarga asociada, resulta más conveniente utilizar el Value Object para transportar la data desde el enterprise bean hacia el cliente.

Cuando el enterprise bean utiliza un value object, el cliente realiza una sola invocación al bean obteniendo como resultado el value object en lugar de realizar múltiples llamadas para obtener cada uno de los atributos. Luego el bean instancia un value object y copia los valores de sus atributos en la instancia del objeto que acaba de crear. Finalmente retorna el value object al cliente.

Estructura del patrón Value Object:



Este patrón simplifica al Entity Bean y a su interfaz remota

El entity bean provee un método getData() para obtener el value object que contiene los valores de los atributos deseados. Esto elimina la necesidad de tener multiples metodos get, así como su respectiva implementación en el bean y su definición en la interfaz remota. Similarmente, si el entity bean provee un método SetData(), se podrían eliminar también los métodos set del bean en la implementación y de igual forma en la interfaz remota.

Reduce el tráfico de la red.

Si se adopta la estrategia del patrón value object los clientes pueden realizar modificaciones en la copia local, es decir, el objeto value object. Una vez que se realizan dichas modificaciones, el cliente puede invocar al método setData() y actualizar los valores en el objeto entidad. Aunque esto puede traer problemas ya que otros clientes pueden haber realizado el mismo procedimiento, ya que no es posible determinar en primera instancia si ya se han modificado otros valores por parte de los clientes.

Sincronización y control de versiones

Cuando se reciben múltiples actualizaciones de dos o más clientes de manera simultánea el bean de entidad debe estar en capacidad de manejar dicha situación. El control de versiones es una manera de prevenir los conflictos que se puedan presentar. bean de entidad puede incluir como uno de sus atributos el número de la versión y la ultima estampa de tiempo (timestamp) relativa a su modificación; ya que así, dicho atributo puede ser utilizado a la hora de resolver conflictos de versiones.

Es posible reducir o eliminar la duplicación de código entre la entidad y su value object correspondiente. Sin embargo, al realizar la implementación de algunas estrategias para este patrón pueden incrementar la complejidad a la hora de realizar la implementación del diseño.

En lugar de realizar múltiples llamadas para obtener los valores de los atributos, ésta solución provee un método único para realizar la llamada. Aunque muchas veces la llamada a dicho método puede retornar gran cantidad de datos. Por lo cual se debe considerar la relación entre el número de llamadas que se deben realizar al bean versus la cantidad de datos obtenidos por llamada.

Patrón Business Delegate

El sistema expone sus servicios de negocio por medio de un API a sus clientes, a través de una red.

Los componentes de la capa de presentación interactúan directamente con los servicios de la lógica de negocios. Esta interacción expone los detalles de implementación del API de la capa de lógica de negocios hacia la capa de presentación. Como resultado de esto, los componentes de la capa de presentación son vulnerables a los cambios de implementación que se realicen en la capa de lógica de negocios.

Adicionalmente, podría existir un detrimento serio en el rendimiento de la red, debido a que los componentes de la capa de presentación que utilizan el API de la capa de lógica de negocios, podrían realizar múltiples invocaciones a través de la red. Esto ocurre cuando los componentes de la capa de presentación no poseen mecanismos de caché o servicios agregados.

Finalmente, la exposición directa del API a los clientes, el mismo, se va obligado a lidiar con la ínter conectividad a la cual esta asociada la tecnología de los EJB.

Ventajas

Los clientes de la capa de presentación necesitan acceder a los servicios de negocio.

Los dispositivos clientes (incluyendo clientes ricos) necesitan acceder a los servicios de la lógica de negocios.

El API de los servicios de la lógica de negocios puede cambiar al igual que los requerimientos del negocio.

La meta debe ser minimizar el acoplamiento entre la capa de presentación y el API de los servicios de la lógica de negocios. Así se ocultan detalles de implementación de los servicios, como la búsqueda y el acceso.

Sería deseable implementar mecanismos de caché para la información de los servicios de la lógica de negocio.

Sería deseable reducir el tráfico de red entre el cliente y los servicios de la lógica de negocios.

El Business Delegate podría ser visto como una capa innecesaria entre el cliente y los servicios. Por otra parte introduce complejidad y decrementa la flexibilidad.

Se puede utilizar el patrón Business Delegate para reducir el acoplamiento entre la capa de presentación y los servicios de la lógica de negocios. El Business Delegate esconde los detalles de implementación de la lógica de negocios, como por ejemplo búsqueda (lookup) y los detalles de acceso a la arquitectura de los EJB.

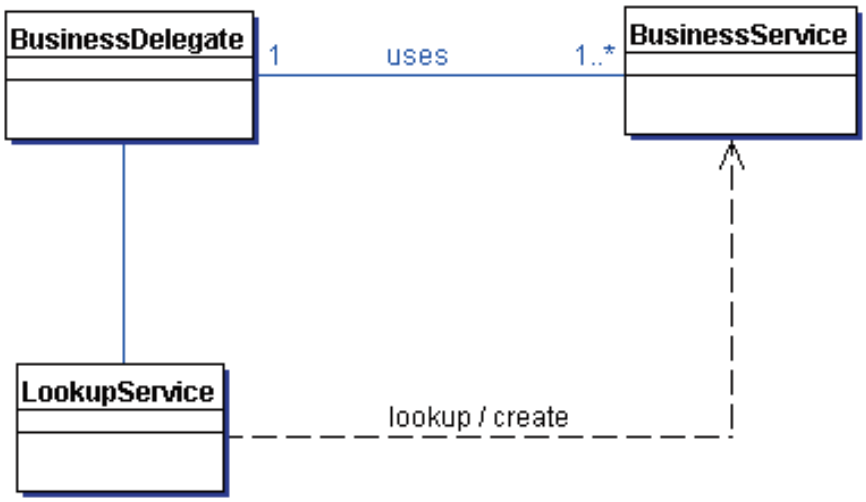
El Business Delegate se podría pensar como una abstracción de la lógica de negocios de lado del cliente. El mismo, también puede manejar las excepciones que se generen desde la lógica de negocios como por ejemplo las excepciones de los EJB, así como también de alguno de los componentes de JMS, entre otras. Con lo cual es posible interceptar dichas excepciones y generar excepciones a nivel de aplicación que busquen explicar con mayor exactitud la falla o el error suscitado en un momento determinado.

Otro beneficio es que dicho patrón puede manejar una caché de resultados, con lo cual aumenta significativamente el rendimiento, debido a que evita el consumo de recursos innecesarios debido a múltiples peticiones viajando por la red.

El Business Delegate utiliza un componente llamado Lookup Service. El Lookup Service es el responsable de ocultar los detalles de implementación del código de búsqueda dentro de la lógica de negocios. El Lookup Service pueden ser escrito como parte del Business Delegate, aunque es recomendable utilizarlo de forma autónoma.

Finalmente, hay que destacar que este patrón puede reducir la complejidad entre otras capas y no sólo entre la capa de presentación y la de lógica de negocios.

Estructura del patrón Business Delegate:



Reduce el acoplamiento, mejora la gestión:

Reduce el acoplamiento entre la capa de presentación y la capa de lógica de negocios, ocultando todos los detalles de implementación. También facilita la gestión a la hora de realizar cambios, ya que todo se encuentra centralizado en un solo lugar, el BusinessDelegate.

El BusinessDelegate es el responsable de traducir cualquier excepción de la red u otras que se puedan presentar en excepciones del negocio, abstrayendo al cliente de los detalles específicos de la implementación.
El BusinessDelegate puede proveer mecanismos de caché (con lo cual mejora el rendimiento) a la capa de presentación para los servicios que realicen peticiones frecuentes.

Patrones de la Capa de Integración

Patrón Data Access Object

El acceso a los datos puede variar dependiendo del origen de los datos(datasource). El acceso a almacenes persistentes de datos, tales como una base de datos, varía grandemente dependiendo del tipo de almacenamiento (bases de datos relaciones, orientadas a objetos, etc) y la implementación del fabricante.

Muchas aplicaciones J2EE pueden requerir el uso de data persistente en algún momento. Para muchas aplicaciones, los repositorios persistentes utilizan mecanismos diferentes y existen grandes diferencias entre las APIs usadas por los distintos repositorios. Otras aplicaciones pueden necesitar acceder a los datos que residen en un mainframe, un servicio B2B o un repositorio LDAP.

Típicamente, las aplicaciones usan componentes distribuidos compartidos para representar la data persistente. Una aplicación es analizada para determinar si debe utilizar persistencia manejada por el bean(BMP) o por el contenedor de beans(CMP).

Las aplicaciones pueden usar el API JDBC para acceder a los datos residentes en una base de datos relacional. El API JDBC permite un acceso estándar y manipulación de datos en un almacenamiento persistente. JDBC permite a las aplicaciones J2EE usar sentencias SQL, las cuales son estándar para el manejo de tablas en bases de datos relacionales. Sin embargo, incluso dentro de un ambiente relacional, la sintaxis y el formato de las sentencias SQL puede variar dependiendo del manejador de base de datos que se utilice.

Existen variaciones mucho mayores que involucran la combinación de varios tipos de almacenamiento. Los mecanismos de acceso, las APIs soportadas, y las características pueden variar drásticamente cuando se trata de comparar un tipo de almacenamiento persistente como una base de datos relacional con otro tipo de almacenamiento persistente. Las aplicaciones que necesitan acceder a datos que sen encuentran en sistemas legacy por lo general necesitan emplear APIs propietarias. Los datasources diferentes ofrecen dificultades a la aplicación y puede crear una dependencia muy fuerte entre el código de la aplicación y el código de integración. Cuando los componentes de negocio necesitan acceder a un data source, pueden usar el API adecuada para lograr conectarse a un datasource y así manipular los datos que se encuentran en él. Pero, incluyendo la conectividad y el código de acceso a los datos dentro de estos componentes puede generar una dependencia entre los componentes y la implementación usada por el datasource. Esa dependencia de código hace difícil y tedioso migrar la aplicación de un tipo de datasource a otro, ya que cuando el datasource cambia los componentes deben ser cambiados también.

Ventajas

Componentes tales como Entity beans de Tipo BMP, Beans de Sesión y Servlets necesitan obtener y almacenar datos de repositorios persistentes y otros Data source como sistemas legacy, B2B, LDAP, etc.

Las APIs usadas para acceder a los repositorios persistentes varían de acuerdo al fabricante del producto. Algunos data source pueden tener APIs no estándares o propietarias. Estas APIs y sus capacidades también varían según el tipo de almacenamiento (Bases de datos relacionales, documentos XML, archivos planos. Etc). Existe una carencia de uniformidad en las APIs para el manejo de los requerimientos de acceso a datos que pueda tener el sistema.

Los componentes que acceden a sistemas de tipo legacy para obtener y almacenar datos por lo general utilizan APIs propietarias.

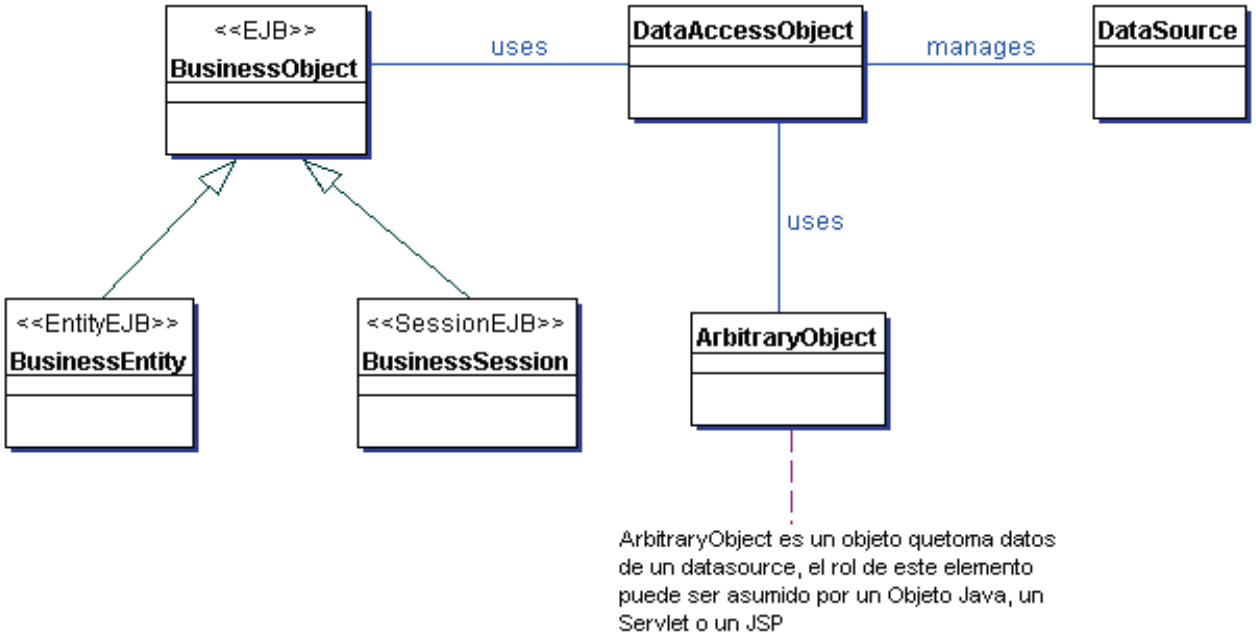
La portabilidad de los componentes se ve afectada directamente cuando mecanismos de acceso específicos y APIs son incluidos en los componentes.

Los componentes necesitan deben tener un acceso a los datos transparentes totalmente independiente de las implementaciones que posean los Data source, para así permitir que se realice más fácilmente la migración entre diferentes productos de diferentes fabricantes.

Usar un Objeto de Acceso a Datos para abstraer y encapsular todos los accesos realizados al origen de los datos. Los Objetos de Acceso a Datos manejan la conexión con el origen de datos para obtener y almacenar datos.

Los Objetos de Acceso a Datos (DAO) son el elemento principal de este patrón. Los DAO implementan los mecanismos de acceso requeridos para trabajar con el origen de los datos. Los orígenes de datos pueden ser un repositorio persistente tal como un Sistema Manejador de Bases de Datos Remoto, un servicio externo (por ejemplo una transacción entre dos empresas), una base de datos LDAP o una función de negocio que puede ser accedida vía CORBA ó IIOP. Los componentes de negocios que dependen de los DAO usan interfaces simples para la interacción con los clientes. Debido a que la interfaz mostrada por los DAO a los clientes no cambia cuando la implementación subyacente del origen de datos cambia, este patrón permite a los DAO adaptarse a diferentes esquemas de almacenamiento sin afectar a sus clientes o a los objetos de lógica de negocio. Los DAO actúan como un adaptador entre los componentes y los data source.

Estructura del patrón Data Access Object.



BusinessObject

La clase BusinessObject representa a los clientes y es un objeto que requiere acceso a los data source para obtener y almacenar datos. Un objeto de tipo Bussiness Object puede ser implementado como un bean de sesión, un entity bean, o algún otro objeto Java, incluyendo un servlet o un helper bean para acceder a los datos.

DataAccessObject (DAO)

Los DAO son el objeto principal de este patrón, estos proveen las características de encapsulación y delegación al Business Object. Los DAO se abstraen de la implementación subyacente para acceso a los datos permitiendo así que los objetos de negocio tengan un acceso transparente al origen de datos. Los objetos de Negocio delegan a los DAO las operaciones de almacenamiento y obtención de los datos.

DataSource

Este clase representa las implementación usada por un data source. Un data source puede ser una base de datos relacional, una base de datos orientada a objetos, un repositorio XML o un sistema de archivos planos entre otros.

Este patrón facilita la Transparencia

Los objetos de negocio pueden usar datos sin conocer los detalles específicos de la implementación del origen de los datos. El acceso es transparente debido a que los detalles de implementación se encuentran encapsulados dentro del Objeto de Acceso a Datos (DAO)

Facilita la migración a diferentes implementaciones de implementaciones de almacenes persistentes

Una capa de Objetos de Acceso a Datos facilita a las aplicaciones el hecho de migrar a diferentes bases de datos. Los objetos de negocio no tienen conocimiento de la implementación subyacente de los datos. Así, al momento de realizar la migración solo se consideran los cambios que deben realizarse a la capa de Acceso a Datos.

Reduce la complejidad de código de los Objetos de Negocio

Debido a que todas las complejidades relacionadas con el acceso a los datos es realizado por los Data Objects, todo el código relacionado con la implementación (por ejemplo sentencias SQL) es colocado fuera de los Objetos de negocio, mejorando con ello la legibilidad del código e incrementando la productividad de desarrollo.

Centraliza todos los accesos a datos en una capa independiente

Gracias a los Data Objects, la capa de acceso a datos puede ser vista como una capa que puede aislar el resto de la aplicación de la implementación de los datos, incrementando con ello la manejabilidad de la aplicación.

Poca utilidad con Entity Beans con Persistencia Manejada por el Contenedor (CMP)

Debido a que el contenedor de beans provee todos los servicios necesarios para el manejo de datos persistente. Las Aplicaciones que usen beans de tipo CMP no necesitan Objetos de Acceso a Datos, ya que el servidor de Aplicaciones donde se encuentre la aplicación provee de forma transparente esta funcionalidad.

Service Activator

Los Enterprise Java Beans (EJB) y otros servicios de negocios necesitan una forma de ser activados de forma asíncrona sin intervención del usuario.

Los EJB implementados en especificaciones anteriores a la 2.0 no soportaban invocación asíncrona. La especificación 2.0 introduce la integración con el servicio de mensajería de Java (Java Message Service) por medio de los Message Driven Bean. Cuando un cliente necesita acceso a los servicios ofrecidos por un enterprise bean, se realiza una búsqueda de la interfaz home del bean, posteriormente se invoca a los métodos create/find/remove de esa interfaz. Todas las interacciones entre el cliente y los componentes EJB se realizan de manera remota y sincrónica. La especificación de los EJB le permite al contenedor remitir un enterprise bean a un repositorio secundario. Como resultado, el Contenedor EJB no posee mecanismos para proveer un servicio donde un bean se encuentre siempre en estado activo. Debido a que el cliente debe interactuar con el bean a través de una interfaz remota,(incluso si el bean se encuentra siempre en el contenedor), por lo cual siempre seguirá interactuando con el bean de modo sincrónico.

Si una aplicación necesita procesamiento sincrónico para componentes de negocio del lado del servidor, entonces la elección apropiada es el uso de Enterprise Java Beans. Algunas aplicaciones cliente pueden requerir algo de procesamiento asincrónico para los objetos de negocio del lado del Servidor debido a que los clientes no necesitan esperar a que todo el procesamiento de datos se realice por completo. En tales casos, la funcionalidad de los beans provista en especificaciones anteriores a la 2.0 no cubre los requerimientos necesarios para brindar una solución completa.

En la especificación 2.0 de los EJB, se introduce un Nuevo tipo de bean denominado Message-Driven Bean el cual es un tipo especial de bean sin estado. Sin embargo, esta nueva especificación no ofrece invocación asincrónica para otros tipos de Enterprise Java Beans. En general, cualquier servicio de negocio que provea únicamente procesamiento sincrónico puede tener limitaciones al trata de brindar facilidades de procesamiento asíncrono.

Ventajas

Los Enterprise Beans son mostrados a sus clientes por medio de sus interfaces remotas, las cuales solo permite acceso sincrónico.

El contenedor controla a los enterprise beans, permitiendo interacciones solo por medio de referencias remotas. El contenedor EJB no permite acceso directo a la implementación del bean o a sus métodos. Así, la implementación de un componente “escucha” de mensajes en un Enterprise Bean no esta permitido ya que esto violaria las especificaciones de EJB anteriores a la 2.0, ya que se estaria permitiendo un acceso directo a la implementación del bean. La especificación de EJB 2.0 plantea un Nuevo tipo de bean denominado Message-Driven Bean que permite el desarrollo de un “manejador de mensajes”.

Una aplicación necesita proveer un framework de publicación/suscripción en el cual los clientes puedan publicar peticiones destinadas a los EJB, los cuales pueden procesar las peticiones de un modo asíncrono.

LA ARQUITECTURA MODELO VISTA CONTROLADOR

Los clientes necesitan capacidades de procesamiento asíncrono de los EJB y otros componentes de negocio síncronos, con lo cual el cliente puede enviar una solicitud para su procesamiento sin necesidad de esperar los resultados.

Los clientes pueden usar las interfaces middleware “orientadas a mensajes” ofrecidas por el Servicio de Mensajes de Java (JMS). Estas interfaces no están integradas en los servidores EJB basados en especificaciones EJB anteriores a la 2.0.

Proveer características similares a las de los procesos “demonio” permitiendo con esto que un bean pueda permanecer inactivo hasta la ocurrencia de un evento o la llegada de un Nuevo mensaje.

Los Message Beans basados en la especificación 2.0 son beans de sesión sin estado, por lo cual, no es posible realizar la invocación asincrónica de cualquier otro tipo de beans.

Usar un Service Activator para recibir peticiones y mensajes asincrónicos del cliente. Al momento de la recepción de un mensaje, el Service Activator localiza e invoca los métodos de negocios necesarios para completar las solicitudes de manera asincrónica.

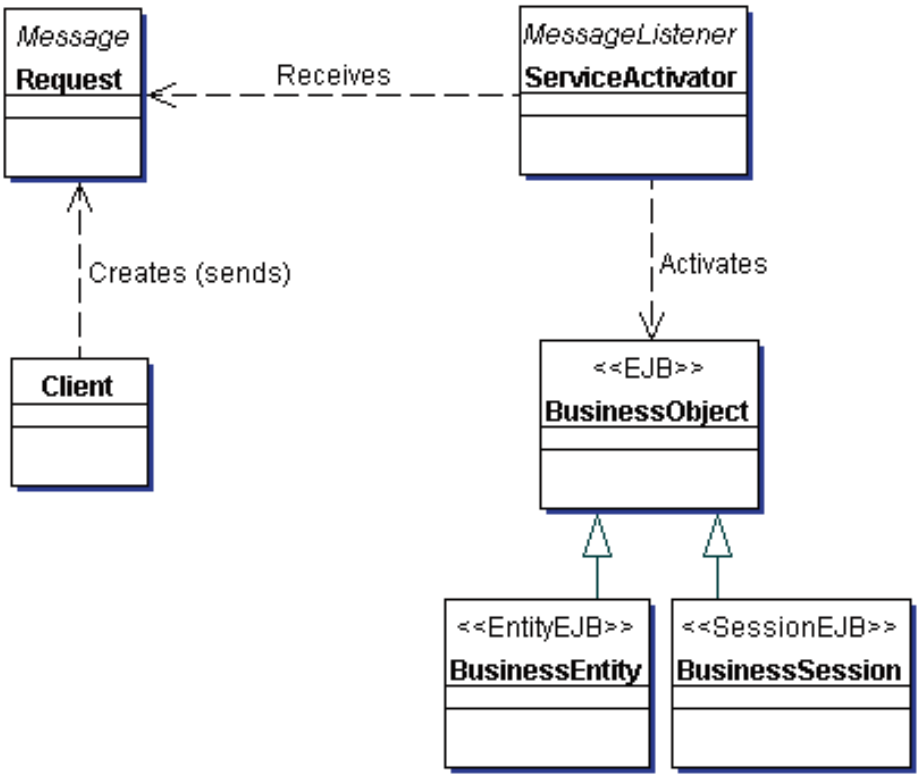
El ServiceActivator es un Manejador de Mensajes y delegador que requiere implementar la interfaz Message Listener para procesar todos los mensajes entrantes. Los Clientes actúan como generadores de mensajes, generando eventos basados en su actividad.

Cualquier cliente que necesite invocar asincrónicamente un servicio, como por ejemplo un enterprise bean, puede crear y enviar un mensaje al Service Activator. El Service Activator recibe los mensajes y los analiza para interpretar la solicitud del cliente. Una vez que se ha identificado la solicitud del cliente, se localizan componentes de negocio necesario para satisfacer los requerimientos del cliente.

El Service Activator puede enviar un reconocimiento al cliente después de completar exitosamente el procesamiento de una solicitud.

El Service Activator puede utilizar los servicios de un Localizador de Servicios para localizar un componente de negocios.

Diagrama de clases del patrón Service Activator



Cliente (Client)

El cliente requiere facilidades de procesamiento asincrónico proveniente de los objetos de negocio provenientes de un flujo de trabajo. El cliente puede ser cualquier tipo de aplicación que tenga la capacidad de crear y enviar mensajes JMS. El cliente puede ser también un enterprise bean que necesite invocar los métodos de otro bean enterprise de manera asíncrona.

Peticione s(Request)

Las peticiones son los mensajes creados por los clientes y enviados al Service Activator. De acuerdo con la especificación de JMS, la petición es un objeto que implementa la interfaz javax.jms.Message. El API de JMS muchos tipos de mensaje tales como: TextMessage,ObjectMessage,etc, que pueden ser usados para generar peticiones.

ServiceActivator

La clase ServiceActivator es la clase principal de este patrón. Esta implementa la interfaz javax.jms.MessageListener, la cual es definida por la especificación de JMS. La clase ServiceActivator posee un método denominado onMessage() el cual es invocado cuando llega un Nuevo mensaje. Posteriormente la clase analiza el mensaje para determinar que debe hacerse para procesar la solicitud del usuario.

BusinessObject

La clase BusinessObject es el objeto al cual el cliente necesita acceder de manera asincrona. El Objeto de Negocio es un rol que puede ser perfectamente desempeñado por un Sesión Bean o un Entity Bean, aunque tambien puede ocurrir que el Objeto de Negocio sea representado por un servicio externo al cual el cliente desea acceder.

Consecuencias

Integración del Servicio de Mensajes de Java en implementaciones anteriores a la especificación EJB 2.0

Antes de la especificación 2.0 de los EJB, no existía ningún tipo de integración entre los EJB y los componentes JMS. Este patrón provee un medio para integrar JMS en una aplicación que utilice EJBs, añadiendo además capacidades de procesamiento asíncrono. La especificación 2.0 define un nuevo tipo de Bean de Sesión, denominado Message Bean, el cual permite realizar la implementación de JMS con los EJB. Los Message Beans son capaces de implementar la interfaz Message Listener y recibir mensajes de forma asíncrona. En este caso, el Servidor de Aplicaciones tomaría el rol de Service Activator.

Procesamiento Asincrono para cualquier Enterprise Bean

Usando el patrón Service Activator es possible proveer invocación asincrona en todos los tipos de beans. Con ello, las aplicaciones no deben esperar por los resultados de todas sus peticiones para continuar con sus funciones habituales.

Estandarización de Procesos

El Activador de Servicios (service Activator) puede ser ejecutado como un proceso común. Sin embargo, en una aplicación critica, este debe ser monitoreado para asegurar disponibilidad. El manejo adicional de este proceso podría generar algo de sobrecarga al sistema.