Patrones de Diseño para aplicaciones J2EE

Tabla de contenidos

Introducción a los Patrones de Diseño

El propósito de este manual es documentar la experiencia en el diseño de software empresarial con la plataforma J2EE en forma de patrones de diseño.

El arquitecto californiano Christopher Alexander describía el concepto de patrón (aplicado a la arquitectura civil) como sigue

"Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema, de tal modo que se pueda aplicar esta solución un millón de veces, sin hacer lo mismo dos veces".

Esta definición es completamente válida para los patrones de diseño orientados a objetos, sólo que nuestras soluciones vendrán dadas en términos de objetos e interfaces, en vez de paredes y puertas.

Los patrones tienen diferentes ámbitos de aplicación dentro de la ingeniería del software. Existen patrones de análisis, patrones de arquitectura, patrones organizacionales, etc. La diferencia entre estos patrones radica sustancialmente en el grado de abstracción y de la etapa de desarrollo en la cual se aplican.

Un patrón de diseño es una solución a un problema de diseño no trivial que es efectiva (ya se resolvió el problema satisfactoriamente en ocasiones anteriores) y reusable (se puede aplicar a diferentes problemas de diseño en distintas circunstancias).

La diferencia entre un diseñador de software orientado a objetos experto y uno que no lo es radica en que el primero será capaz de encontrar soluciones más eficientes, flexibles y reutilizables basándose en la experiencia. Y es justamente esa experiencia la que documentarán los patrones, convirtiéndose por tanto en potentes herramientas para la formación intelectual de nuevos diseñadores.

Un patrón de diseño nomina, abstrae e identifica los aspectos clave de una estructura de diseño común: identifica las clases e instancias, sus roles y colaboraciones, así como la distribución de responsabilidades.

Otro aspecto importante de los patrones es que facilitan la comunicación entre diseñadores, pues establecen un marco de referencia (terminología, justificación).

A diferencia de lo que muchos creen, los patrones de diseño no tienen como objetivo ahorrarnos trabajo si por ello entiende escribir menos código o utilizar la estrategia de copiar y pegar. Los patrones de diseño nos servirán para crear mejores soluciones y eso a veces puede significar la codificación de artefactos (clases, interfaces, ...) que en principio no contemplabamos.

Un patrón de diseño no tiene un reflejo directo en el código de nuestra aplicación. Un patrón no es un algoritmo, es algo más. La creatividad será necesaria cuando utilizemos patrones ya que deberemos darles forma en nuestro contexto.

Los algoritmos y estructuras de datos son más concretos que los patrones y se suelen encargar de cuestiones referentes a la complejidad computacional y no a cuestiones subyacentes más complejas (reusabilidad, mantenibilidad, adaptabilidad, ...) que son las que abordan los patrones. Un desarrollador de software debe conocer tanto los algoritmos y estructuras de datos como los patrones de diseño.

Cómo resuelven los patrones los problemas de diseño

En la programación orientada a objetos resulta complicado descomponer el sistema en objetos (encapsulación, granularidad, dependencias, flexibilidad, reusabilidad, etc.), los patrones de diseño nos permitirán identificar a los objetos apropiados. A veces será casi

imprescindible su uso para identificar abstracciones nada obvias que son descubiertas en etapas tardías del desarrollo, cuando se trata de hacer al diseño más flexible y reutilizable.

También abordarán la especificación de las interfaces, identificando los elementos claves en ellas y las relaciones existentes entre distintas interfaces. De igual modo nos facilitará la especificación de la implementaciones.

De forma casi automática, nos ayudan a reutilizar código, facilitando la decisión entre "herencia o composición" (favorece la composición sobre la herencia y hace uso de la delegación).

Y, quizás uno de las características más importantes, nos permiten hacer un diseño preparado para el cambio que por naturaleza existe en los entornos empresariales.

Mecanismos de descripción de los patrones de diseño

Para describir un patrón de diseño, además de las notaciones gráficas (diagramas UML) necesitamos documentación textual. Para reutilizar el diseño, debemos hacer constar las decisiones, alternativas y ventajas e inconvenientes que dieron lugar a él. También son importantes los ejemplos concretos porque nos ayudan a ver el diseño en acción.

En este manual hemos descrito los patrones de diseño J2EE en un formato consistente. Cada patrón se divide en secciones de acuerdo a una plantilla. Nuestra plantilla no es la única posible, de hecho encontrará tantas plantillas distintas como autores de bibliografía sobre patones.

Frameworks

Un framework es un conjunto de clases cooperantes que constituyen un diseño reutilizable para una clase específica de software.

En este manual encontrará continuas referencias al framework Struts (http://struts.apache.org/) del proyecto Apache. Struts es un framework orientado a la construcción de aplicaciones Web Java basadas en la arquitectura "Model 2", una variación del clásico paradigma de diseño Model-View-Controller (MVC). Como irá descubriendo, Struts utiliza en su diseño muchos de los patrones aquí explicados.

Los frameworks están ganando terreno y adquiriendo importancia ya que son el modo en el que los sistemas OO consiguen mayor reutilización.

Patrones VS Frameworks

- ✓ Los patrones de diseño son más abstractos que los frameworks: Los frameworks se plasman en código pero sólo los ejemplos de patrones se pueden plasmar en código.
- ✓ Los patrones de diseño son elementos arquitectónicos más pequeños que los frameworks

Introducción a la plataforma J2EE

La plataforma J2EE (Java™ 2 Platform, Enterprise Edition) soporta un modelo de aplicación distribuida multinivel basada en componentes Java. Los componentes software son unidades binarias de producción, adquisición y despliegue independientes que interactúan entre si para conseguir formar un sistema. En la especificación de J2EE aparecen los siguientes tipos de componentes

- ✓ Componentes clientes: aplicaciones cliente y applets
- ✓ Componentes web: Servlets y JavaServer Pages (JSP)
- ✓ Componentes de negocio: Enterprise JavaBeans (EJB)

Además podemos utilzar componentes JavaBeans pero estos no son considerados parte de la especificación J2EE.

J2EE separa los aspectos de la lógica de negocio de otros soportados nativamente por la plataforma: transacciones, seguridad, ejecución multihilo, gestión de instancias y ciclo de vida de los componentes, etc.

J2EE cubre los aspectos de desarrollo, despliegue y ejecución del ciclo de vida de una aplicación.

Los componentes se ejecutan en contenedores. Un contenedor se encarga de la gestión del ciclo de vida de los componentes y proporciona a estos el acceso a los servicios antes mencionados que aporta la plataforma (transacciones, seguridad, ...). Existen contenedores EJBs, contenedores web, contenedores de aplicación cliente y contenedores de applets (browser + plug-in).

Para desarrollar una aplicación J2EE debemos

- ✓ Implementar los componentes de la aplicación (Servlets, JSPs, EJBs,...)
- ✓ Escribir los descriptores de despligue (ficheros XML) que se encarguen de describir la aplicación en términos de componentes

Componentes de negocio J2EE

Los componentes de negocio encapsulan la lógica que resuelve las necesidades de un determinado dominio de la aplicación. En J2EE se implementan con Enterprise JavaBeans.

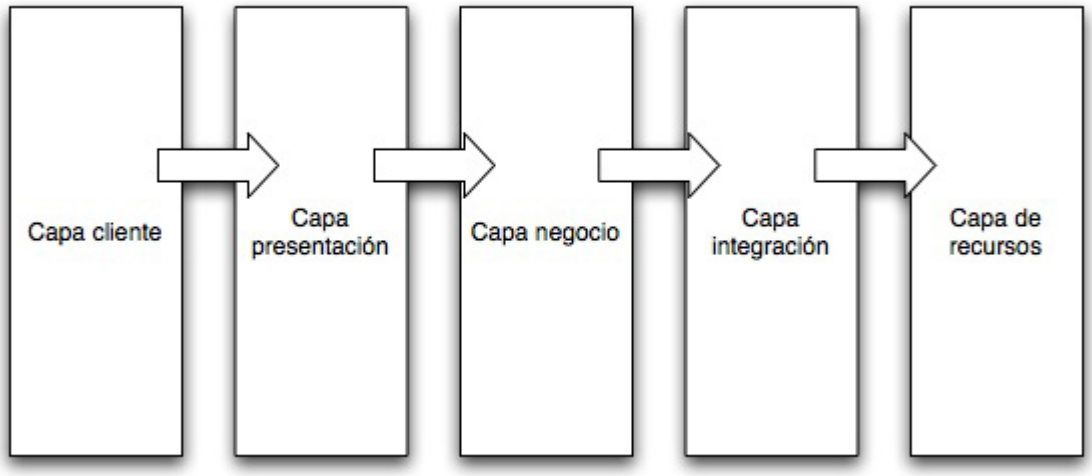
Existen 3 tipos de EJBs

- ✓ EJBs de sesión: Realiza una tarea para un cliente. Su vida está limitada por el tiempo de vida que el cliente interactúa con el servidor. Pueden tener estado (statefull) para modelar procesos conversacionales o pueden prescindir de él (stateless).
- ✓ EJBs de entidad: Representa un objeto de negocio en un sistema de almacenamiento persistente (base de datos, sistema legado, ...)
- ✓ EJBs dirigidos a mensajes: Gestionan mensajes asíncronos (JMS) que llegan al servidor para los que están suscritos

El modelo de desarrollo en capas de J2EE

J2EE al igual que otras plataformas como .NET se basa en la separación de capas. Dicha separación permite una delimitación de responsabilidades a la vez que satisface los requisitos no funcionales de este tipo de aplicaciones (escalabilidad, extensibilidad, flexibilidad, ...) y disminuye el acoplamiento entre las diferentes partes de la misma.

El número de capas puede variar según nuestras necesidades, no obstante, el modelo sugerido y prácticamente estandarizado para aplicaciones J2EE es un modelo en 5 capas



Este modelo de capas ha dejado obsoleto el clásico modelo cliente servidor.

Capa de cliente

Es la capa donde se localizan los diferentes clientes de nuestra aplicación. En entornos empresariales serán de tipos y funcionalidades muy diversos.

Los tipos más frecuentes son aplicaciones de escritorio, navegadores web y clientes para dispositivos móviles.

Capa de presentación

Contiene toda la lógica de interacción entre el usuario y la aplicación. Además está encargada de controlar la interacción entre el usuario y la lógica de negocio, generando las vistas necesarias para mostrar la información en la forma adecuada.

Capa de lógica de negocio

En ella se localiza el código y las reglas que sirven como núcleo de nuestras aplicaciones empresariales. Debe cumplir una serie de requisitos no funcionales como extensibilidad, mantenibilidad, reutilización, flexibilidad y fácil adopción de tecnologías, etc.

Capa de integración

Realiza las tareas necesarias para integrar nuestra aplicación con otros sistemas como sistemas de acceso a datos, sistemas legado, motores de workflow, etc.

Esta capa debe ser extensible para que acepte nuevas fuentes sin que esto afecte a la lógica de negocio.

Capa de recursos

En ella se encuentran los diferentes Sistemas de Información de nuestra empresa: bases de datos, sistemas de ficheros COBOL, sistemas ERP, CRM, etc.

Patrones de la Capa de Presentación

Intercepting Filter

Contexto y problema

El mecanismo que maneja las peticiones que envía la capa de presentación recibe múltiples tipos de solicitudes. Algunas de ellas deberán ser preprocesadas y postprocesadas antes de que lleguen a los componentes que se encargarán de atenderlas.

El conjunto de tareas de preproceso y postproceso que se le aplican a varios tipos de solicitudes puede variar en número, tipo y orden.

Por ejemplo, ciertas peticiones requerirán que antes de ser atendidas el sistema compruebe si el usuario está autenticado en el sistema y tiene una sesión válida, mientras que otras requerirán ser descomprimidas y posteriormente comprobada la validez de su sesión. Por tanto, las formas en las que se combinan las tareas de preprocesamiento y postprocesamiento son variadas y pueden cambiar sin que el resto de componentes de la aplicación sufra cambios.

Las tareas de preprocesamiento se encargan habitualmente de decidir si se atiende o no una petición (¿el usuario está autenticado? ¿son fiables los datos? ¿ha caducado la sesión? ¿la IP emisora de la petición está en la lista de permitidas?) o de modificar la petición (descompresión de datos). Por su parte, las tareas de postprocesado se encargan de modificar la respuesta que el sistema proporciona al cliente (adaptación de la codificación, compresión de datos, etc..)

Necesidades y motivaciones

- ✓ Centralizar la lógica común que se le aplica a las peticiones como chequeos de autenticación, tareas de codificación, filtrado de parámetros, etc…
- ✓ Facilitar y flexibilizar la gestión (adicción/eliminación) de las tareas de preprocesamiento y postprocesamiento para las peticiones que recibe el sistema. Dicha gestión puede ser llevada a cabo en tiempo de despliegue (instalación / configuración)
- ✓ Separar claramente el código de los componentes existentes y el código de las tareas de preprocesado
- ✓ Crear componentes de preprocesamiento y postprocesamiento independientes entre si que puedan ser reutilizados
- ✓ Evitar duplicar el código de preprocesado y postprocesado en múltiples sitios
- ✓ Mejorar la mantenibilidad y flexibilidad de nuestra aplicación

Solución

Usar el patrón *Intercepting Filter* para el pre y post-procesado de peticiones y respuestas.

Intercepting Filter posee un manejador de filtros que combina en una cadena una serie de filtros con bajo acoplamiento a los que se delega el manejo de las peticiones y respuestas. De esta forma podemos añadir, borrar o combinar filtros de multiples formas cambiando únicamente el código del manejador y dejando inalterados el resto de componentes existentes.

Un filtro intercepta las peticiones (request) y las respuestas (response) en su camino hacia un objetivo (generalmente un servlet) y de vuelta al cliente. Su objetivo es

transformarlas o utilizar la información que contienen para tomar decisiones sobre el flujo que deben seguir dentro de la aplicación.

A partir de este momento nos centraremos en los filtros para los servlets (Servlets Filters), pero no debe perder de vista que existen otra serie de filtros, por ejemplo Message Handlers para peticiones SOAP.

Los filtros pre-procesan la petición antes de que ésta llegue al objetivo, el servlet, y/o post-procesan la respuesta después de salir de él. Su existencia es ignorada por el cliente y por el servlet y es totalmente transparente para ambos.

Los filtros no son servlets, porque ellos mismos no crean una respuesta a la petición del usuario (o al menos no es la funcionalidad pensada inicialmente para ellos).

Los filtros son importantes por varias razones. Primero, proporcionan la habilidad para encapsular tareas recurrentes en unidades reutilizables. El código modular es más manejable, más fácil de depurar y puede ser reutilizado con facilidad. Segundo, los filtros pueden ser usados para transformar la respuesta de un servlet o de un JSP. Con la introducción de filtros como parte de la especificación de Java Servlet, los desarrolladores tienen la oportunidad de escribir componentes realmente reutilizables por muchas aplicaciones web.

Un filtro puede realizar diversas transformaciones, tanto a la respuesta como a la petición antes de devolver esos objetos. Un filtro puede:

- ✓ Interceptar la petición a un servlet antes de que esta llegue a su destino
- ✓ Examinar la petición antes de llegar al servlet
- ✓ Modificar las cabeceras y datos de la petición proporcionando una versión personalizada del objeto ServletRequest

- ✓ Interceptar la respuesta del servlet antes de que esta llegue a su destinatario
- ✓ Modificar las cabeceras y datos de la respuesta, proporcionando una versión personalizada del objeto ServletResponse

Estructura

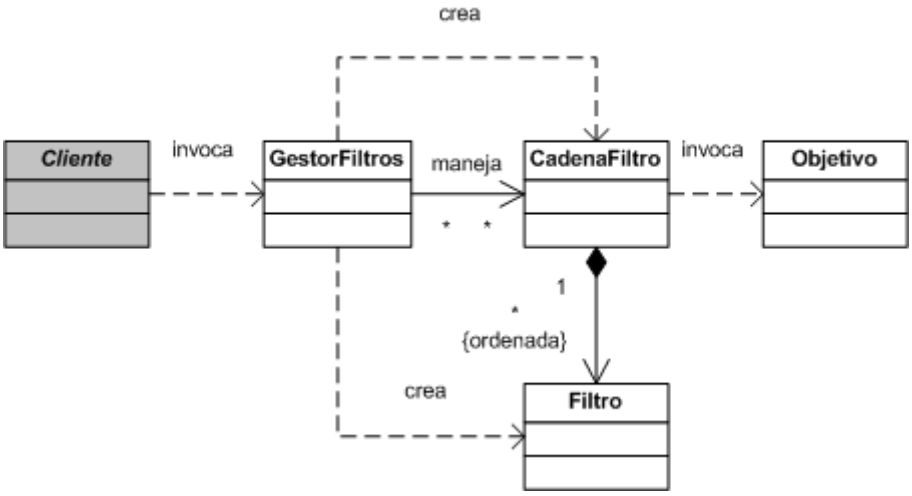
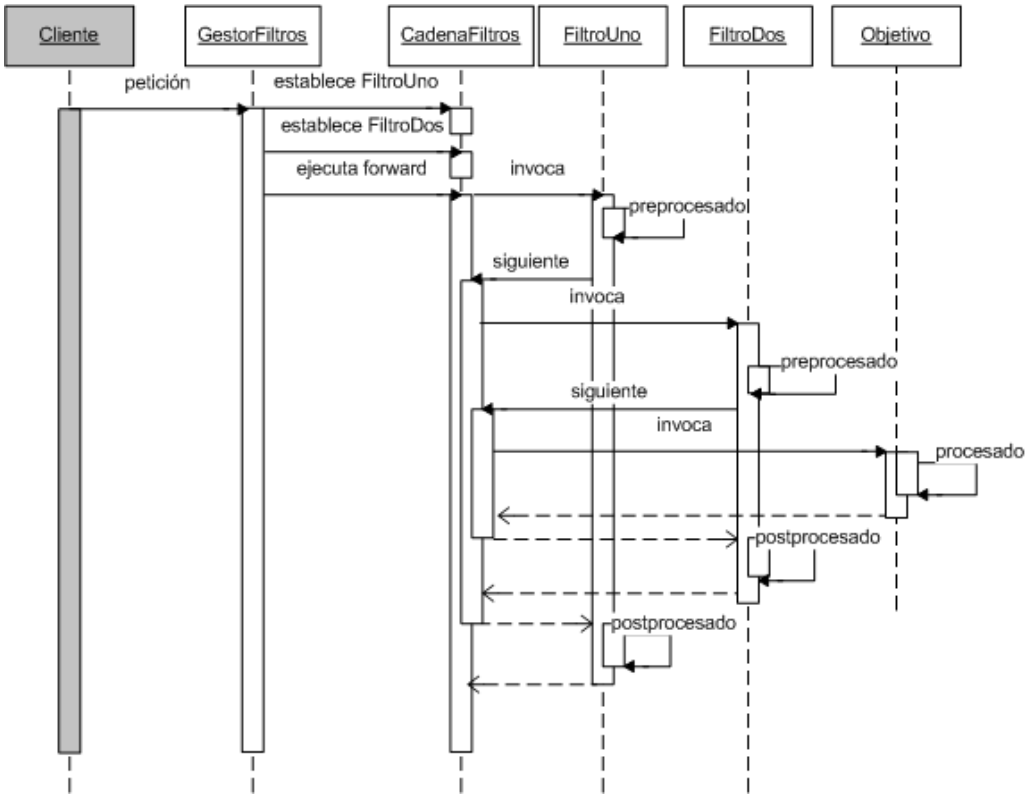


Diagrama de clases del patrón *Filtro de Intercepción*.

Participantes y responsabilidades



Posible diagrama de secuencia del patrón *Filtro de Intercepción*

En el diagrama de secuencia se representa un escenario en el que un cliente solicita la ejecución de un objetivo (por ejemplo un servlet) y cómo el manejador de filtros se responsabiliza de redirigir esa petición a los diferentes filtros que se habían mapeado con anterioridad.

Cliente

Envía la petición al FilterManager

GestorFiltros

Maneja los filtros. Para cada objetivo crea una cadena de filtros con los filtros apropiados y en el orden correcto.

CadenaFiltros

Se encarga de coordinar el flujo de la petición por los filtros. Contiene una colección de filtros independientes y ordenados.

FiltroUno, FiltroDos

Representan filtros individuales que se han mapeado para un objetivo concreto. CadenaFiltros coordina el procesamiento de los filtros.

Objetivo

Recurso concreto que había pedido el cliente (servicio, servlet, ...).

Estrategias

Estrategia estándar de filtrado

Los filtros se controlan declarativamente usando un descriptor de despliegue. La especificación 2.3 de Servlet incluye un mecanismo estándar para construir cadenas de filtros de un modo no obstrusivo, permitiendo añadir o borrar filtros de dichas cadenas.

Los filtros están basados en una interfaz estándar `javax.servlet.Filter` que deben realizar. De esta forma conseguimos filtros reutilizables y con un bajo acoplamiento con el resto de componentes.

Ejemplo

Imagine una aplicación que trabaje con dos tipos de formularios: formularios de datos y formularios con archivos adjuntos. Los formularios pueden ser enviados utilizando diversas codificaciones, por tanto, su información no puede ser capturada directamente invocando métodos `getParameter()`.

Implementaremos un conjunto de filtros que se encarguen de preprocesar las peticiones para recuperar la información correctamente y que se encarguen de traducir las diferentes codificaciones a un formato único consistente.

En nuestro ejemplo podemos encontrarnos con dos tipos diferentes de codificación:

- La codificación estándar de los formularios de tipo `application/x-www-form-urlencoded.`
- La codificación menos usual de tipo `multipart/form-data` utilizada por los formularios que envían archivos adjuntos.

Para cada una de dichas codificaciones implementaremos un filtro que se encargará de preprocesar la petición. Posteriormente, la acción solicitada sobre los datos se hará de forma independiente a la codificación que tuviera el formulario enviado.

Ejemplo de filtro genérico que será especializado - BaseEncodeFilter.java

```
public class BaseEncodeFilter implements
      javax.servlet.Filter {
  private javax.servlet.FilterConfig myFilterConfig;
  public BaseEncodeFilter()     {  }
  public void doFilter(
    javax.servlet.ServletRequest servletRequest,
    javax.servlet.ServletResponse servletResponse,
    javax.servlet.FilterChain filterChain)
  throws java.io.IOException,
    javax.servlet.ServletException {
    filterChain.doFilter(servletRequest,
        servletResponse);
  }
  public javax.servlet.FilterConfig getFilterConfig()
  {
    return myFilterConfig;
  }

  public void setFilterConfig(
    javax.servlet.FilterConfig filterConfig) {
      myFilterConfig = filterConfig;
  }
}
```

Ejemplo de filtro especializado – StandardEncodeFilter.java

```
public class StandardEncodeFilter
  extends BaseEncodeFilter {
  // Constructor vacío
  public StandardEncodeFilter()   {  }
  public void doFilter(javax.servlet.ServletRequest
    servletRequest,javax.servlet.ServletResponse
    servletResponse,javax.servlet.FilterChain
    filterChain)
  throws java.io.IOException,
    javax.servlet.ServletException {
    String contentType =
      servletRequest.getContentType();
    if ((contentType == null) ||
      contentType.equalsIgnoreCase(
        "application/x-www-form-urlencoded"))     {
      translateParamsToAttributes(servletRequest,
        servletResponse);
    }
    filterChain.doFilter(servletRequest,
      servletResponse);
  }
  private void translateParamsToAttributes(
    ServletRequest request, ServletResponse response)
  {
    Enumeration paramNames =
        request.getParameterNames();
    while (paramNames.hasMoreElements())     {
      String paramName = (String)
          paramNames.nextElement();
      String [] values;
```

```
      values = request.getParameterValues(paramName);
      System.err.println("paramName = " + paramName);
      if (values.length == 1)
       request.setAttribute(paramName, values[0]);
      else
        request.setAttribute(paramName, values);
    }
 }
}
```

Ejemplo filtro especializado – MultipartEncodeFilter.java

```
public class MultipartEncodeFilter extends
  BaseEncodeFilter {
  public MultipartEncodeFilter() { }
  public void doFilter(javax.servlet.ServletRequest
    servletRequest, javax.servlet.ServletResponse
    servletResponse,javax.servlet.FilterChain
    filterChain)
  throws java.io.IOException,
    javax.servlet.ServletException {
    String contentType =
      servletRequest.getContentType();
    // Sólo filtramos la petición si es multipart encoding
    if (contentType.startsWith(
                "multipart/form-data")){
      try {
        String uploadFolder =
          getFilterConfig().getInitParameter(
```

```
            "UploadFolder");
      if (uploadFolder == null) uploadFolder = ".";
      /** The MultipartRequest class is:
      * Copyright (C) 2001 by Jason Hunter
      * <jhunter@servlets.com>. All rights reserved.
      **/
      MultipartRequest multi = new
        MultipartRequest(servletRequest,
                         uploadFolder,
                         1 * 1024 * 1024 );
      Enumeration params =
               multi.getParameterNames();
      while (params.hasMoreElements()) {
        String name = (String)params.nextElement();
        String value = multi.getParameter(name);
        servletRequest.setAttribute(name, value);
      }
      Enumeration files = multi.getFileNames();
      while (files.hasMoreElements()) {
        String name = (String)files.nextElement();
        String filename = multi.getFilesystemName(name);
        String type = multi.getContentType(name);
        File f = multi.getFile(name);

      }
    }
    catch (IOException e)
    {
      LogManager.logMessage(
        "error reading or saving file"+ e);
    }
  } // fin if
```

```
    filterChain.doFilter(servletRequest,
                         servletResponse);
  } // fin método doFilter()
}
```

Ejemplo de descriptor de despliegue – web.xml

En el archivo XML de despliegue tenemos que declarar los filtros y mapearlos a los servlets construyendo cadenas de filtros ordenadas.

En nuestro ejemplo tenemos dos filtros `StandardEncodeFilter` y `MultipartEncodeFilter`. La descripción de los filtros se encapsularía dentro de las etiquetas `<filter>` y `</filter>`. El nombre de un filtro lo definimos con el contenido de `<filter-name>` y nos servirá para mapear los filtros posteriormente a los target. `<filter-class>` debe contener la ruta de paquetes hasta el servlet que implementa el filtro. El orden de los filtros en la cadena asociada a un target es el que aparece en el descriptor de despliegue (el de las declaraciones `<filter-mapping>`).

Los filtros pueden mapearse no sólo a un servlet concreto sino a un conjunto de ellos. Para hacer esto último tendremos que utilizar expresiones regulares en el contenido de `<url-pattern>`.

```
.
.
.
<filter>
```

```
    <filter-name>StandardEncodeFilter</filter-name>
    <display-name>StandardEncodeFilter</display-name>
    <description></description>
    <filter-class>filters.encodefilter.
            StandardEncodeFilter</filter-class>
</filter>
<filter>
    <filter-name>MultipartEncodeFilter</filter-name>
    <display-name>MultipartEncodeFilter</display-name>
    <description></description>
    <filter-class>filters.encodefilter.
            MultipartEncodeFilter</filter-class>
    <init-param>
      <param-name>UploadFolder</param-name>
      <param-value>/home/files</param-value>
    </init-param>
</filter>
.
.
.
<filter-mapping>
    <filter-name>StandardEncodeFilter</filter-name>
    <url-pattern>/EncodeTestServlet</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>MultipartEncodeFilter</filter-name>
    <url-pattern>/EncodeTestServlet</url-pattern>
</filter-mapping>
.
.
```

Consecuencias

- ✓ Centraliza el control con manejadores poco acoplados
- ✓ Aumenta la reusabilidad. Los filtros mejoran el particionamiento de la aplicación y facilitan la reutilización del código de preprocesado y postprocesado.
- ✓ Configuración flexible y declarativa. Múltiples filtros pueden ser permutados de forma fácil y sencilla gracias a los descriptores de despliegue.
- ✓ El paso de información entre los filtros es ineficiente. Los filtros, al ser completamente independientes unos de otros, no comparten información, por lo que esta tendrá que ser transvasada a todos ellos. Si el volumen de información es grande puede ser una operación costosa.

Patrones J2EE relacionados

- ✓ Front Controller: Resuelve problemas similares pero para los procesamientos centrales en vez de para los preprocesamientos y postprocesamientos.

View Helper

Contexto y problema

El sistema genera la presentación de los contenidos que se mostrará al usuario y para ello necesita procesar los datos de negocio obtenidos dinámicamente.

De las tres capas existentes en las aplicaciones empresariales, la capa de presentación es la que está más expuesta a sufrir cambios. Si mezclamos en ella la lógica de acceso a los datos de negocio y el código de formateo de los mismos, entonces, el mantenimiento y la reutilización se ven perjudicados y el sistema se vuelve más difícil de modificar.

Además, en los equipos de desarrollo suele haber personas que no conocen Java y que sin embargo si conocen los lenguajes de marcas (HTML, XHTML, XML, …) que se usan en el desarrollo web. Dichas personas deben poder trabajar la capa de presentación sin tener conocimiento de las tareas que desarrollan los programadores.

Necesidades y motivaciones

- ✓ Queremos evitar embeber lógica programada en las vistas. Este código embebido es proclive a fallos ya que alienta al programador a usar la técnica de copy & paste para reutilizarlo en múltiples vistas y deteriora la mantenibilidad de nuestros componentes
- ✓ Queremos usar un sistema de plantillas para las vistas
- ✓ Es deseable promover una separación limpia entre las tareas adjudicadas a cada uno de los miembros del equipo de desarrollo creando los roles de desarrollador de software y desarrollador/diseñador web

Solución

Las vistas delegan la obtención y procesamiento/adaptación del modelo a unos nuevos componentes denominados ayudantes (Helpers).

Encapsular la lógica de negocio en un ayudante en vez de en una vista hace nuestro código más modular, limpio y reutilizable. Múltiples clientes (controladores o vistas) podrían utilizar un ayudante para acceder y adaptar el mismo estado del modelo y presentarlo de distinta forma. Si la lógica de negocio estuviera en las vistas, la única forma de reutilizarla sería copiar y pegar, haciendo más arduo el mantenimiento del código.

Una señal inequívoca de que necesita aplicar este patrón en su aplicación es cuando tiene mucho código scriptlet embebido en las vista. Con la creación de ayudantes la lógica de negocio saldría fuera de las vistas.

No obstante, si el código embebido era utilizado para tareas de filtrado o control común a lo largo de múltiples peticiones, como por ejemplo, tareas de autenticación, compresión/descompresión, adaptación de los datos de las peticiones, etc…, entonces será conveniente que le eche un vistazo a los patrones *Intercepting Filter* y *Front Controller*. Dicha lógica de negocio es mejor encapsularla en componentes centralizados que actúen antes que las vistas.

Estructura

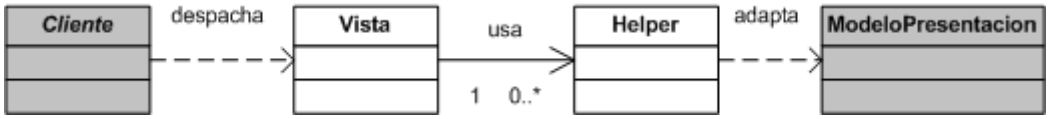
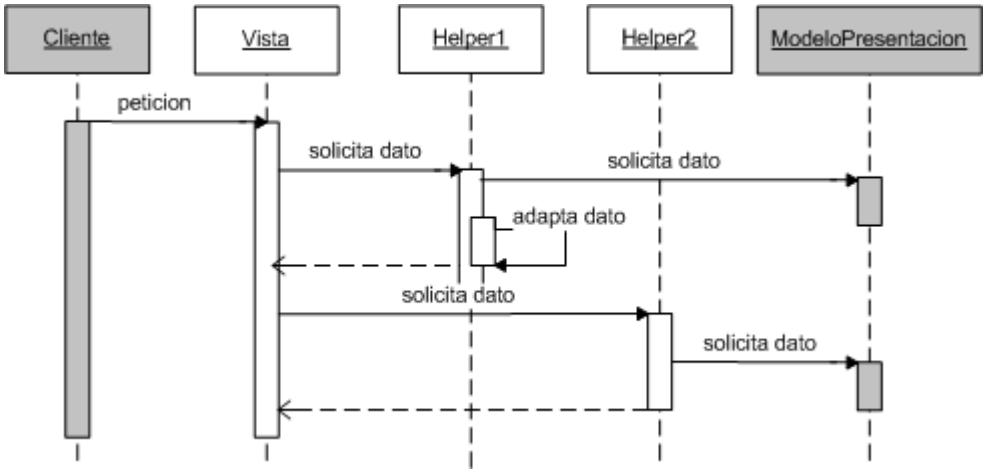


Diagrama de clases del patrón *View Helper*

Note la multiplicidad de la asociación entre la vista y los Helpers. Cada una de ellas puede tener varios ayudantes.

Participantes y responsabilidades



Posible diagrama de secuencia del patrón *View Helper*

En el diagrama de secuencias se muestra cómo una vista puede hacer uso de dos ayudantes. El primero de ellos se encarga tanto de almacenar el modelo intermedio (datos) que necesita la vista como de adaptarlos. El segundo sin embargo tan sólo los almacena.

Cliente

Consume la vista. El cliente podría ser, por ejemplo, un *Front Controller* que hubiera mediado entre la solicitud del cliente y la vista.

Vista

Representa y muestra la información al cliente. La información dinámica es obtenida y convertida por un Helper.

Helper1, Helper2

Encapsula la lógica para procesar o adaptar los datos del modelo. Pueden ser JavaBean o tags (JSTL o personalizados). Ambos contienen los datos de negocio a los que queríamos acceder, o lo que es lo mismo, un estado intermedio del modelo que será usado por la vista.

Los Helpers juegan el rol de Objeto de Transferencia entre la capa de negocio y la capa de presentación (*Transfer Object*).

Estrategias

A continuación se explican dos estrategias para implementar la Vista y dos estrategias para implementar al Ayudante.

Estrategia JSP para la vista

Esta estrategia sugiere usar una JavaServlet Page como componente para la Vista. Las principales ventajas de hacer esto es que los Diseñadores Web están más acostumbrados a ver lenguajes de marcado que código Java.

A continuación se muestra un ejemplo en el que se ha usado un JSP para codificar la Vista. Puede notar que la Vista hace uso de un Ayudante implementado con un JavaBean.

```
<jsp:useBean id="ayudanteBienvenida" scope="request"
  class="ayudantes.ayudanteBienvenida" />

<html>
<body>
<% if (ayudanteBienvenida.ExisteNombre())
{
%>
<div id=”saludo”>
<h3>Bienvenido
<jsp:getProperty name="ayudanteBienvenida" property="nombre" />
</h3>
<%
}
%>
<h4>Gracias por visitar nuestro sitio</h4>

</body>
</HTML>
```

Estrategia Servlet para la vista

La opción alternativa a JSP es usar un Servlet y embeber las etiquetas de marcado (XML, XHTML, …) que serán escritas hacia un objeto PrintWriter. Esta estrategia dificulta la separación de roles entre los miembros de los equipos de trabajo ya que, como

comentamos anteriormente, en estos suele haber personas que no tienen conocimientos de Java. Además, el código resultante es menos claro y, por tanto, más difícil de actualizar y modificar. No obstante, semánticamente esta estrategia es equivalente a la anterior.

A continuación se muestra un Servlet que hace las veces de Vista.

```
public class Controller extends HttpServlet {

    ...

  /** Procesa las peticiones HTTP, tanto GET como POST
   */
  protected void processRequest(HttpServletRequest
    request, HttpServletResponse response)
    throws ServletException, java.io.IOException {
    String title = "Servlet View Strategy";
    try {
      response.setContentType("text/html");
      java.io.PrintWriter out = response.getWriter();
      out.println("<html><title>"+title+"</title>");
      out.println("<body>");
      out.println("<h2><center>Employees List</h2>");
      EmployeeDelegate delegate =
          new EmployeeDelegate();

      /** ApplicationResources permite acceder mediante
        * una API a constantes y otros valores
        * preconfigurados **/
      Iterator employees = delegate.getEmployees(
```

```
            ApplicationResources.getInstance().
                getAllDepartments());
      out.println("<table border=2>");
      out.println("<tr><th>First Name</th>" +
        "<th>Last Name</th>" +
          "<th>Designation</th><th>Id</th></tr>");
      while (employees.hasNext())                          {
        out.println("<tr>");
        EmployeeTO emp = (EmployeeTO)employees.next();
        out.println("<td>"+emp.getFirstName()+
            "</td>");
        out.println("<td>"+emp.getLastName()+
            "</td>");
        out.println("<td>"+emp.getDesignation()+
            "</td>");
        out.println("<td>"+emp.getId()+"</td>");
        out.println("</tr>");
}
      out.println("</table>");
      out.println("<br><br>");
      out.println("</body>");
      out.println("</html>");
      out.close();
    }
    catch (Exception e) {
      LogManager.logMessage("Handle this exception",
        e.getMessage() );
    }
  }

  /** Handles the HTTP <code>GET</code> method.
    * @param request servlet request
```

```
    * @param response servlet response
    */
  protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, java.io.IOException {
        processRequest(request, response);
  }

  /** Handles the HTTP <code>POST</code> method.
    * @param request servlet request
    * @param response servlet response
    */
  protected void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, java.io.IOException {
        processRequest(request, response);
  }
    ...
}
```

Estrategia JavaBean para el Ayudante

El ayudante se implementa mediante un componente JavaBean. De esta forma separamos la presentación de la lógica. La lógica encapsulada en el JavaBean se encargará de recuperar el contenido dinámico y de almacenar y adaptar el estado del modelo para que la vista lo use.

Esta estrategia es muy fácil de aplicar y los desarrolladores sin experiencia serán capaces de acostumbrarse rápidamente al desarrollo de componentes JavaBean. No obstante,

esta estrategia está limitada ya que los JavaBean tan sólo tienen métodos get y set lo que obliga en ocasiones a añadir código de la lógica en las vistas.

Estrategia de Tags personalizados para el Ayudante

El ayudante se implementa vía tags personalizados (disponibles a partir de JSP 1.1 o superior).

Implementar los tags personalizados es más complejo que implementar JavaBeans. El desarrollo de tags personalizados conlleva la codificación del tag, de un descriptor de librería, la declaración de los tags en el descriptor de despliegue, etc…

Esta estrategia es más potente que la estrategia JavaBean para el Ayudante porque no está limitada a usar métodos get/set, con lo que eliminaremos completamente la lógica (scriptles) de nuestras vistas.

A continuación se muestra un ejemplo en el que la vista imprime un listado de empleados. Tan sólo mostramos la vista y no el código de los artefactos que sería necesario implementar.

Ejemplo de vista con etiquetas personalizadas – listado_empleados.jsp

```
<%@ taglib uri="/WEB-INF/tienda.tld" prefix="tienda"%>
<html>
 <head>Ejemplo de vista con etiquetas personalizadas</head>
 <body>
    <h1>Listado de empleados</h1>
    <tienda:listado entidad=”empleado” filtro=”ninguno” numero=”10”/>
 </body>
```

```
</html>
```

Estrategia XSLT para el ayudante

El ayudante es implementado mediante una hoja de transformación XSLT. Esta estrategia está indicada cuando el modelo está estructurado con un lenguaje de marcado (XML).

Consecuencias

- ✓ Mejora la separación de tareas en la aplicación. La presentación y los procesos de negocio quedan claramente separados
- ✓ Ayuda a la separación de roles en nuestro equipo de desarrollo
- ✓ Eliminamos el código Scriptlet embebido en las vistas
- ✓ Facilitamos las tareas de testeo
- ✓ Si usamos los Helpers con la misma filosofía de los Scriptlets, esto es, los usamos exponiendo demasiado los detalles de implementación, entonces no resolvemos nada y la aplicación del patrón se vuelve inútil. Un ejemplo sería codificar etiquetas personalizadas que hicieran las tareas de control condificional que hacían las sentencias if y else en los scriptlets y usarlas de manera intensiva.

Patrones J2EE relacionados

- ✓ *Business Delegate*: Los ayudantes necesitarán acceder a servicios de negocio. Para reducir el acoplamiento entre la capa de presentación (en la que se encuentran los Helpers) y la capa de negocio es deseable que hagamos los accesos vía un delegado de negocio. Éste oculta a los ayudantes las tareas de búsqueda de los servicios y cacheo de resultados.

- ✓ *Front Controller*: Normalmente el cliente que solicite las vistas será un objeto de control centralizado. Esto será así cuando utilizemos *Service to Worker*.

Composite view

Contexto y problema

En los portales corporativos y páginas empresariales es usual que varios componentes de la presentación aparezcan en múltiples páginas. Esto ocurre con menús, encabezados, pies de página, formularios de búsqueda, titulares, etc. Las páginas que el usuario visualiza obtienen datos de múltiples fuentes y están compuestas por diferentes secciones o subvistas. Dichas subvistas pueden estar maquetadas en diferentes zonas dependiendo de la vista que queramos componer.

Las vistas se construyen embebiendo código (etiquetas HTML, por ejemplo) directamente en ellas. Si varias vistas comparten una misma subvista ambas deben embeber su código en el lugar adecuado dependiendo de su maquetación.

Modificar la maquetación de múltiples vistas es una tarea compleja y propensa a errores. Así mismo, la modificación de la información de una subvista requiere que los cambios se hagan en todas las vistas.

Necesidades y motivaciones

- ✓ Queremos reutilizar subvistas de tal forma que podamos componer nuevas vistas compuestas que incluyan a estas. Las subvistas aparecerán posicionadas en distintos lugares dependiendo de la maquetación elegida para la nueva vista.
- ✓ Necesitamos independizar la maquetación del contenido y gestionar ambas partes de forma independiente.
- ✓ El contenido de las subvistas incluidas cambia con frecuencia o debe ser protegido para que sólo ciertos usuarios puedan modificarlo.

- ✓ Queremos evitar la duplicación de subvistas comunes a varias vistas ya que esta práctica dificulta la mantenibilidad del código.

Solución

Usar *Composite View* para generar vistas compuestas de subvistas atómicas según las directrices de una plantilla (template), de tal forma que cada componente de la plantilla será incluido dinámicamente en el todo y la apariencia de la página pueda ser manejada independientemente del contenido.

Estructura

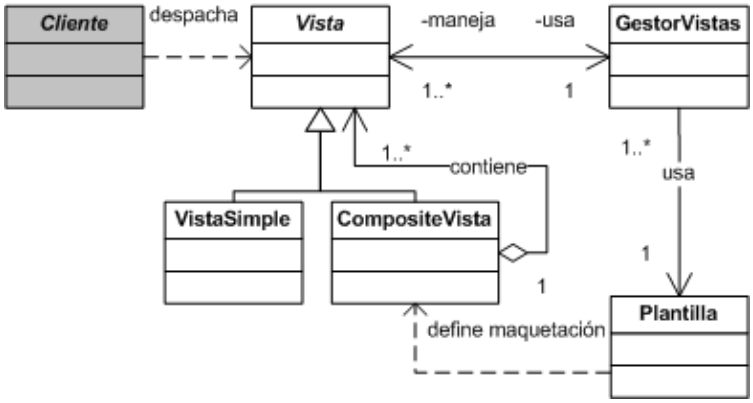


Diagrama de clases del Patrón *Composite View*

Participantes y responsabilidades

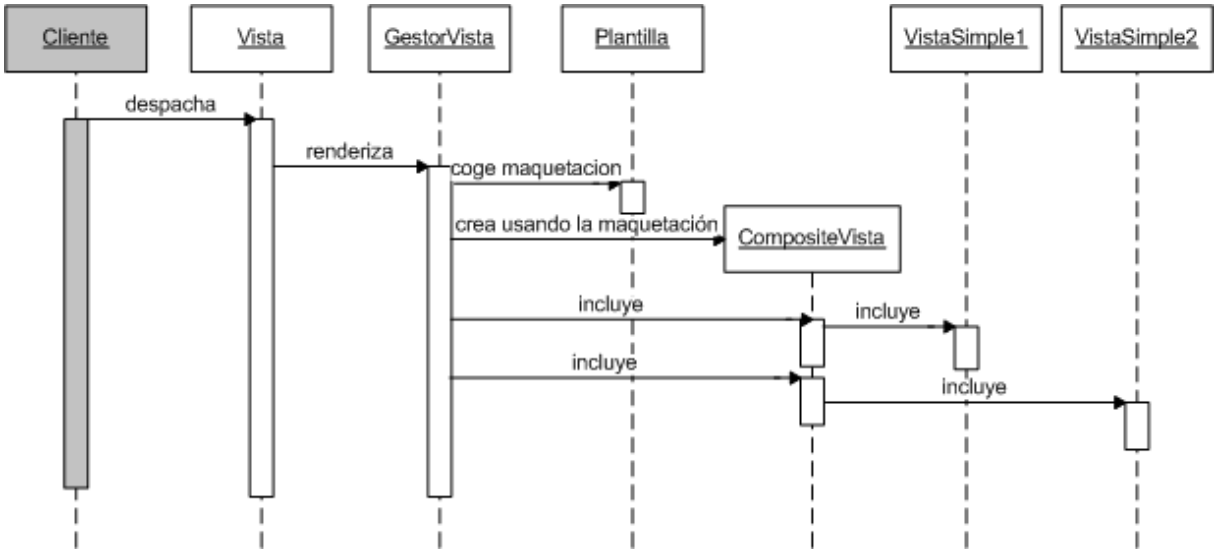


Diagrama de secuencia del *Patrón Composite View*

Cliente

Despacha la vista. Generalmente el rol del cliente lo toma un *Application Controller* o, en menos ocasiones, un *Front Controler*.

Vista

Presenta al usuario información extraida de varias subvistas según la maquetación que define la plantilla.

GestorVista

Se encarga de ensamblar las diferentes subvistas según las directrices de la plantilla

Plantilla

Contiene información sobre la maquetación

CompositeVista

Vista compuesta por vistas simples o por otras vistas que a su vez son compuestas.

VistaSimple1, VistaSimple2

Vistas simples que poseen contenido.

Estrategias

Estrategia estándar para la gestión de vistas

Implementamos View Manager haciendo uso de etiquetas estándar de JSP, en concreto `<jsp:include>`. Esta estrategia es muy fácil de implementar pero no aporta la flexibilidad y potencia que requieren los grandes proyectos. Una desventaja clara de esta estrategia es que la información de maquetación de cada página está embebida en ella. Como consecuencia, la modificación de la maquetación requerirá la modificación de múltiples JSP.

Ejemplo de vista compuesta con generación en tiempo de ejecución – todo.jsp

```
<%@ page contentType="text/html;charset=ISO-8859-1"%>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <title>Ejemplo Patr&oacute;n Composite View - Inclusi&oacute;n en tiempo
de Ejecuci&oacute;n</title>
    <jsp:include page="banner.html" flush="true"/>
    <table width="100%">
      <tr align="left" valign="middle">
        <td width="20%">
        <jsp:include page="panelUsuario.jsp" flush="true"/>
        </td>
        <td width="70%" align="center">
        <jsp:include page="contenido.jsp" flush="true"/>
        </td>
      </tr>
    </table>
    <jsp:include page="pieDePagina.html" flush="true"/>
  </body>
</html>
```

En este ejemplo puede ver cómo hemos implementado el View Manager y cómo, efectivamente, la información de maquetación está incluida en el código del JSP.

Las subvistas incluidas contienen tanto contenido estático (páginas HTML) como contenido dinámico (generado por otras páginas JSP). Dichas subvistas se incluyen en

tiempo de ejecución, por tanto, las subvistas pueden sufrir modificaciones *on the fly* que se plasmarán en las vistas la próxima vez que un usuario las solicitara. Esta característica nos aporta dinamismo pero tiene un impacto negativo en el rendimiento.

Una alternativa es que las subvistas se incluyan en tiempo de traducción de tal modo que el rendimiento no se vea afectado. No obstante, perderíamos dinamismo ya que las modificaciones de las subvistas no serían efectivas en las vistas hasta que no volviéramos a recompilarlas. A continuación se muestra el último ejemplo con inclusión en tiempo de traducción.

Ejemplo de vista compuesta con generación en tiempo de traducción – todo2.jsp

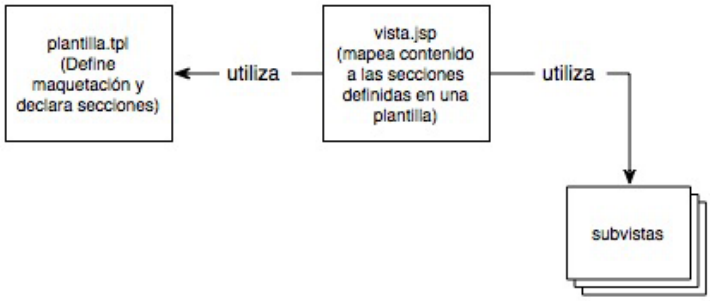
```
<%@ page contentType="text/html;charset=ISO-8859-1"%>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <title>Ejemplo Patr&oacute;n Composite View - Inclusi&oacute;n en tiempo
de Traducci&oacute;n</title>
    <%@ include file="banner.html" %>
    <table width="100%">
      <tr align="left" valign="middle">
        <td width="20%">
        <%@ include file="panelUsuario.jsp" %>
        </td>
        <td width="70%" align="center">
        <%@ include file="contenido.jsp" %>
        </td>
      </tr>
    </table>
    <%@ include file="pieDePagina.html" %>
```

```
  </body>
</html>
```

Estrategia de etiquetas personalizadas para la gestión de vistas

Esta estrategia es la más adecuada ya que aporta flexibilidad y potencia al manejo de la maquetación y el contenido de las vistas. Además, permite personalizar fácilmente la maquetación en función de las políticas de seguridad o los roles de los usuarios. No obstante, su implementación requiere más esfuerzo por parte de los programadores ya que tendrán que desarrollar etiquetas personalizadas y los numerosos artefactos que ello conlleva. Una alternativa para evitar esta desventaja es usar etiquetas desarrolladas por terceros como las del siguiente ejemplo.

El ejemplo consta de varios ficheros. Las vistas serán ficheros .jsp en los que, a diferencia de la estrategia anterior, no existe información sobre la maquetación. Dicha información estará contenida en un fichero de plantilla. La plantilla se encargará de definir las secciones que aparecen en una maquetación dada. Es responsabilidad de la vista mapear los contenidos que tendrán las secciones.



Ejemplo plantilla – plantilla.tpl

```
<region:render section='banner'/>
<table width="100%">
    <tr align="left" valign="middle">
        <td width="20%">
      <!-- menu region -->
      <region:render section='controlpanel' />
        </td>
        <td width="70%" align="center">
      <!-- contents -->
      <region:render section='mainpanel' />
        </td>
    </tr>
</table>
```

Ejemplo mapeo de contenido a la plantilla – vista.jsp

```
<region:render template='plantilla.tpl'>
```

```
  <region:put section='banner' content = 'banner.jsp' />
  <region:put section = 'controlpanel' content =
      'PanelPerfil.jsp' />
  <region:put section='mainpanel' content =
      'PanelPrincipal.jsp' />
  <region:put section='footer' content='pie_pagina.jsp' />
</region:render>
```

Consecuencias

- ✓ Creamos un diseño modular y más mantenible
- ✓ Reutilizamos las subvistas
- ✓ La construcción de nuestras vistas se hace más flexible y potente
- ✓ Reduce el rendimiento de nuestra aplicación si nuestra estrategia incluye las subvistas en tiempo de ejecución

Patrones J2EE relacionados

- ✓ View Helper: La estrategia de transformación XSLT para el ayudante es una alternativa a este patrón

Context Object

Contexto y problema

Las aplicaciones acceden a Sistemas de Información que están fuera de su contexto mediante protocolos haciendo uso de APIs específicas. Usualmente la información consultada está relacionada con la seguridad, las peticiones del cliente o la configuración y la utilizan múltiples componentes en repetidas ocasiones.

Cuando los componentes de dicha aplicación utilizan un Sistemas de Información se hacen dependientes de su protocolo específico. Por tanto, cada uno de los componentes queda fuertemente acoplado con la interfaz del protocolo reduciendo así su flexibilidad y reusabilidad.

Por ejemplo, los componentes web reciben solicitudes HTTP. Si el resto de componentes trabajaran con dichas solicitudes quedarían fuertemente acoplados a ella y cualquier cambio en el protocolo o en los detalles de acceso a la información que queremos recuperar afectaría a todos y cada uno de ellos. Piense que si accediéramos al campo de un formulario y este cambiara de nombre todos los componentes que utilizaran dicha información dejarían de poder acceder a ella.

En las aplicaciones empresariales generalmente existen varios tipos de clientes que interactúan por medio de distintos protocolos. Si nuestros componentes son dependientes de un protocolo específico, entonces, deberemos duplicarlos y especializarlos para cada uno de los protocolos utilizados.

Necesidades y motivaciones

- ✓ Tenemos componentes y servicios que acceden a Sistemas de Información externos vía APIs específicas de sus respectivos protocolos de comunicación

- ✓ Queremos desacoplar los componentes y servicios de los protocolos específicos de los Sistemas de Información que accedamos
- ✓ Tenemos múltiples clientes que utilizan protocolos distintos
- ✓ En sistemas multicapa, alguna de las capas puede querer acceder a información de otra capa sin acoplarse a esta
- ✓ Sólo queremos exponer a cada componente las APIs relevantes en su contexto

Solución

Encapsular el estado en un *Context Object* para que los componentes no estén acoplados a protocolos específicos de Sistemas de Información externos.

Recuperando el ejemplo, podemos encapsular las peticiones HTTP en un *Context Object* de manera que el resto de componentes accedan a dicho estado a través de él y sean independientes al protocolo HTTP. De esta manera, si tenemos varios clientes que usan protocolos distintos podremos reutilizar los componentes y tan sólo deberemos reescribir la parte de nuestra aplicación que construye el objeto de contexto.

No debe confundir los *Context Object* con los *Transfer Object*. Los primeros tienen como objetivo reducir la dependencia de sus componentes con sistemas externos y los segundos están diseñados para reducir las comunicaciones remotas e invocaciones a métodos de fina granularidad entre distintas capas de nuestra aplicación.

Estructura

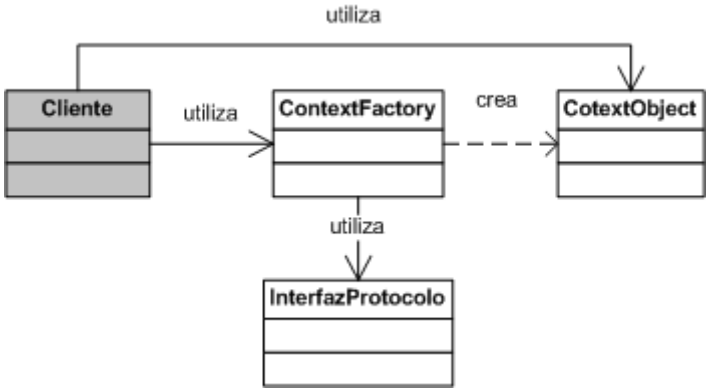


Diagrama de clases del patrón *Context Object*

Participantes y responsabilidades

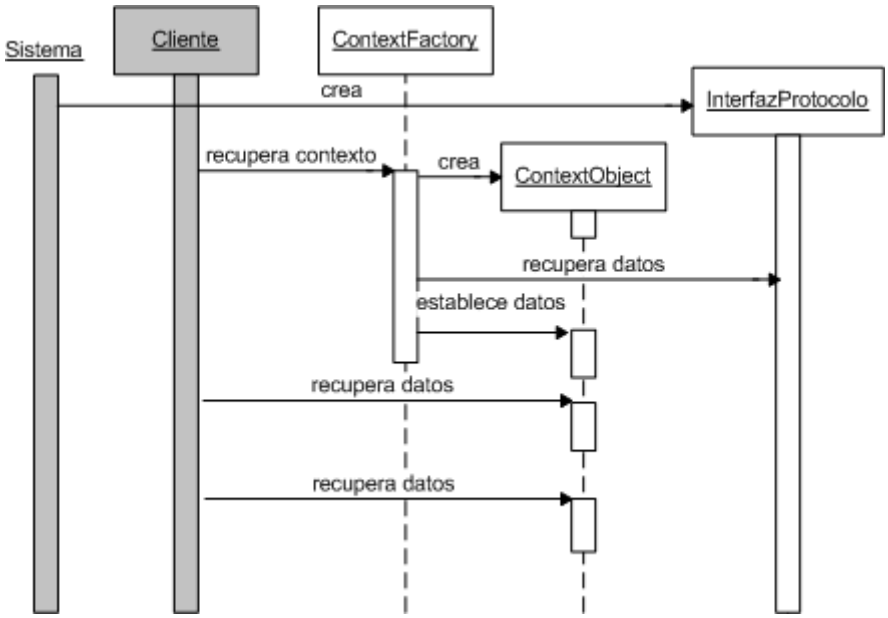


Diagrama de secuencia del patrón *Context Object*

El Cliente usa un objeto factoría para crear un Context Object que a su vez usará a la interfaz específica del protocolo que queremos ocultar al resto de componentes. Observe cómo el cliente se independiza de la interfaz del protocolo.

ProtocolInterface

Interfaz del protocolo que usará la factoría para obtener los datos con los que creará el objeto de contexto.

ContextFactory

Tiene por responsabilidad crear Context Objects independientes del protocolo

ContextObject

Objeto genérico usado para encapsular los interfaces específicos de los protocolos.

Estrategias

Estrategia del Mapa de contexto

Es la estrategia más simple para implementar un objeto de contexto que se encargue de adaptar el estado de las solicitudes de los clientes. Se basa en la transferencia del estado a un Map estándar.

Veamos una posible solución usando esta estrategia para la transferencia de los parámetros enviados vía HTTP por un cliente que ha rellenado un formulario en su navegador.

Ejemplo de componente que tomaría las responsabilidades de creación de los objetos de contexto - FrontController.java

```
package control;
```

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.util.Map;
import java.util.HashMap;

public class FrontController extends HttpServlet
{
  private static final String CONTENT_TYPE = "text/html; charset=ISO-8859-1";

  public void init(ServletConfig config) throws ServletException
  {
    super.init(config);
  }

  public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
  {
    procesarPeticion(request, response);
  }

  public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
  {
    procesarPeticion(request, response);
  }

  public void procesarPeticion(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException
  {
    /* Creamos el objeto de contexto. */
```

```
    Map requestContextMap = new HashMap(request.getParameterMap());
    /* A partir de este punto el resto de componentes trabajarían
     * con el objeto de contexto que encapsula las peticiones HTTP */
    ApplicationController s = new ApplicationControllerImp();
    s.handleRequest(requestContextMap);
     //...
  }

}
```

Ejemplo componente independiente del protocolo - ApplicationControllerImp.java

```
package control;
import java.util.Map;
import java.util.Collection;
import java.util.Iterator;

public class ApplicationControllerImp implements ApplicationController
{
  public ApplicationControllerImp(){}

  /* Manejamos la petición a través de un objeto Map
   * Este componente NO está acoplado con el protocolo HTTP
   * ni conoce los detalles para acceder a la información
   * de las peticiones HTTP.
   *
   * */
  public ResponseContext handleRequest(Map requestContextMap)
  {
```

```
    Collection parametros = requestContextMap.values();
    Iterator it = parametros.iterator();
    while(it.hasNext())
    {
      //....
    }
  }
}
```

Estrategia POJO para Objetos de Contexto de peticiones

Otra posible aproximación es usar un POJO (JavaBean) para encapsular la petición. De esta manera podemos, además de guardar el estado de la petición, asociarle al objeto diferentes operaciones que puedan ser interesantes en nuestra aplicación, por ejemplo, funciones de validación del estado almacenado o funciones que generen propiedades complejas no almacenadas en el estado basándose en otras que si lo estén.

Ejemplo de objeto de contexto POJO – CustomerBean.java

```
public class CustomerBean {
    // Atributos del Bean
    private String lastName;
    private String firstName;
    private String street;
    private String city;
    private String state;
    private String postalCode;
    private String phone;
```

```
    protected boolean nullOrBlank (String str) {
        return ((str == null) || (str.length() == 0));
    }
    // Método para validar el estado de la solicitud
    public  Error validate(HttpServletRequest request) {
        Error errors = new Error();
        if (nullOrBlank(lastName)) {
            errors.add("lastName",
                    new Error("LastName missing"));
        }
        if (nullOrBlank(firstName)) {
            errors.add("firstName",
                    new Error("FirstName missing"));
        }
        // Más reglas de validación
        // ...
        if (nullOrBlank(phone)) {
            errors.add("phone",
                    new Error("Phone missing"));
        }
        return errors;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getFirstName() {
```

```
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    // Más métodos consultores / modificadores (uno por atributo)
    // ...

    public void setPhone(String phone) {
        this.phone = phone;
    }
}
```

La gran desventaja de esta estrategia con respecto a Request Context Map Strategy es que requiere demasiado trabajo repetitivo y engorroso (hay que escribir métodos consultores y modificadores para cada atributo). No obstante existen herramientas en el mercado que facilitan su creación basándose en información declarativa. Por ejemplo, el framework Struts asocia a cada formulario un objeto POJO que encapsula sus datos. El programador puede generarlos de manera automática proporcionándole al framework información sobre el formulario en un fichero XML.

Ejemplo Creación de un Bean que actúa como objeto de contexto usando DynaActionForm del framework Struts – web.xml

```
<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
```

```
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd" >
<struts-config>
    <!-- Form Bean Definitions -->
    <form-beans>
        <form-bean name="employeeForm"
type="org.apache.struts.validator.DynaValidatorForm">
            <form-property name="name" type="java.lang.String"/>
            <form-property name="age" type="java.lang.String"/>
            <form-property name="department" type="java.lang.String"
initial="2" />
            <form-property name="flavorIDs" type="java.lang.String[]"/>
            <form-property name="methodToCall" type="java.lang.String"/>
        </form-bean>
    </form-beans>
```

Consecuencias

- ✓ Mejora la reusabilidad y mantenibilidad de nuestros componentes al ser estos más genéricos. Varios tipos de clientes podrán reutilizarlos.
- ✓ Reduce el rendimiento de la aplicación al tener que transpasar el estado de un objeto a otro. Esta reducción en el rendimiento suele ser aceptable teniendo en cuenta las ventajas que nos aporta en términos de reusabilidad y mantenibilidad.
- ✓ Facilita el testeo de los componentes al eliminar dependencias con protocolos. Nuestras herramientas de testeo (JUnit por ejemplo) podrán trabajar directamente con objetos de contexto.

Patrones J2EE relacionados

- ✓ Front Controller: este componente suele ser el responsable de la creación de los objetos de contexto.
- ✓ Intercepting Filter: la alternativa a *Front Controller* para la creación de *Context Object*.

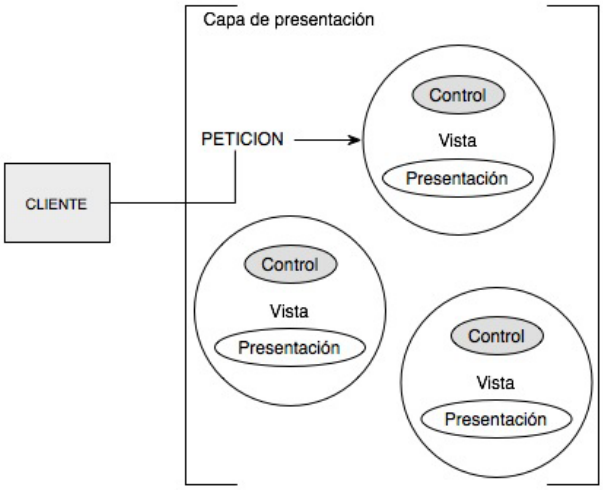
Front Controller

Contexto y problema

En una aplicación empresarial es necesario controlar y gestionar los procesos que realiza cada usuario a lo largo de múltiples peticiones.

Si no incluimos un mecanismo centralizado, entonces, el código de control se encuentra distribuido y frecuentemente duplicado en múltiples lugares.

Por ejemplo, la situación más común es que el código de control este embebido en múltiples vistas JSP. Cada vista se encarga de trabajar con los servicios del sistema lo que desemboca en una duplicación de código. Además, la vista se responsabiliza de gestionar el flujo de navegación lo que conlleva que el código que se encarga de presentar el contenido y el código de navegación estén mezclados. Hacernos una idea del workflow de navegación que tiene nuestra aplicación puede tornarse una tarea titánica ya que tendremos que buscarlo entre el código de múltiples vistas.



Esta práctica dificulta el mantenimiento ya que un cambio en la lógica de control requiere múltiples cambios en diferentes archivos. Además, la modularidad y mantenibilidad de la aplicación se ve afectada al estar el código de control mezclado con el código de las vistas.

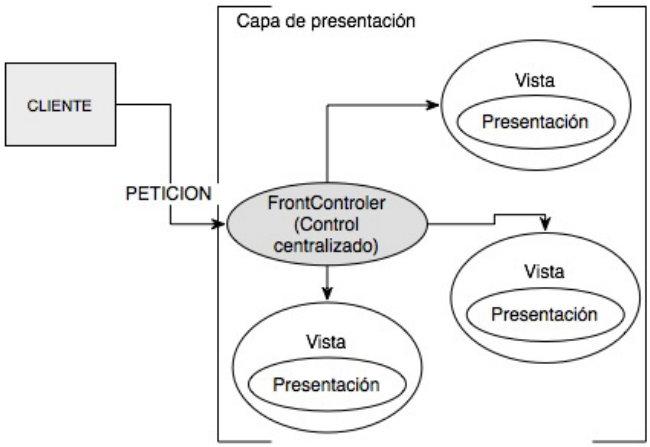
Necesidades y motivaciones

- ✓ Evitar lógica de control duplicada
- ✓ Aplicar lógica de control común a múltiples peticiones
- ✓ Separar la lógica de control de las vistas
- ✓ Tener los puntos de acceso a la aplicación centralizados y controlados

Solución

Un controlador frontal recibe todas las peticiones entrantes de los clientes.

Este patrón es similar al *Filtro de Intercepción* ya que se encarga de sacar fuera y de cohesionar el código de control. No obstante, *Intercepting Filter* proporciona, a diferencia de *Front Controller*, una conjunto de procesadores independientes entre sí y cuenta con estrategias específicas para las tareas de preprocesado y postprocesado de las



peticiones. Por su parte, los controladores frontales son los puntos ideales para implementar servicios de seguridad, tratamiento de errores, y la gestión del control para la generación de contenidos.

Está recomendado que *Front Controller* delegue en un *Application Controller* la gestión de las acciones y la gestión de la navegación. De esta forma conseguimos mantener simple el código del controlador frontal y hacer independientes del protocolo las tareas de gestión de vistas y acciones de tal manera que puedan ser reutilizadas por distintos clientes. *Application Controller* es un patrón de nueva aparición por lo que puede que encuentre documentación sobre *Front Controller* en la que este objeto no aparezca y las tareas delegadas las asuma el controlador.

Quizás sean necesarios varios *Front Controller* que se encarguen cada uno de mapear un conjunto de servicios diferente.

Estructura

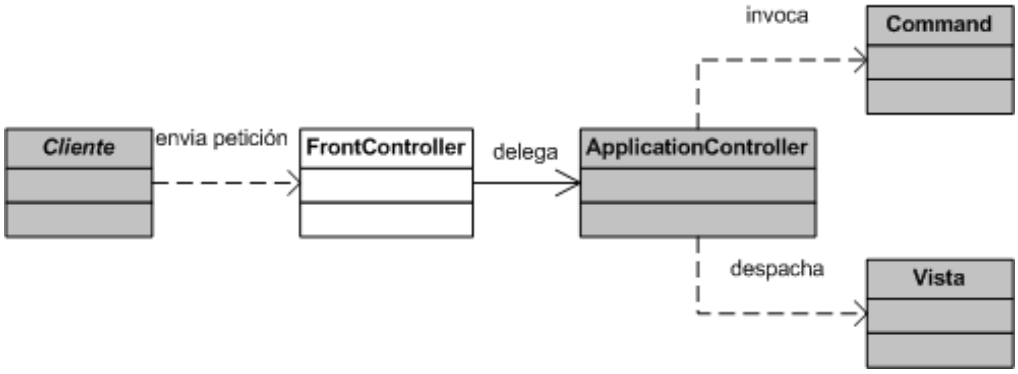


Diagrama de clases del patrón *Front Controller*

Participantes y responsabilidades

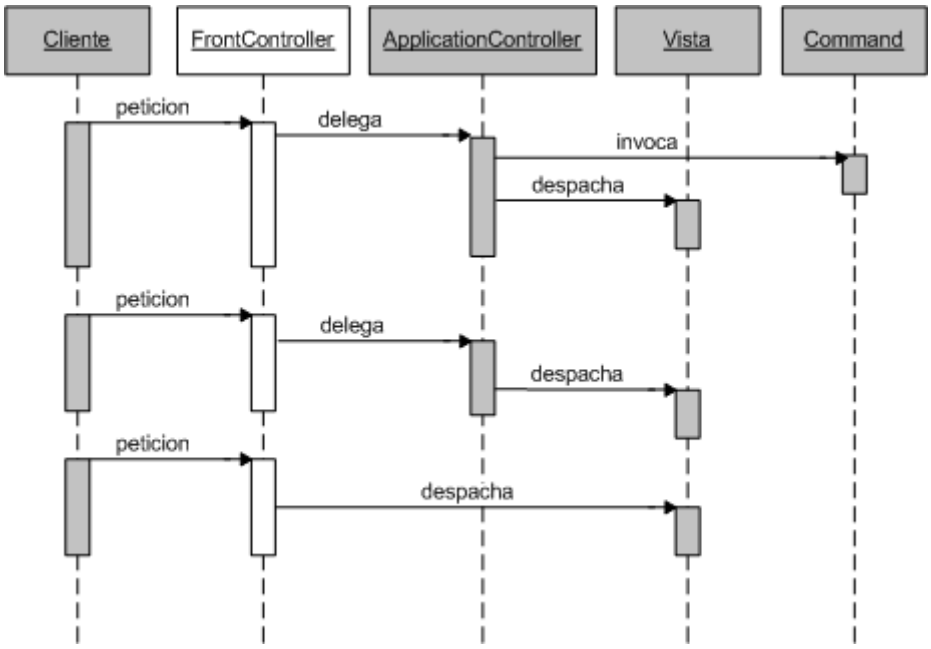


Diagrama de secuencia del patrón *Front Controller*

Cliente

Envía la petición de servicio

Front Controller

Componente que centraliza la lógica de control común a todas las peticiones. Se responsabiliza normalmente de encapsular la petición en un objeto de contexto. Las tareas de decisión del servicio de negocio que debe ser invocado y gestión de vistas se delegan en el *Application Controller*.

Application Controller

Se encarga de decidir qué acción debe atender la solicitud, localizarla e invocarla. Posteriormente y en función de la respuesta de la acción despachará la vista adecuada.

Command

Lleva a cabo la acción que solicitaba el cliente

Vista

Devuelve al cliente una representación del estado del modelo después de que se haya ejecutado la acción. La vista que se debe presentar es elegida por *Application Controller*

Estrategias

Estrategia Servlet para el controlador frontal

En esta estrategia el controlador es implementado con un servlet. Veamos un ejemplo

```
public class FrontController extends HttpServlet {
    // ... métodos de inicialización y destrucción del servlet
    // método para el procesado de peticiones (tanto GET como POST)
   protected void processRequest(HttpServletRequest request,
       HttpServletResponse response) throws ServletException,
       java.io.IOException {

     String page;
     /**ApplicationResources provee una API para acceder a costantes
      * y otros valores preconfigurados **/
     ApplicationResources resource = ApplicationResources.getInstance();
     try {
       // creamos el objeto de contexto para independizar del protocolo
       RequestContext requestContext = new RequestContext(request, response);

       // Invocamos el Application Controller para que se encargue de
       // manejar la petición (gestión de acciones y vistas)
      ApplicationController applicationController = new
           ApplicationControllerImpl();
       ResponseContext responseContext =
           applicationController.handleRequest(requestContext);
        applicationController.handleResponse(requestContext, responseContext);
     } catch (Exception e) {
       // Si se produce un error
       LogManager.logMessage("FrontController:exception : " +
           e.getMessage());
```

```
        request.setAttribute(resource.getMessageAttr(),
            "Exception occurred : " + e.getMessage());
        page = resource.getErrorPage(e);
        // transfiere el control a la vista de ERROR
        dispatch(request, response, page);
      }
    }
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        java.io.IOException {
      processRequest(request, response);
    }

  protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, java.io.IOException {

      processRequest(request, response);
    }
...
    protected void dispatch(HttpServletRequest request,
        HttpServletResponse response, String page)
        throws javax.servlet.ServletException, java.io.IOException {
      RequestDispatcher dispatcher =
this.getServletContext().getRequestDispatcher(page);
      dispatcher.forward(request, response);
    }
...
}
```

Para mapear todas las peticiones al controlador frontal deberemos configurarlo en el descriptor de despliegue. Debemos elegir una convención para ver qué peticiones mapeamos a cada controlador, en el caso de que tuvieramos varios.

Por ejemplo, si queremos mapear todas las peticiones que tengan la forma "accion.do" a un Front Controller implementado por el servlet UsuarioFrontController, entonces nuestro descriptor debería contener la siguiente información

Ejemplo mapeo de peticiones a un controlador – web.xml

```
...
<servlet>
    <servlet-name>UsuarioFrontController</servlet-name>
    <servlet-class>servlets.controllers.UsuarioFrontController</servlet-class>
</servlet>
...
<servlet-mapping>
  <servlet-name>UsuarioFrontController</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
...
```

Estrategia JSP para el controlador frontal

Aunque está documentada, esta estrategia está ampliamente desaconsejada debido a que los JSPs están diseñados para la presentación de contenidos y no para encapsular lógica de negocio. No obstante, semánticamente es equivalente a la estrategia anterior.

Consecuencias

- ✓ Centraliza el control
- ✓ Mejora la manejabilidad y seguridad de nuestra aplicación
- ✓ El código de control es reutilizable
- ✓ Aporta una mejor división de responsabilidades dentro de nuestra aplicación

Patrones J2EE relacionados

- ✓ Application Controller: Generalmente un *Front Controller* delegará la gestión de acciones, vistas y control de la navegación en un *Application Controller*.

Application Controller[1]

Contexto y problema

Cuando una petición llega a la capa de presentación la aplicación debe resolver qué acción debe gestionar dicha solicitud y qué vista se construirá y/o mostrará al usuario.

Aunque es una práctica extendida en aplicaciones sencillas, dar la responsabilidad de controlar el flujo de acciones y navegación a las vistas en aplicaciones empresariales es una mala práctica que reduce su reusabilidad y mantenibilidad. Sólo sería aceptable si la navegación fuera lineal, escenario infrecuente. El workflow de navegación de las aplicaciones empresariales es normalmente complejo, debe ser fácilmente identificable y modificable. Por tanto, atar las vistas a un determinado workflow no es una decisión acertada.

Las decisiones de qué acciones y vistas despachar para una petición podrían tomarse en el controlador frontal (vea el patrón *Front Controller*) de la aplicación pero esto suele desembocar en que el código de dicho controlador se vuelve demasiado complejo. Además, el controlador frontal es dependiente del protocolo y la gestión de acciones y vistas no debería serlo para facilitar su reutilización.

Necesidades y motivaciones

- ✓ Centralizar, modularizar y reutilizar la gestión de acciones y vistas

---

[1] *ApplicationController* es un patrón de reciente aparición. Puede encontrar referencias a él como *Request Processor*.

- ✓ Garantizar la extensibilidad de nuestro código de gestión de peticiones de tal forma que podamos añadir funcionalidad a la aplicación de manera incremental
- ✓ Facilitar el testeo del código de gestión de peticiones de forma independiente al contenedor web
- ✓ Controlar el orden en el que los usuarios visualizan las pantallas de nuestra aplicación

Solución

Usar el patrón *Application Controller* para centralizar la gestión e invocación de los componentes que manejan las peticiones. Esta gestión se hará de forma independiente del protocolo ya que trabajaremos con un objeto de contexto (*Context Object*) que encapsulará el estado de las peticiones.

Una aplicación puede tener múltiples *Application Controllers* para manejar diferentes partes. Un *Application Controllers* puede trabajar con varios clientes diferentes al ser independiente del protocolo. Por ejemplo, su aplicación podría tener un cliente web, un cliente para PDA, etc…

El *Application Controller* gestiona el flujo de navegación del usuario dependiendo del estado de ciertos objetos.

Estructura

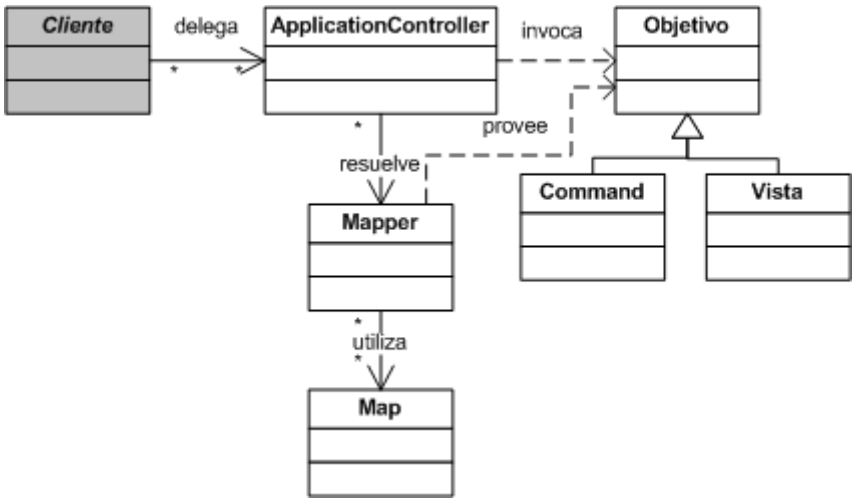


Diagrama de clases del Patrón *Application Controller*

Participantes y responsabilidades

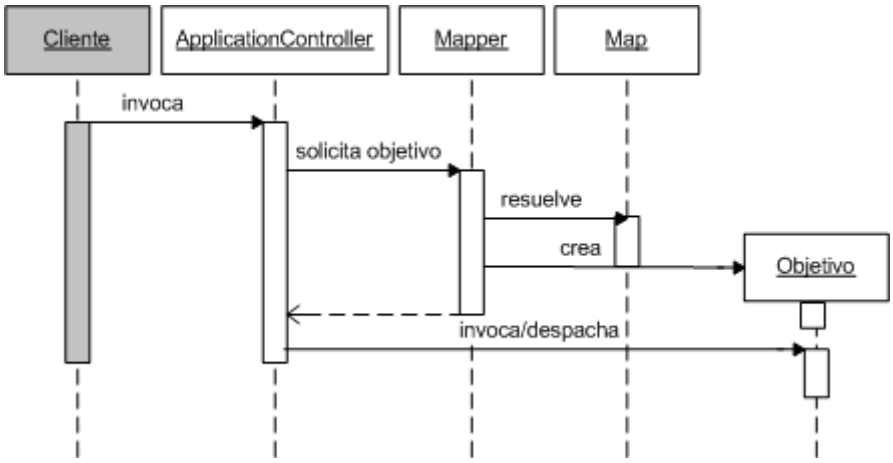


Diagrama de secuencia del patrón *Application Controller*

Cliente

Invoca al ApplicationController. Generalmente esta tarea la asume un *FrontController* o *InterceptingFilter*.

Mapper

Utiliza el Map para traducir la petición entrante a una acción y una vista.

Map

Relaciona una petición con la acción y la vista adecuadas.

Objetivo

El target que se encargará de procesar una petición particular.

Estrategias

Estrategia Command para el manejador de acciones

Se fundamente en la obtención e invocación de un objeto *Command* para atender la petición. Dicho objeto se obtiene por medio de una factoría y encapsula la lógica de negocio de un caso de uso específico. Los *Command* actuarán vía una interfaz común, el método *execute*. De esta forma se crea un manejador de acciones reutilizable y fácilmente extensible. Podremos añadir nueva funcionalidad y casos de uso a nuestra aplicación con sólo codificar nuevos objetos *Command*.

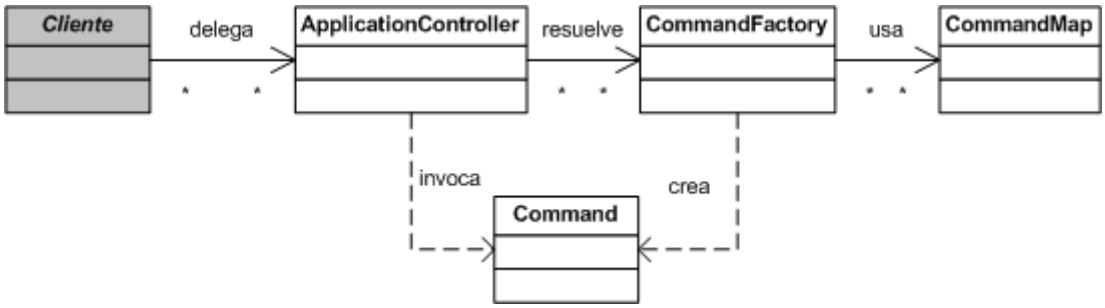


Diagrama de clases de la estrategia Command para el manejador de acciones

Ejemplo interfaz – ApplicationController.java

```
interface ApplicationController {
   void init();
   ResponseContext handleRequest(RequestContext requestContext);
   void  handleResponse(RequestContext requestContext, ResponseContext
responseContext);
   void destroy();
 }
```

Ejemplo Implementación de un ApplicationController WebApplicationController.java

```
// Application Controller responsable de manejar las peticiones web

 class WebApplicationController implements ApplicationController {

   public void init() { }

   public ResponseContext handleRequest(RequestContext requestContext) {
     ResponseContext responseContext = null;
     try {
       // Identificamos el nombre del comando (varias estrategias)
       String commandName = requestContext.getCommandName();

       // Obtenemos el objeto Command adecuado
       CommandFactory commandFactory = CommandFactory.getInstance();
       Command command = commandFactory.getCommand(commandName);

       // Invocamos el Command
       responseContext = command.execute(requestContext);
```

```
        // Manejamos excepciones

      } catch (java.lang.InstantiationException e) {
            // . . .
      } catch (java.lang.IllegalAccessException e) {
            // . . .
      }
      return responseContext;
    }
    . . .
}
```

Estrategia para el manejador de vistas

Para el manejo de las vistas, el Application Controller hará uso de la información devuelta por la acción ejecutada. En otras palabras, la vista que el usuario recibirá depende del resultado de la acción ejecutada para atender la petición.

La acción podría decidir directamente qué vista despachar (por ejemplo cle_formacion.jsp) o podemos introducir un grado de indirección para añadir aún más flexibilidad. En este último caso, la acción devolvería una referencia a la vista que queremos despachar (una vista lógica), por ejemplo la cadena "exito", y sería el Application Controller el encargado de mapear dicha referencia a una vista (física en este caso) con la información que le proporciona el Mapper, por ejemplo cle_formacion.jsp.

Esta última forma de proceder es la que implementa el framework Struts.

Estrategia de Transformación para el manejador

En esta estrategia, muy similar a la anterior, el Application Controller no obtiene una vista sino una hoja de transformación XSL que será procesada por un motor de transformación TransformHandler para terminar generando una vista.

Estrategias de control del flujo de navegación

Para controlar que el usuario va viendo las pantallas de nuestra aplicación en un orden determinado podemos utilizar varias estrategias

- ✓ Asociar precondiciones a las páginas para comprobar si se cumplen antes de mostrarsela al usuario. Ejemplo ¿está logueado el usuario? ¿ha enviado el pedido?
- ✓ Podemos modelar el flujo de navegación con una máquina de estados finita. Es una solución potente pero que requiere mucho trabajo de codificación
- ✓ Para evitar peticiones duplicadas podemos añadir tokens de sincronización (patrón Synchronizer Token).

El framework Struts implementa esta última estrategia haciendo uso de tokens. Cuando un Command es ejecutado se genera un token único para el usuario que depende de la hora y el ID de la sesión. Este token es almacenado en la sesión del usuario. Posteriormente, la vista incluirá este token en un campo hidden que será enviado junto a la petición. Cuando el controlador reciba dicha petición deberá comprobar si el token es válido comparándolo con el que tiene almacenado el usuario en la sesión. Si es válido, la petición es atendida y el token de la sesión se pone a null. De esta forma, aunque el usuario reenvíe el formulario no se atenderá varias veces.

A continuación se muestran las funciones de Struts que hacen las operaciones de guardar un token y de validar el token.

```
protected void saveToken(HttpServletRequest request) {

HttpSession session = request.getSession();
String token = generateToken(request);
if (token != null)
  session.setAttribute(TRANSACTION_TOKEN_KEY, token);
}
```

```
protected boolean isTokenValid(HttpServletRequest request) {

    // Recupera el TOKEN guardado en la sesión
    HttpSession session = request.getSession(false);
    if (session == null)
        return (false);
    String saved = (String)
        session.getAttribute(TRANSACTION_TOKEN_KEY);
    if (saved == null)
        return (false);
    // recupera el TOKEN de la petición

    String token = (String)
        request.getParameter(Constants.TOKEN_KEY);
    if (token == null)
        return (false);

    // ¿Son iguales ambos tokens?
```

```
    return (saved.equals(token));

}
```

Consecuencias

- ✓ Mejora la modularidad y división de responsabilidades
- ✓ Facilita la reutilización
- ✓ Facilita el testeo del código de gestión de acciones y vistas
- ✓ Crea una arquitectura extensible en la que podemos añadir nuevas funcionalidades de forma fácil

Patrones J2EE relacionados

- ✓ *Front Controller*: El controlador frontal delega en Application Controller el manejo de acciones y vistas.

Service to Worker

Contexto y problema

El problema es la unión de varios de los problemas solventados con anterioridad por los patrones *Front Controller*, *Application Controller* y *View Helper*.

Tenemos una aplicación en la que el código de control está disperso en múltiples vistas. El código de la lógica de negocio y el código de formateo de la presentación se encuentran mezclados, la gestión del flujo de navegación es complicada y depende del estado de ciertos objetos.

Necesidades y motivaciones

- ✓ Tenemos lógica de negocio que será ejecutada como respuesta a una petición y que queremos encapsular en componentes reutilizables.
- ✓ La lógica de negocio se encargará de recuperar el contenido que será utilizado para generar respuestas dinámicamente
- ✓ El control del flujo de navegación es complicado. Para una misma petición tenemos múltiples vistas que podrían ser servidas. La vista seleccionada dependerá de la respuesta que la capa de negocio proporcione a la petición.
- ✓ Queremos definir claramente los roles de los miembros de nuestro equipo de trabajo

Solución

Combinar las soluciones proporcionadas por los patrones *Front Controller*, *Application Controller* y *View Helper*. El *Front Controller* se responsabilizará de centralizar el control de las peticiones (autenticación, checkeos de privilegios, …) y delegará en el *Application*

Controller la gestión de las acciones y de vistas. La lógica de negocio se ejecutará antes de que las vistas tomen el control. Los *View Helpers* se encargarán de almacenar y adaptar el modelo intermedio que necesitan las vistas necesitan para generar una respuesta dinámica.

Estructura

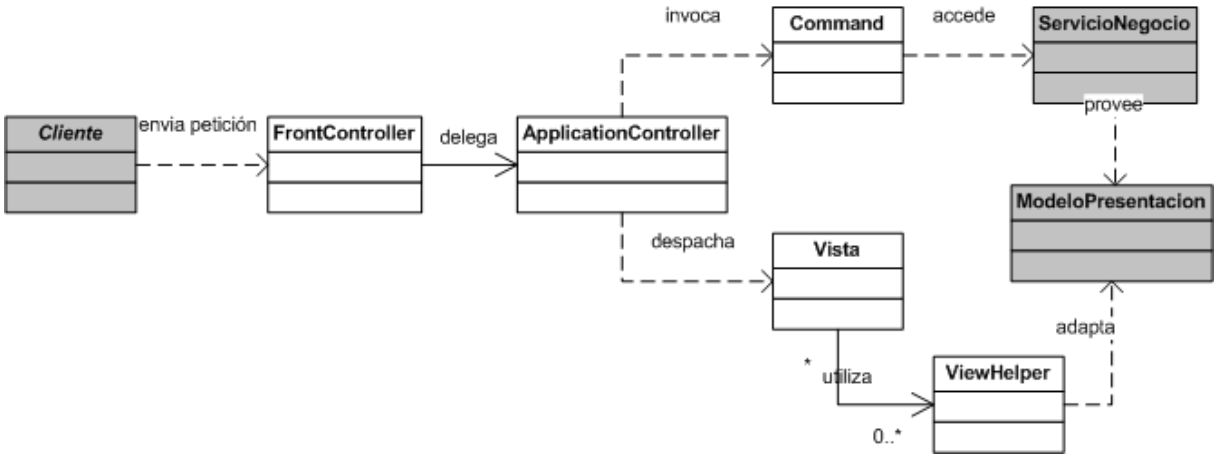


Diagrama de clases del patrón Service to Worker

Participantes y responsabilidades

FrontController

Inicia el manejo de la petición y delega en el *ApplicationController*

ApplicationController

Responde ante la gestión de acciones y vistas

ModeloPresentacion

Contiene los datos del modelo que va a mostrar la vista y que han sido proveidos por el servicio de negocio

ServicioNegocio

Encapsula la lógica de negocio y el estado. Puede ser accedido mediante *Business Delegate* si se trata de un servicio remoto

Patrones de la capa de negocio

Business Delegate[2]

Contexto y problema

Si los clientes remotos interactúan directamente con los servicios de negocio entonces aparecen un buen número de problemas que pasaremos a describir.

En primer lugar, la interacción directa con la interfaz de los servicios crea acoplamiento entre cliente-servicios, lo que disminuye la mantenibilidad de nuestra aplicación.

Por otro lado, el rendimiento de nuestra red puede ser pobre debido a que si los clientes remotos interactuan directamente con la API de servicios, una simple acción puede conllevar múltiples interacciones de menor granularidad con los servicios de negocio.

Por último, la interacción directa conlleva a que el cliente deba utilizar servicios de nombres (como JNDI o UDDI), manejar los errores de la red y otras tareas que incrementan su complejidad.

Necesidades y motivaciones

- ✓ Queremos acceder a los componentes de la capa de negocio minimizando el acoplamiento con ellos
- ✓ Queremos evitar la invocación innecesaria de servicios remotos

---

[2] Aunque este patrón esté catalogado como un patrón de la capa de negocio ello no quiere decir que no pueda residir físicamente en otra capa. De hecho, así será cuando lo utilizemos, por ejemplo, con un cliente de la capa de presentación.

- ✓ Queremos traducir los errores producidos por la red a errores de la aplicación o excepciones de usuario
- ✓ Queremos evitarle complejidades innecesarias a los clientes

Solución

Usamos un *Business Delegate* para encapsular los accesos a los servicios de negocio. De esta forma ocultaremos la implementación de los mecanismos de búsqueda, creación e invocación de servicios. Generalmente, los mecanismos de búsqueda son delegados en un *Service Locator*.

Business Delegate maneja las excepciones que produzcan los servicios de negocio, por ejemplo, las excepciones `java.rmi.Remote`, las excepciones JMS u otras. Dichas excepciones serán transformadas a excepciones más manejables por los clientes, es decir, excepciones a nivel de aplicación y no de servicios.

Un delegado puede cachear los resultados y referencias de servicios remotos para mejorar el rendimiento de nuestra aplicación y de la red.

Estructura

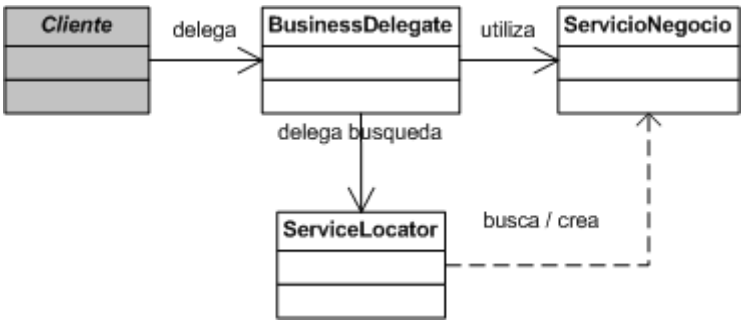


Diagrama de clases del patrón *Business Delegate*

Participantes y responsabilidades

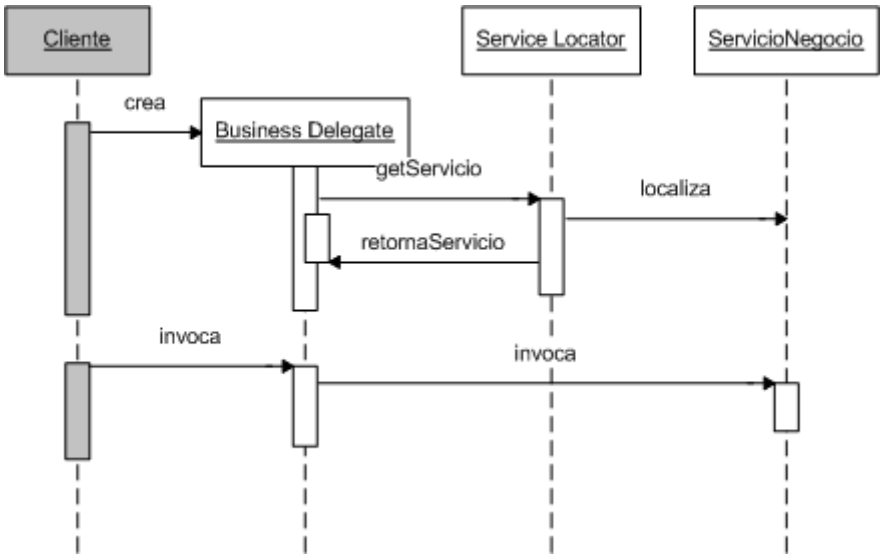


Diagrama de secuencia del patrón *Business Delegate*

Consecuencias

- ✓ Reduce el acoplamiento y mejora la mantenibilidad
- ✓ Traduce las excepciones del nivel servicio a nivel de aplicación
- ✓ Mejora la disponibilidad ya que puede implementar mecanismo de recuperación ante fallos de infraestructura (fallos de red por ejemplo)

- ✓ Mejora el rendimiento gracias al cacheado
- ✓ Encapsula el carácter remoto de los servicios a los clientes

Patrones J2EE relacionados

- ✓ *Service Locator* es utilizado por *Business Delegate* para la localización de servicios
- ✓ *Business Delegate* suele mantener una correspondencia de uno a uno con *Session Façade* que interpreta el rol de ServicioNegocio. Cada vez que un desarrollador implementa una fachada de servicio suele proveernos también de su delegado correspondiente.

Service Locator

Contexto y problema

Las aplicaciones empresariales J2EE están fuertemente distribuidas y para llevar a cabo una funcionalidad tendremos que localizar servicios o recursos e invocarlos. Por ejemplo, podríamos tener que buscar componentes de negocio EJBs, componentes Java Message Service, Web Services, conexiones JDBC, etc…

La localización e invocación de servicios y recursos heterogéneos suele ser una tarea compleja compuesta por las siguientes etapas

- ✓ Crear un objeto de contexto inicial `InitialContext`. Este objeto es vendor-specific
- ✓ Utilizar un registro de servicios (JNDI, UDDI, …) para localizar el servicio en el registro de servicios
- ✓ El registro de servicio suele contener un objeto administrado que actúa de factoría de los objetos de servicio que necesitamos y que es devuelto al cliente
- ✓ El cliente debe trabajar con la factoría de los objetos de servicio para buscar o crear una instancia del objeto de servicio

Por ejemplo, para usar un EJB deberemos establecer el objeto de contexto inicial InitialContext adecuado, crear una conexión con un servicio de nombres JNDI, localizar el EJB Home (que actúa como factoría del objeto de servicio que sería el EJB) y por último buscar una instancia del bean que esté disponible o crearla.

En el caso de un componente JMS, el cliente deberá crear el objeto de contexto inicial InitialContext adecuado, crear una conexión con un servicio de nombres JNDI, localizar la factoría de conexiones JMS y posteriormente obtener una conexión o sesión JMS.

Las operaciones de creación del contexto inicial y la búsqueda de servicios en los registros de servicios son complejas y costosas y su repetición en múltiples clientes perjudica el rendimiento y la mantenibilidad de nuestra aplicación.

A continuación se muestra un ejemplo de creación de contexto inicial

```
public static InitialContext getInitialContext() {
    Properties env = new Properties();
    env.put(Context.SECURITY_PRINCIPAL, "invitado");
    env.put(Context.SECURITY_CREDENTIALS, "invitado");
    env.put(Context.INITIAL_CONTEXT_FACTORY,
       "weblogic.jndi.WLInitialContextFactory");
    env.put(Context.PROVIDER_URL, "t3://localhost:7001");
    return new InitialContext(env);
}
```

Solución

Creamos un nuevo objeto *Service Locator* que se responsabilize de la localización y obtención de todos los servicios y recursos que pueda necesitar nuestra aplicación sean del tipo que sean. De esta forma ocultamos la complejidad de las operaciones que ello conlleva, evitamos duplicar código, creamos un único punto de localización de servicios, eliminamos la dependencia de nuestros componentes con las especificidades de cada proveedor de servicios y mejoramos el rendimiento de nuestra aplicación con técnicas de cacheado.

El objeto Service Locator es implementado como un Singleton [GoF], es decir, un objeto del que sólo puede existir una instancia.

Estructura

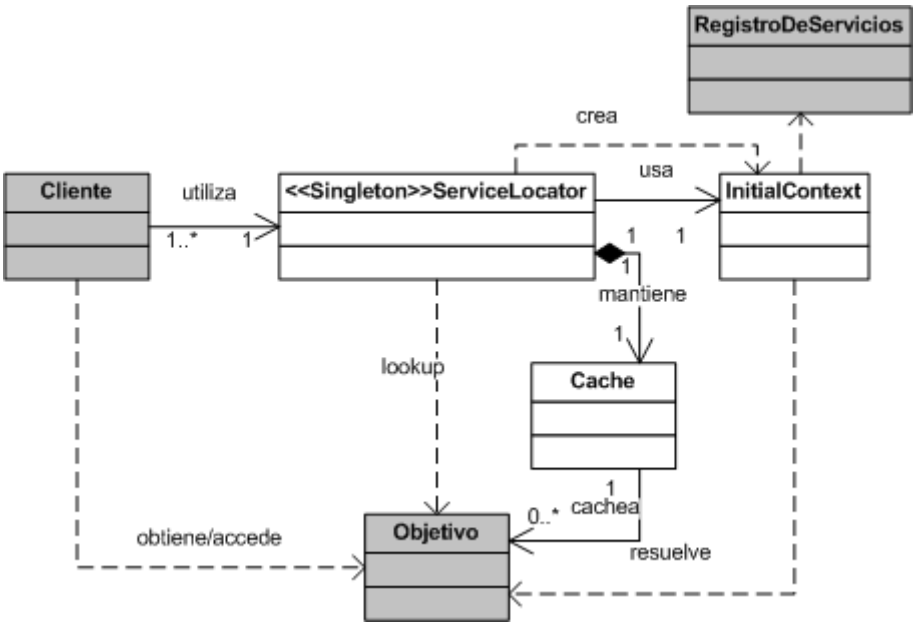


Diagrama de clases del patrón *Service Locator*

Participantes y responsabilidades

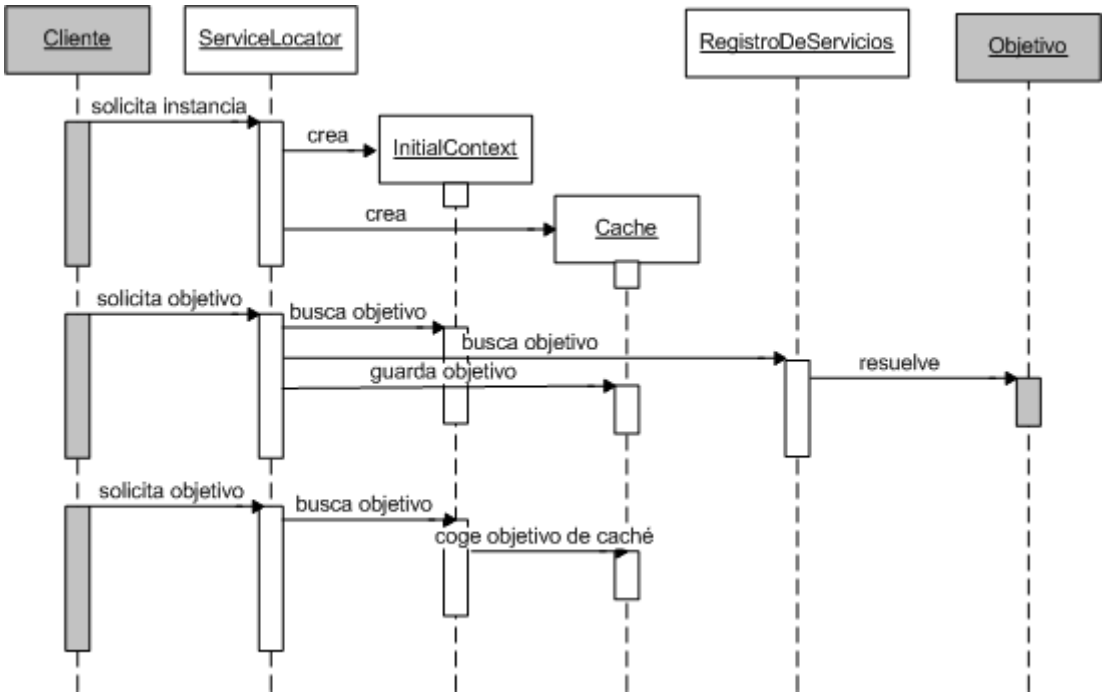


Diagrama de secuencia del patrón *Service Locator*

En el diagrama de secuencia se muestran dos posibles escenarios: una búsqueda de un objetivo que no está cacheado y posteriormente la búsqueda del mismo objetivo. En este segundo caso la lectura se hace de caché.

Cliente

Representa un componente que quiere acceder a un servicio o componente de la capa de negocio o de integración. Posibles clientes son los objetos DAO que buscarán instancias de fuentes de datos JDBC o *Business Delegate* que deben localizar y acceder a EJB que ejerzan de *Session Façades*

Objetivo

El servicio o componente de la capa de negocio o de integración que el cliente requiere localizar

ServiceLocator

Encapsula las tareas de localización de servicios y recursos. Oculta la complejidad a los clientes y facilita la reutilización de código y recursos

InitialContext

Es el punto de partida en un proceso de búsqueda y creación de objetivos. Todos los nombres de objetivos que nos proporcione el registro de servicios será un nombre relativo a nuestro contexto inicial. Este objeto es vendor-specific.

Cache

Elemento opcional que guarda referencias a objetivos previamente localizados

RegistroDeServicios

Representa la implementación del registro de servicios concreta que guarda las referencias a los servicios y componentes registrados por los proveedores del cliente. Este elemento podría ser un JNDI, un registro UDDI (para servicios web), etc…

Estrategias

A continuación se muestra una estrategia para localizar componentes EJB y otra para localizar servicios JMS. Para cada tipo de recurso tendrá que seguir una estrategia distinta. Todas ellas se deben implementar en una única clase *ServiceLocator*.

Ejemplo *ServiceLocator* para EJB – ServiceLocator.java

```
// importaciones
public class ServiceLocator {

    private InitialContext ic; // sólo definiremos una vez el contexto inicial
    private Map cache; // caché para no tener que repetir las búsquedas de
                       // recursos en el JNDI
    private static ServiceLocator me; // única instancia de la clase
    // creación de la instancia
    static {
      try {
        me = new ServiceLocator();
      } catch(ServiceLocatorException se) {
        System.err.println(se);
        se.printStackTrace(System.err);
      }
    }

///// MÉTODOS PARA LA LOCALIZACION DE COMPONENTES EJB ////
```

```
   // constructor PRIVADO. Para obtener una instancia de la clase
   // debemos llamar al método getInstance()
   private ServiceLocator() throws ServiceLocatorException  {
      try {
        ic = new InitialContext();
        cache = Collections.synchronizedMap(new HashMap());
      } catch (NamingException ne) {
           throw new ServiceLocatorException(ne);
      } catch (Exception e) {
           throw new ServiceLocatorException(e);
       }
    }

    static public ServiceLocator getInstance() {
      return me;
    }

    /* Método que devuelve el EJBHome local */
    public EJBLocalHome getLocalHome(String jndiHomeName) throws
ServiceLocatorException {
      EJBLocalHome home = null;
      try {
        if (cache.containsKey(jndiHomeName)) {
           home = (EJBLocalHome) cache.get(jndiHomeName);
        } else {
           home = (EJBLocalHome) ic.lookup(jndiHomeName);
           cache.put(jndiHomeName, home);
        }
       } catch (NamingException ne) {
           throw new ServiceLocatorException(ne);
       } catch (Exception e) {
```

```
            throw new ServiceLocatorException(e);
        }
        return home;
    }


    public EJBHome getRemoteHome(String jndiHomeName, Class className) throws
ServiceLocatorException {
      EJBHome home = null;
      try {
        if (cache.containsKey(jndiHomeName)) {
            home = (EJBHome) cache.get(jndiHomeName);
        } else {
            Object objref = ic.lookup(jndiHomeName);
            Object obj = PortableRemoteObject.narrow(objref, className);
            home = (EJBHome)obj;
            cache.put(jndiHomeName, home);
        }
       } catch (NamingException ne) {
            throw new ServiceLocatorException(ne);
       } catch (Exception e) {
            throw new ServiceLocatorException(e);
       }

       return home;
    }

//// MÉTODOS PARA LA LOCALIZACION DE COMPONENTES JMS /////

    public  QueueConnectionFactory getQueueConnectionFactory(String
qConnFactoryName)
```

```
                                                    throws
ServiceLocatorException {
      QueueConnectionFactory factory = null;
      try {
        if (cache.containsKey(qConnFactoryName)) {
            factory = (QueueConnectionFactory) cache.get(qConnFactoryName);
        } else {
            factory = (QueueConnectionFactory) ic.lookup(qConnFactoryName);
            cache.put(qConnFactoryName, factory);
        }
      } catch (NamingException ne) {
        throw new ServiceLocatorException(ne);
      } catch (Exception e) {
            throw new ServiceLocatorException(e);
      }
      return factory;
    }

    public  Queue getQueue(String queueName) throws ServiceLocatorException {
      Queue queue = null;
      try {
        if (cache.containsKey(queueName)) {
            queue = (Queue) cache.get(queueName);
        } else {
            queue =(Queue)ic.lookup(queueName);
            cache.put(queueName, queue);
        }
      } catch (NamingException ne) {
            throw new ServiceLocatorException(ne);
      } catch (Exception e) {
            throw new ServiceLocatorException(e);
      }
```

```
      return queue;
    }

    public  TopicConnectionFactory getTopicConnectionFactory(String
topicConnFactoryName) throws ServiceLocatorException {
      TopicConnectionFactory factory = null;
      try {
        if (cache.containsKey(topicConnFactoryName)) {
            factory = (TopicConnectionFactory)
cache.get(topicConnFactoryName);
        } else {
            factory = (TopicConnectionFactory)
ic.lookup(topicConnFactoryName);
            cache.put(topicConnFactoryName, factory);
        }
      } catch (NamingException ne) {
         throw new ServiceLocatorException(ne);
      } catch (Exception e) {
            throw new ServiceLocatorException(e);
      }
      return factory;
    }


    public  Topic getTopic(String topicName) throws ServiceLocatorException {
      Topic topic = null;
      try {
        if (cache.containsKey(topicName)) {
            topic = (Topic) cache.get(topicName);
        } else {
            topic = (Topic)ic.lookup(topicName);
```

```
            cache.put(topicName, topic);
          }
        } catch (NamingException ne) {
           throw new ServiceLocatorException(ne);
        } catch (Exception e) {
            throw new ServiceLocatorException(e);
        }
        return topic;
    }
    // más funcionalidades de localización aquí

}
```

Consecuencias

- ✓ Abstrae la complejidad
- ✓ Provee una forma uniforme de localización y acceso a servicios y componentes heterogéneos
- ✓ Descongestiona la red y mejora el rendimiento

Patrones relacionados

- ✓ Business Delegate utiliza los servicios de este patrón para localizar componentes de negocio distribuidos
- ✓ Session Façade, Transfer Object Assembler y Data Acces Object son otros de los clientes frecuentes que utilizan ServiceLocator.

Session Façade

Contexto y problema

Queremos exponer nuestros servicios de negocio a clientes remotos. No obstante, si nuestros objetos de negocio empresariales implementados con EJB de Entidad, POJOs son accedidos directamente por los clientes vía sus respectivas interfaces, entonces, aparecen los siguientes problemas

- ✓ Los clientes son dependientes de los objetos de negocio, lo que significa que cualquier cambio en uno de los objetos supondrá un cambio potencial en todos los clientes que lo utilizan
- ✓ Para llevar a cabo un proceso de negocio los clientes deben trabajar y coordinar todos los objetos de negocio que intervengan en él, lo que implica conocer los mismos a fondo. Por tanto, la complejidad y responsabilidades del cliente se ven aumentadas
- ✓ Cuando tenemos distintos tipos de clientes que acceden a los componentes de negocio, entonces, cada uno de ellos deberá contener el código que se encarga de la interacción con los mismos. Esto empeora considerablemente la mantenibilidad y flexibilidad de nuestra aplicación
- ✓ El número de peticiones remotas que hacen los clientes es elevado ya que deberán comunicarse con múltiples objetos de negocio remotos. El número de peticiones remotas aumenta de forma directamente proporcional al número de objetos que participen en el proceso de negocio que queramos llevar a cabo. Esto aumenta considerablemente el tráfico de la red

- ✓ Al exponer directamente la interfaz de los objetos de negocio cada cliente puede implementar una estrategia distinta para acceder a ellos y esto puede dar lugar a usos erróneos de los mismos

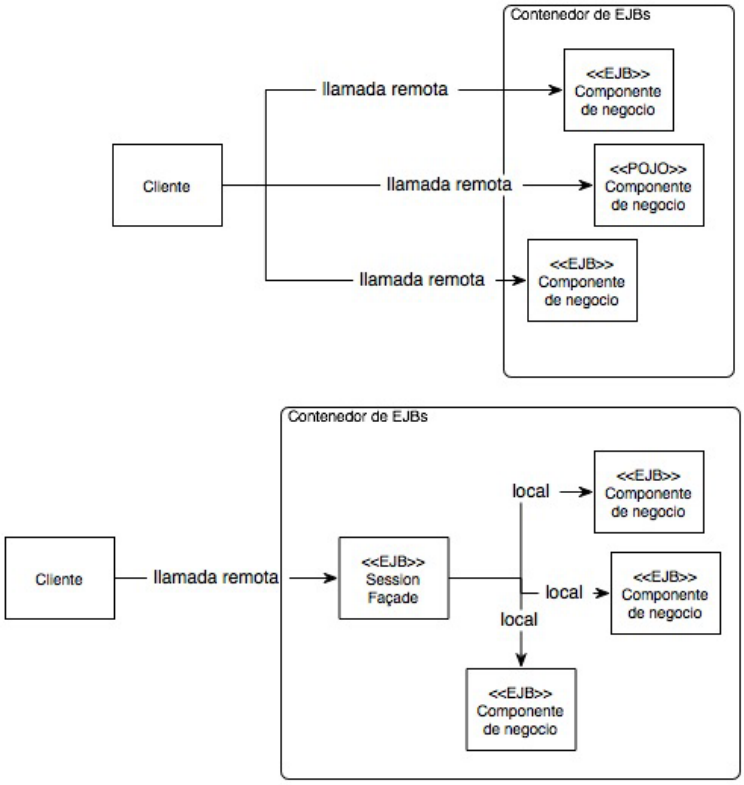
Solución

Usar un objeto que actúe a modo de fachada denominado *Session Façade* que encapsule los objetos de negocio y exponga los servicios que requieren los clientes remotos. Dicho objeto estará diseñado para cumplir dos objetivos fundamentales: controlar el acceso a los objetos de negocio y limitar el tráfico de red.

Con la introducción de una fachada la granularidad de los accesos a la capa de negocio se ve aumentada y evitamos complejidades innecesarias a los clientes ya que estos no trabajarán directamente con los objetos de negocio sino con los servicios que *Session Façade* les ofrece. Esto redunda en una disminución drástica del número de peticiones remotas ya que el acceso y coordinación de los objetos de negocio será responsabilidad de la fachada. Por tanto, convertimos llamadas remotas en locales. El número de llamadas remotas se hace independiente del número de objetos remotos involucrados en un servicio.

Además añadimos uniformidad a la hora de acceder a los objetos de negocio ya que exponemos una interfaz simple a los clientes que evitará complejas interacciones con los mismos.

Session Façades trabaja mejor cuando no contiene lógica de negocio asociada o esta es muy simple. Si la interacción con los objetos de negocio llevara asociada una lógica de negocio compleja, entonces es conveniente usar el patrón *Application Service.*



Reducción de llamadas remotas que produce la aplicación de *Session Façade*

Estructura

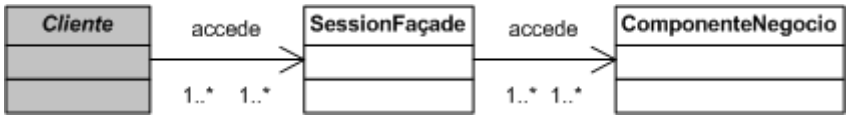


Diagrama de clases del patrón *Session Façade*

Participantes y responsabilidades

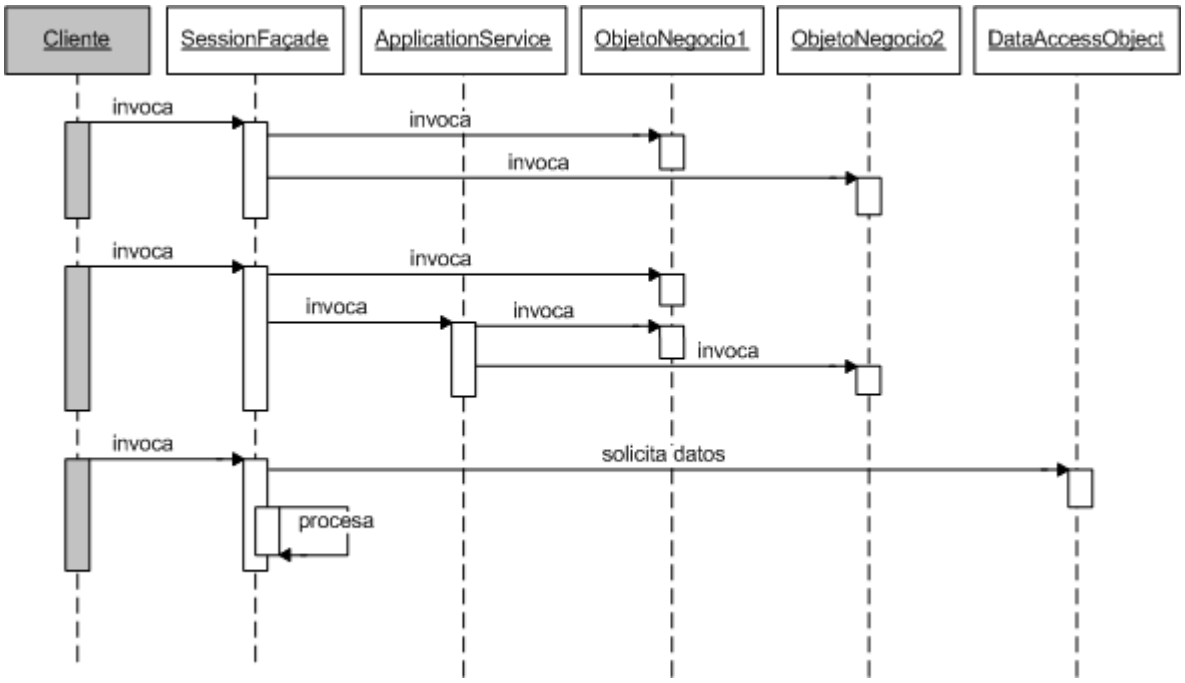


Diagrama de secuencia del patrón *SessionFaçace*

En el diagrama de secuencia se muestran varios los posibles comportamientos que podría tener una fachada a la hora de resolver una solicitud de servicio por parte del cliente.

Estrategias

La estrategia que escojamos para implementar *Session Façade* dependerá de los procesos de negocio que esta modele.

Stateless Session Façade Strategy

Usaremos un EJB de sesión sin estado (stateless) para codificar los procesos de negocio no conversacionales, esto es, cuando nuestra *Session Façade* no requiera guardar información entre distintas llamadas. El cliente hará una sola invocación a un método de la fachada para llevar a cabo un caso de uso. Cuando el método termina, el caso de uso también lo hace y no es necesario guardar estados entre invocaciones a métodos.

Statefull Session Façade Strategy

Cuando un proceso de negocio necesita realizar múltiples llamadas a métodos de la fachada para completar un servicio, entonces, necesitaremos guardar el estado entre las sucesivas peticiones del cliente. Estos procesos de negocio, conocidos como conversacionales, requerirán una fachada con estado que deberá ser implementada con un EJB de sesión statefull.

Consecuencias

- ✓ Introduce una capa de servicios para los clientes remotos de tal forma que estos no tienen que conocer la complejidad de la capa de negocio ni acoplarse a ella
- ✓ La fachada puede beneficiarse de los servicios que ofrece el contenedor de EJB para control de transacciones, control de seguridad, etc… Dichos controles se

deben centralizar en la fachada para garantizar la seguridad y la integridad de los datos.

- ✓ Aumenta la flexibilidad y manejabilidad de la aplicación al introducir una nueva capa y centralizar las interacciones con el sistema
- ✓ Mejora el rendimiento al evitar accesos de fina granularidad

Patrones relacionados

- ✓ Service Locator: Generalmente es usado por la fachada para localizar los componentes de negocio
- ✓ Data Acces Object: En los casos en los que la lógica de negocio sea trivial, la fachada puede interactuar directamente con un objeto de acceso a datos

Application Service[3]

Contexto y problema

Cuando la coordinación de los objetos de negocio que implementan un caso de uso expuesto por una fachada es compleja y requiere lógica no trivial, entonces, es mejor no incluirla en la fachada. Esto es así porque de lo contrario podríamos tener duplicaciones de código en múltiples fachadas.

Necesidades y motivaciones

- ✓ Queremos reducir al máximo la lógica de negocio que contienen las fachadas a los servicios
- ✓ Queremos proveer una API de granularidad gruesa a los componentes y servicios de la capa de negocio
- ✓ Encapsularemos la lógica de casos de uso específicos fuera de los objetos de negocio

Consecuencias

- ✓ Mejoramos la reusabilidad de la lógica de negocio
- ✓ Eliminamos la duplicación de código en fachadas y ayudantes
- ✓ Simplificamos la implementación de las fachadas.

---

[3] Este patrón ha aparecido recientemente al igual que *Application Controller*. Encontrará documentación en la que no aparezca y sus funciones las ejerza directamente *Session Façade*

- ✓ Composite Entity

Contexto y problema

Decidir cómo mapear nuestro modelo de objetos en un modelo de Enterprise JavaBeans es un problema recurrente a la hora de desarrollar aplicaciones J2EE y que este patrón de diseño aborda.

Existen dos tipos de objetos según su granularidad. Por objetos de granularidad gruesa u objetos padres entendemos aquellos que son reutilizables, independientes, manejan su ciclo de vida y sus relaciones con otros objetos. Por el contrario, existen objetos de granularidad fina u objetos dependientes que son poseídos por uno o más objetos padres, no existen por sí solos y su ciclo de vida es manejado por el padre.

En especificaciones anteriores a la EJB 2.0 los EJBs de entidad sólo podían accederse de manera remota lo que causaba grandes problemas de red si la granularidad de los objetos que eran modelados como EJB de entidad era fina (una situación parecida a la que abordamos en el patrón *Transfer Objects*).

A partir de la especificación EJB 2.0 los EJBs de entidad pueden accederse de forma local, lo que ha resuelto los problemas de red que citabamos anteriormente. No obstante, este patrón aún sigue siendo de aplicación ya que tendremos que decidir qué objetos implementar como EJB de entidad remotos, cómo implementar la persistencia de nuestros objetos en beans de entidad y cómo manejar las relaciones con otros objetos.

Necesidades y motivaciones

- ✓ Queremos evitar las desventajas de los beans de entidad remotos
  - × Sobrecarga de la red
  - × Relaciones inter-entidades remotas muy costosas y complejas

- ✓ Queremos implementar de forma eficiente en nuestros beans de entidad las relaciones padre-hijo que se dan entre objetos
- ✓ Queremos encapsular el esquema de la base de datos a los clientes

Solución

Usar un *CompositeEntity* para generar un agregado de objetos de negocio en un Entity Bean de granularidad gruesa.

Los objetos hijos serán implementados mediantes POJOs o EJBs de entidad locales (si trabajamos con una especificación EJB superior a la 2.0) para evitar la degradación del rendimiento que conllevaría hacerlo con EJBs de entidad remotos.

Podemos utilizar las dos estrategias de persistencia, CMP (gestionada por el contenedor) o BMP (gestionada por el Bean). Para gestionar las relaciones con los objetos hijos podemos utilizar la estrategia gestionada por el Bean o, por el contrario, podemos optar por CMR si usamos CMP y nuestros objetos hijos han sido implementados como EJBs de entidad locales.

Estructura

Diagrama de clases del patrón *Composite Entity*

Participantes y responsabilidades

CompositeEntity

Representa el objeto padre (de granularidad gruesa). Se implementa con un EJB de entidad.

ObjetoNegocioDependiente

Representa los objetos hijos (de granularidad fina) implementados mediante POJOs o EJB de entidad locales.

Consecuencias

- ✓ Incrementa la mantenibilidad
- ✓ Reduce el número de EJBs de entidad e incrementa la granularidad de los objetos
- ✓ Reduce la dependencia con el esquema de nuestra Base de Datos
- ✓ Facilita la creación de *Composite Transfer Object*

Patrones J2EE relacionados

- ✓ Este patrón crea un *Composite Transfer Object* que será devuelto al cliente
- ✓ *Session Façade* suele servir para ocultar este componente a los clientes de la aplicación y que los accesos se hagan de un modo indirecto
- ✓ *Transfer Object Assembler* tiene un comportamiento similar en lo referente a la creación de *Composite Transfer Object* aunque sus fuentes de datos son más amplias y heterogeneas que las de *Composite Entity*

Transfer Object[4]

Contexto y problema

Las aplicaciones empresariales poseen componentes de negocio que devuelven datos a los clientes. Usualmente dichos componentes son implementados como objetos remotos mediante beans de sesión o beans de entidad. Éstos poseen métodos de acceso a los atributos (get/set) de granularidad muy fina. Un cliente podría realizar múltiples peticiones a métodos de este tipo para obtener toda la información que necesita del bean. Cuando las peticiones se hacen de forma remota, esta práctica genera una sobrecarga innecesaria en la red. Hacer múltiples llamadas remotas para acceder al conjunto de la información es extremadamente ineficiente y puede conllevar a problemas graves de rendimiento y congestión de la red.

Necesidades y motivaciones

- ✓ Poseemos componentes que son accedidos desde otras capas para recabar y actualizar datos
- ✓ Queremos reducir las llamadas remotas
- ✓ Queremos reducir el tráfico de la red
- ✓ Queremos evitar la degradación del rendimiento de la red causado por las múltiples llamadas remotas que realizan nuestras aplicaciones empresariales

---

[4] *Value Object* y *Data Transfer Objects* son otras denominaciones con las que encontraremos este patrón en la bibliografía

Solución

Los *Transfer Object* están diseñados para optimizar el paso de datos entre capas.

La solución pasa por encapsular todos los datos de negocio en un objeto de transferencia que recuperaremos con una única llamada remota entre capas. El resto de las llamadas necesarias para obtener cada una de las propiedades de interés las haremos de forma local, accediendo al *Transfer Object*. Este objeto es construido por el componente y devuelto por valor al cliente. Esto supone que habrá una copia local y otra remota.

El cliente puede crear sus propias instancias de Transfer Object y enviarlas al componente para llevar a cabo actualizaciones de los datos.

Los objetos de transferencia deben ser serializables, por tanto, implementarán la interfaz *java.io.Serializable*.

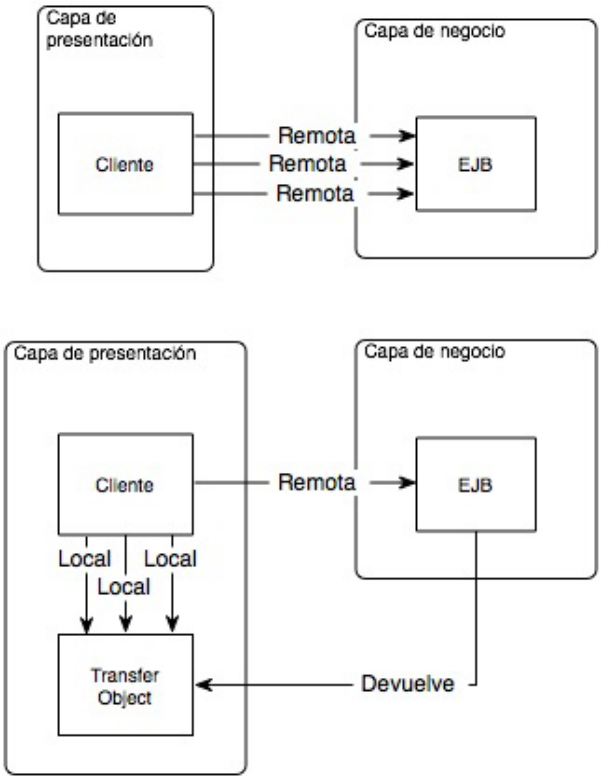


Figura que muestra la reducción drástica del número de llamadas remotas al aplicar el patrón *Transfer Object*

Estructura

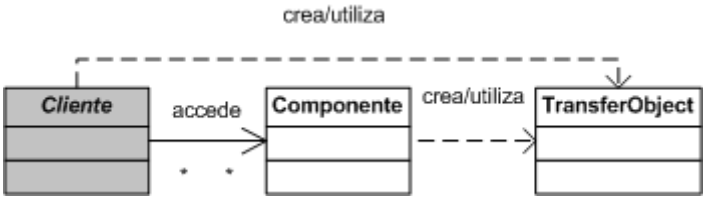


Diagrama de clases del patrón *Transfer Object*

Participantes y responsabilidades

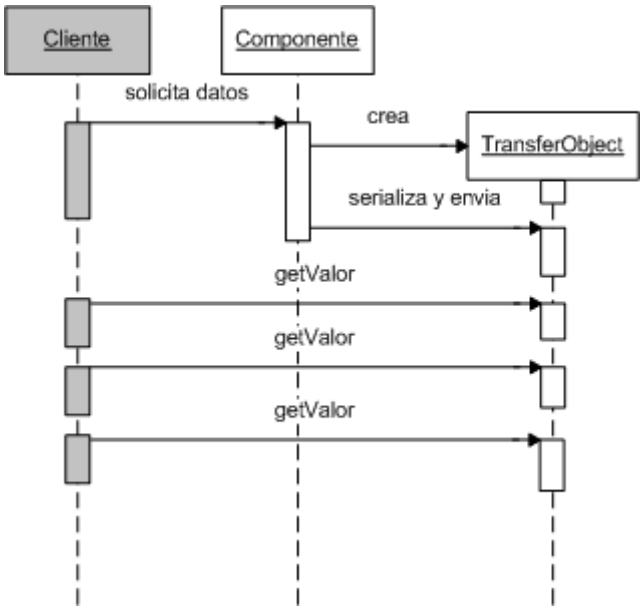


Diagrama de secuencia del patrón *Transfer Object*

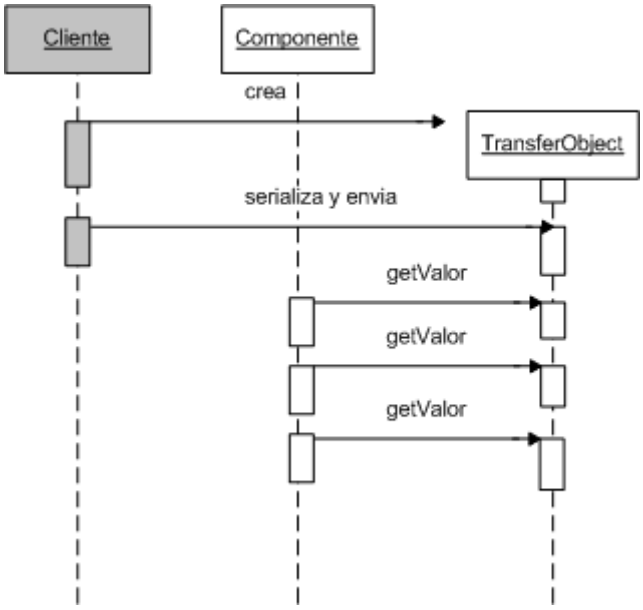


Diagrama de secuencia del patrón *Transfer Object*

En el primer diagrama de secuencia se muestra cómo el componente crea y devuelve un objeto de transferencia para atender la solicitud de datos del cliente.

El segundo diagrama de secuencia muestra cómo el cliente crea una instancia de *TransferObject* para hacer una actualización de datos y enviársela al componente.

Cliente

Entidad que necesita acceder al componente para recibir o enviar datos

Componente

Cualquier componente alojado en una capa distinta a la del cliente que envía o recibe datos de él

TransferObject

Es un POJO serializable que encapsula todos los datos que el cliente requiere para poder servirlos con tan sólo una llamada a un método

Estrategias

Estrategia de actualización con Transfer Objects

Los clientes necesitan obtener datos del componente y actualizarlos. La actualización se llevará a cabo a través de un método `setDatos(t TransferObject)` de granularidad gruesa que debe implementar el componente.

Al utilizar esta estrategia debemos tener en cuenta la sincronización entre varios clientes y llevar un control de versiones de los TransferObject para poder gestionarla.

Estrategia con múltiples Transfer Objects

En algunos casos, los componentes de negocio de nuestra aplicación pueden ser muy complejos y encapsular varios tipos de datos. Es probable que dichos componentes necesiten producir varios tipos de *Transfer Objects* distintos. Para ello deberán implementar métodos get para cada tipo diferente de *Transfer Object*.

Estrategia de especialización del Transfer Object

Cuando el componente está implementado como un EJB de entidad es frecuente que los datos que el cliente solicita se mapeen directamente de los atributos del componente. Para evitar la duplicación del código de los métodos get y set podemos hacer uso de la herencia.

La estrategia consiste en hacer que el componente sea una especialización del *Transfer Object*, de tal modo que ambos compartan los atributos y los métodos get y set.

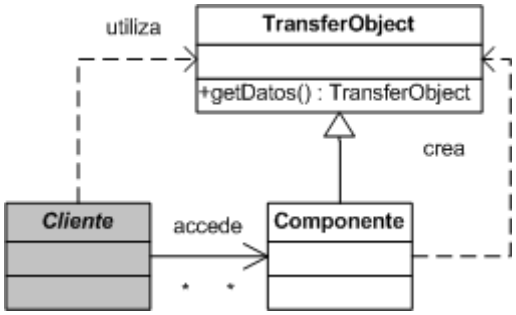


Diagrama de clases de la estrategia de especialización para el patrón *Transfer Object*

Consecuencias

- ✓ Reduce el tráfico de la red y mejora su rendimiento
- ✓ Simplifica los objetos remotos y sus interfaces remotas al implementar métodos de granularidad gruesa `getDatos` y `setDatos`

- ✓ Incrementa la complejidad de la sincronización y el control de versiones cuando usamos la estrategia de actualización con Transfer Objects

Patrones J2EE relacionados

- ✓ *Session Façade* utiliza frecuentemente los objetos de transferencia para intercambiar datos con los componentes que participan en los servicios de negocio. Además, dichos datos pueden ser transferidos al cliente por medio de *Transfer Object*
- ✓ *Value List Handler* provee los resultados de búsquedas como listas de *Transfer Objects* generadas dinámicamente
- ✓ *Transfer Object Assembler* construye objetos de transferencia partiendo de los datos de varias fuentes que suelen ser a su vez *Transfer Objects*

Transfer Object Assembler

Contexto y problema

Generalmente los clientes acceden a múltiples componentes de negocio (POJOs, EJBs. *Data Access Objects*, *Applications Services*, …) con la intención de construir un modelo de la aplicación a partir de los datos que estos devuelven encapsulados en distintos *Transfer Objects*.

Si el cliente es el encargado de construir dicho modelo, entonces, se acopla con múltiples componentes que debe conocer y que pueden estar en capas distintas. Al igual que ocurría en el patrón *Transfer Object* es posible que necesitemos un gran número de llamadas remotas para construir el modelo. Por si fuera poco, la complejidad del código del cliente aumenta y si tenemos múltiples clientes, entonces el código para generar el modelo estaría duplicado en todos ellos causando un empeoramiento de la mantenibilidad de nuestra aplicación.

Necesidades y motivaciones

- ✓ Queremos encapsular la lógica de negocio que genera el modelo y evitar su implementación por los clientes
- ✓ Queremos reducir el número de llamadas remotas así como el tráfico de la red cuando estamos construyendo modelos de la capa de negocio
- ✓ Queremos crear un modelo complejo para que este sea presentado (sólo lectura) por el cliente y ocultarle la complejidad del primero a este último
- ✓ Deseamos evitar relaciones de dependencia entre el cliente y los componentes de negocio

Solución

Aplicando el patrón Composite [GOF], crearemos un objeto nuevo que se encargue de agregar múltiples Transfer Objects para componer el modelo que solicita el cliente. Dicho modelo será de sólo lectura.

Estructura

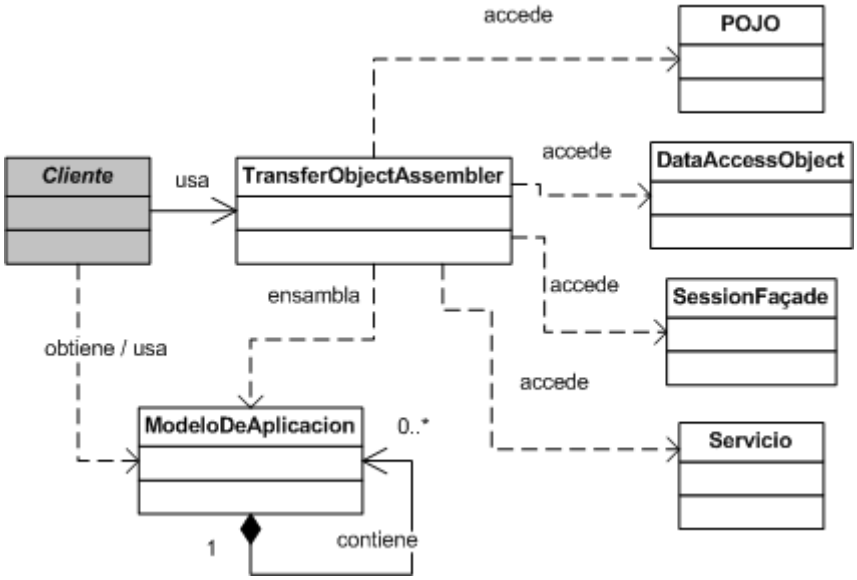


Diagrama de clases del patrón *Transfer Object Assembler*

Participantes y responsabilidades

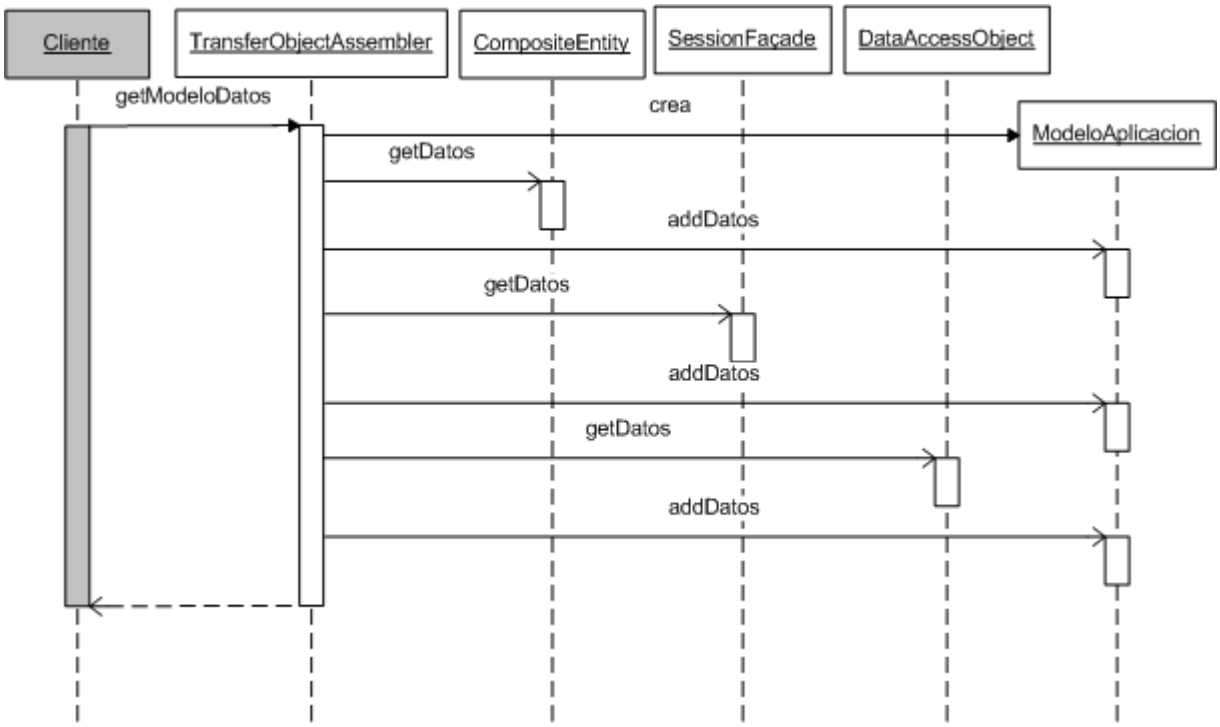


Diagrama de secuencia del patrón *Transfer Object Assembler*

TransferObjectAssembler

Se encarga de ensamblar todos los datos que le devuelven los distintos componentes para construir el modelo de la aplicación

ModeloAplicacion

Es un Composite *TransferObject* que será devuelto al cliente

SessionFaçade, CompositeEntity, Data Access Object

Son los componentes de los que el TransferObjectAssembler recuperará los datos necesarios para construir el modelo

Estrategias

Los Transfer Object Assembler pueden ser implementados como simples POJOs o con EJB de sesión. Si decide implementarlo como un Enterprise JavaBean de sesión, entonces probablemente se preguntará si este debe tener estado o no. En la mayoría de los casos la respuesta a esta pregunta es no ya que múltiples clientes estarán interactuando con la aplicación y el modelo que quiera construir estará en continuo cambio, por tanto, almacenarlo en el estado del EJB no tendrá sentido.

Consecuencias

- ✓ Separa la lógica de negocio y simplifica a los clientes
- ✓ Reduce el acoplamiento de los clientes con el modelo de la aplicación
- ✓ Mejora el rendimiento de la red al reducir el tráfico (menor número de llamadas remotas)

- ✓ Mejora el rendimiento del cliente ya que se construye el modelo sin usar recursos de este

Patrones J2EE relacionados

- ✓ Los *Transfer Object* son usados tanto como fuente de datos como medio para representar y transmitir el modelo al cliente
- ✓ *Composite Entity, Session Façade, Data Access Object*, ... son usados como fuente de datos
- ✓ *Service Locator* es usado por *Transfer Object Assembler* para encontrar los componentes de negocio que intervienen en el proceso de construcción del modelo

Value List Handler

Contexto y problema

Las aplicaciones J2EE necesitan tener funcionalidades de búsqueda y listado de entidades. Frecuentemente dichas funcionalidades son demandadas por la capa de presentación, ejecutadas por la capa de negocio y devueltos los resultados al cliente para que se los muestre al usuario.

Si estas funcionalidades devuelven un conjunto de resultados grande, enviarlos a través de la red al cliente es una operación costosa y generalmente innecesaria ya que el cliente tan sólo mostrará al usuario un subconjunto del total.

Por ejemplo, si en nuestra aplicación empresarial estamos listando el total de proveedores es más que probable que paginemos los resultados y que haya páginas de resultados que el cliente no llegue a ver porque encuentre antes el proveedor que busca.

Si nuestros objetos de negocio están implementados con Enterprise JavaBeans de entidad y hacemos uso de un método ejbFind la operación de búsqueda se tornará muy costosa. Esto es debido a que el método deberá acceder a cada uno de los beans de entidad, lo que conlleva asociado un gran número de llamadas a métodos get potencialmente remotas (lo serán en el caso de que el bean de entidad fuera remoto) y posteriormente un conjunto de referencias a objetos remotos (si el EJB es remoto) o, en el mejor de los casos, un conjunto de referencias a objetos locales (si el EJB es local). Como mínimo, estas referencias contendrán la clave primaria de la entidad resultado. No obstante, algunos contenedores devuelven adicionalmente las propias instancias, innecesarias al implementar el tipo de funcionalidades que estamos tratando.

Los métodos ejbFind están diseñados para hacer búsquedas con criterios que retornen pocos resultados.

Necesidades y motivaciones

- ✓ Evitar la sobrecarga producida por el uso de métodos ejbFind para búsquedas extensas o con muchos resultados
- ✓ Necesitamos un mecanismo que cachee el conjunto de resultados y que evite enviarlos en su totalidad al cliente
- ✓ Debemos poder mejorar la eficiencia de las peticiones que sean ejecutadas en repetidas ocasiones sobre datos razonablemente estáticos
- ✓ Queremos proporcionar mecanismos de iteración eficientes sobre los resultados

Solución

El *Value List Handler* asume las responsabilidades de ejecución de las búsquedas, cacheado de los resultados e iteración sobre ellos.

Para llevar a cabo una búsqueda *Value List Handler* utiliza directamente un objeto de acceso a datos *DAO*. El *DAO* devolverá los resultados en una colección que será recorrida por objetos iteradores.

Estructura

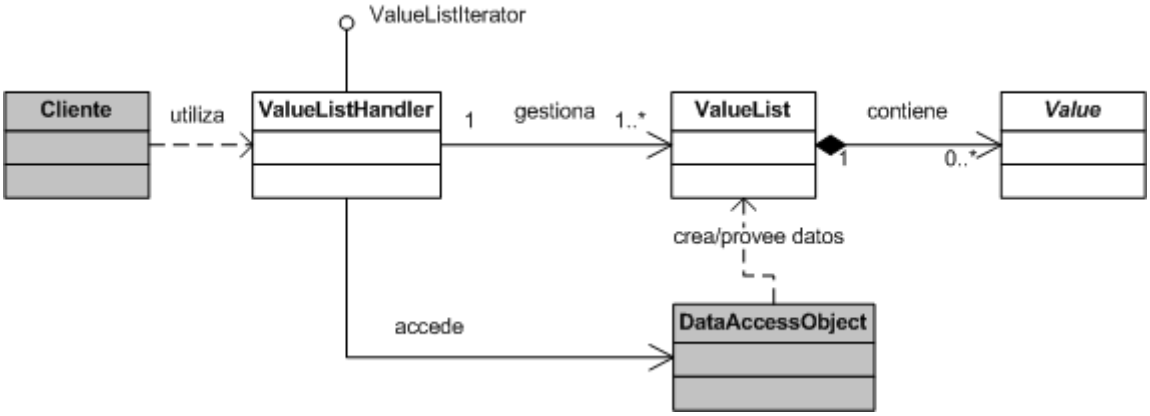


Diagrama de clases del patrón *Value List Handler*

Participantes y responsabilidades

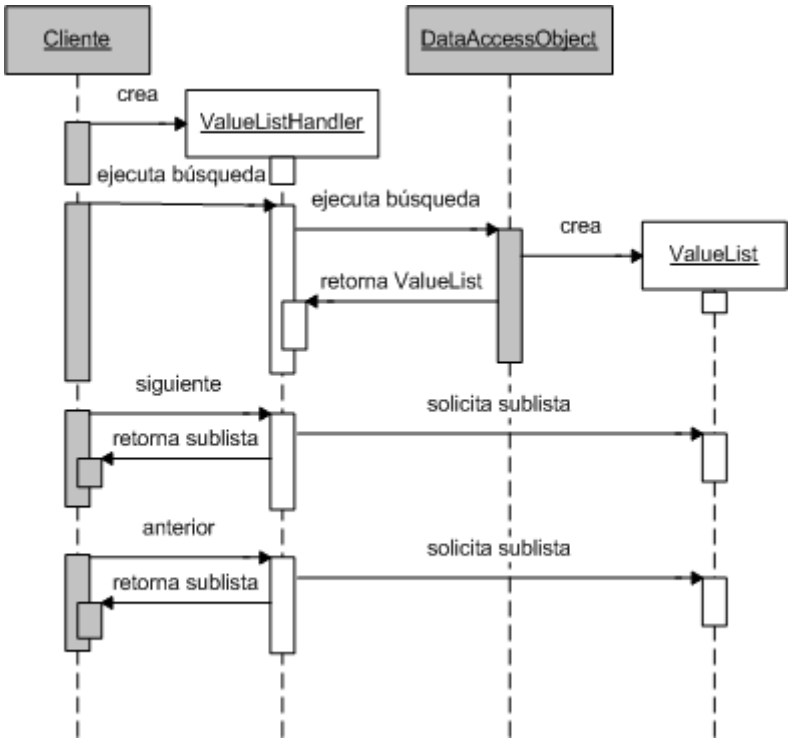


Diagrama de secuencia del patrón *Value List Handler*

Client

Pide la ejecución de la consulta que devuelve un gran número de resultados. Puede ser una *Session Façade* o un componente de la capa de presentación.

ValueListHandler

Ejecuta la consulta solicitada haciendo uso de un objeto de acceso a datos. Es el responsable también de cachear los resultados (si se considerara) y de manejarlos. Almacena los resultados en un atributo colección privado. Además, implementa la interfaz ValueListIterator que provee facilidades para iterar sobre los resultados de la consulta.

Cuando un cliente solicita resultados, *ValueListHandler* se encarga de recuperar, del total que tiene cacheados en su colección privada, una sublista de resultados. Posteriormente crea una nueva colección de *Transfer Objects* y la devuelve serializada al cliente.

ValueList

Colección de Transfer Objects que representan el resultado de una búsqueda

DataAccessObject

Value List Handler usa un objeto *DAO* para mantener separado el código de acceso a los datos del sistema de almacenamiento persistente con el que trabajemos.

Su responsabilidad es ejecutar sobre la base de datos (o el almacenamiento que usemos) la consulta y devolver los resultados. Dichos resultados no podrán ser devueltos como una estructura de datos dependiente de nuestro sistema de almacenamiento (base de datos, sistema legado, etc…) ya que sino estaríamos rompiendo los principios de encapsulación y protección.

Estrategias

POJO Handler Strategy

Value List Handler se implementa como un objeto POJO. Esto permite que sea utilizado por componentes de cualquiera de las capas que necesite las funcionalidades de búsqueda o listado.

Esta estrategia es adecuada para aplicaciones simples que no usen Enterprise JavaBean, por ejemplo, aplicaciones web simples con JSP y servlets, Business Delegates y Data Access Objects.

Ejemplo Value List Handler genérico – ValueListHandler.java

```
// ... importaciones ...

public class ValueListHandler
implements ValueListIterator {

  protected List list;
  protected ListIterator listIterator;

  public ValueListHandler() {
  }

  protected void setList(List list)
  throws IteratorException {
    this.list = list;
    if(list != null)
      listIterator =  list.listIterator();
    else
      throw new IteratorException("Lista vacia");
```

```
}

public Collection getList(){
  return list;
}

public int getSize() throws IteratorException{
  int size = 0;

  if (list != null)
    size = list.size();
  else
    throw new IteratorException(...); //No hay datos

  return size;
}

public Object getCurrentElement()
throws IteratorException {

  Object obj = null;

  if (list != null)
  {
    int currIndex = listIterator.nextIndex();
    obj = list.get(currIndex);
  }
  else
    throw new IteratorException(...);
  return obj;

}
```

```
public List getPreviousElements(int count)
throws IteratorException {
  int i = 0;
  Object object = null;
  LinkedList list = new LinkedList();
  if (listIterator != null) {
    while (listIterator.hasPrevious() && (i < count)){
      object = listIterator.previous();
      list.add(object);
      i++;
    }
  }// fin if
  else
    throw new IteratorException(...);

  return list;
}

public List getNextElements(int count)
throws IteratorException {
  int i = 0;
  Object object = null;
  LinkedList list = new LinkedList();
  if(listIterator != null){
    while(  listIterator.hasNext() && (i < count) ){
      object = listIterator.next();
      list.add(object);
      i++;
    }
  } /  / end if
  else
```

```
      throw new IteratorException(...);

    return list;
  }

  public void resetIndex() throws IteratorException{
    if(listIterator != null){
      listIterator = list.ListIterator();
    }
    else
      throw new IteratorException(...);
  }
  ...
}
```

Ejemplo Value List Handler específico – ProjectListHandler.java

```
// importaciones ...

public class ProjectListHandler
extends ValueListHandler {

  private ProjectDAO dao = null;
  // usa ProjectTO para determinar los criterios de búsqueda
  private ProjectTO projectCriteria = null;

  // El cliente crea una instancia de ProjectTO, establece
  // los valores que servirán para establecer los criterios
  // de la búsqueda y se la pasa al constructor del gestor

  public ProjectListHandler(ProjectTO projectCriteria)
```

```
  throws ProjectException, ListHandlerException {
    try {
      this.projectCriteria = projectCriteria;
      this.dao = PSADAOFactory.getProjectDAO();
      executeSearch();
    } catch (Exception e) {
      // Handle exception, throw ListHandlerException
    }
  }

  public void setCriteria(ProjectTO projectCriteria) {
    this.projectCriteria = projectCriteria;
  }

  // ejecuta una búsqueda con los criterios establecidos
  public void executeSearch()
  throws ListHandlerException {
    try {
      if (projectCriteria == null) {
        throw new ListHandlerException(
          "Criterios de búsqueda requeridos");
      }
      List resultsList =
        dao.executeSelect(projectCriteria);
      setList(resultsList);
    } catch (Exception e) {
      // Manejador de excepciones
    }
  }
}
```

Value List Handler Session Façade Strategy

Si nuestra aplicación J2EE utiliza beans empresariales, entonces, es mejor utilizar esta estrategia. Consiste en añadir a la construcción anterior una fachada implementada con un bean de sesión. De esta forma proveemos una interfaz para el acceso remoto a Value List Handler. El bean deberá tener estado para facilitar el cacheado y la iteración.

Consecuencias

Patrones J2EE relacionados

Patrones de la capa de integración

Data Access Object

Contexto y problema

Nuestras aplicaciones empresariales obtienen datos de múltiples fuentes (bases de datos relacionales u orientadas a objetos, directorios LDAP, sistemas legado, servicios externos B2B, …) y es común que se añadan o modifiquen algunas de estas fuentes durante la vida de la aplicación. Los mecanismos de acceso y las APIs de dichos sistemas de almacenamiento persitente de la información son heterogéneas y poseen extensiones propietarias (no estándares).

Si mezclamos la lógica de nuestra aplicación con el código de acceso a los datos estaremos introduciendo dependencia con las fuentes. Un cambio en una fuente de datos implicaría la modificación de la lógica de nuestra aplicación para adaptarla a las nuevas formas de acceder y manejar los datos.

Necesidades y motivaciones

- ✓ Los componentes de nuestra aplicación necesitan acceder y almacenar datos en diferentes sistemas de almacenamiento persistente y otras fuentes de datos
- ✓ La portabilidad de los componentes se ve afectada si accedemos al sistema de almacenamiento persistente a través de una API propietaria
- ✓ Queremos separar e independizar el código de nuestra aplicación del código de acceso a datos

- ✓ Deseamos presentar una API uniforme de acceso a los datos para todos los posibles almacenamientos de datos persistentes, como RDBMS, OOBD, LDAB, respositorios XML, B2B, …
- ✓ Necesitamos encapsular las funcionalidades propietarias específicas de nuestras fuentes para incrementar la mantenibilidad y portabilidad de nuestra aplicación

Solución

Usar un *Data Access Object* (conocido como *DAO*) para abstraer y encapsular todos los accesos a los sistemas de almacenamiento persistente de la información. El DAO gestionará todas las conexiones con las fuentes de datos para obtener y almacenar los datos.

El objeto *DAO* ocultará completamente la implementación de la fuente de datos y su API a los componentes de nuestra aplicación, por tanto, la misión esencial de un DAO es la de ejercer de adaptador entre los componentes y las fuentes de datos.

Los objetos *DAO* no deben lanzar excepciones específicas de la fuente de datos al igual que no debe exponer ninguna estructura de datos u objeto específico de estas. Por ejemplo, si tenemos un *DAO* que encapsula las conexiones JDBC no debemos lanzar excepciones `java.sql.*` ni `javax.sql.*` a los clientes del *DAO* ni enviarles objetos `ResultSet`. En su lugar deberemos capturar las excepciones específicas de JDBC en el DAO y lanzar otras que tengan sentido para los clientes (por ejemplo, `cuentaInexistenteException()`). Así mismo deberemos devolver los datos en estructuras independientes de la fuente de datos, normalmente *TransferObjects* (por ejemplo en un objeto de transferencia `CuentaTO`).

Los objetos DAO no tienen estado ni cachean los resultados de las consultas, por tanto, son objetos muy ligeros que pueden ser accedidos de forma concurrente.

Estructura

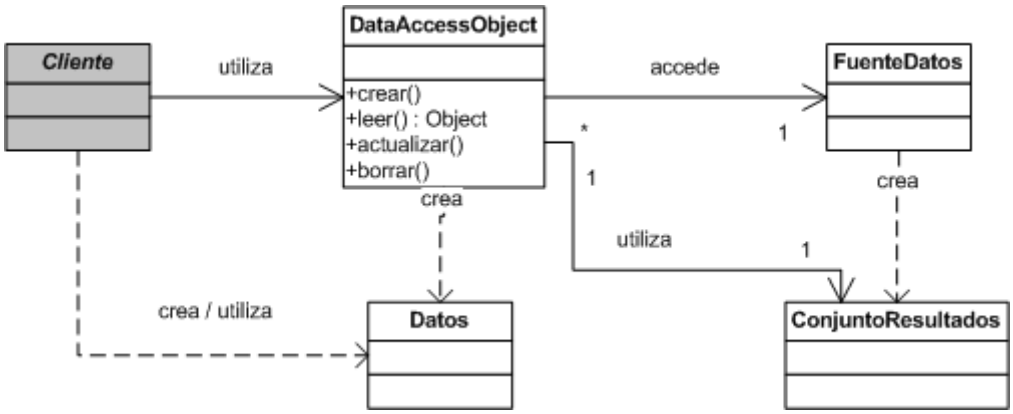


Diagrama de clases del Patrón *Data Access Object*

Participantes y responsabilidades

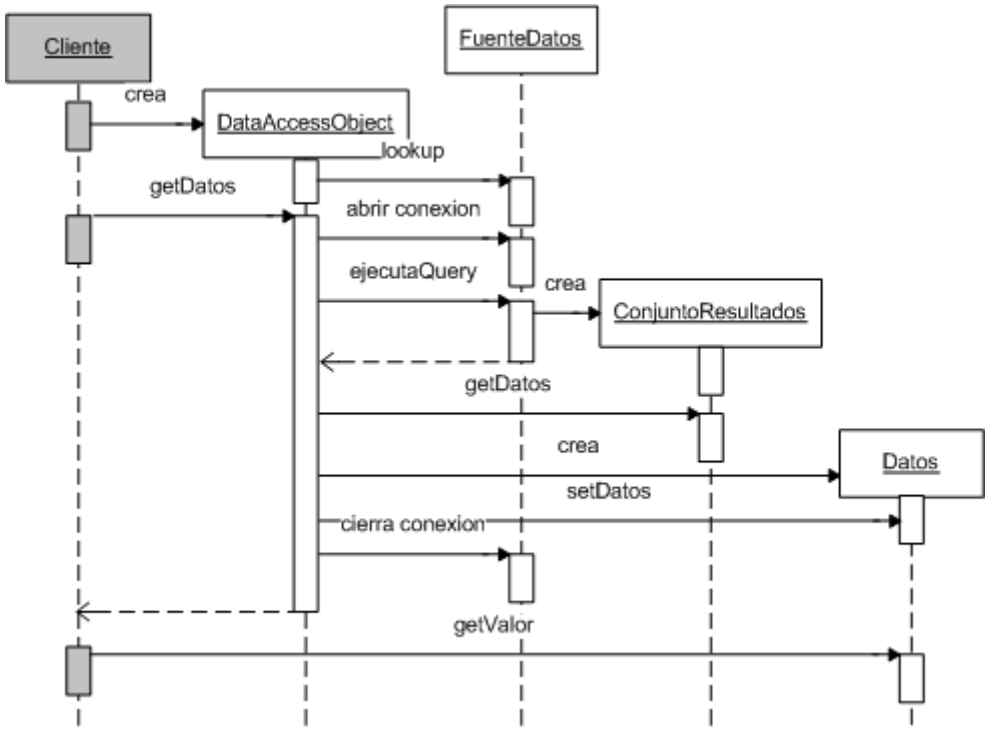


Diagrama de secuencia del patrón *Data Access Object*

DataAccessObject

Encapsula el acceso a datos para el cliente

FuenteDatos

Puede ser una base de datos (Oracle, MySQL, SQL Server, ...) , un sistema legado, un servicio web, o cualquier otro sistema persistente de datos

ConjuntoResultados

Representa el resultado de la consulta. Es dependiente de la fuente de datos, por ejemplo, si usamos la API JDBC como fuente de datos, un objeto java.sql.ResultSet podría tomar este rol

Datos

Es un TransferObject que se devolverá al cliente de tal modo que este será totalmente independiente de la fuente de datos concreta que utilizemos.

Consecuencias

- ✓ Centraliza el acceso a datos y lo hace transparente para los clientes
- ✓ Facilita la migración a distintas fuentes de datos
- ✓ Encapsula el esquema de la base de datos y provee una visión Orientada a Objetos
- ✓ Reduce la complejidad del código de los clientes
- ✓ Introduce complejidad al obligar a los clientes a que trabajen con objetos en vez de con ResultSet

Patrones J2EE relacionados

- ✓ *TransferObject* : Es utilizado para transferir los datos al cliente

Service Activator

Problema

Nuestra aplicaciones J2EE pueden tener procesos de negocio pesados que requieran un tiempo y recursos considerables para obtener una respuesta. Por tanto, si llamamos a estos procesos de manera síncrona, el cliente quedará bloqueado a la espera de la respuesta. Frecuentemente esta situación no es asumible ya que el cliente.

Necesidades y motivaciones

- ✓ Necesitamos invocar servicios de negocio, POJOs o componentes EJB de una manera asíncrona

Solución

Añadir un Service Activator que implemente un JMS Listener (*Java Message Service*). Podemos usar un bean dirigido a mensajes (MDB) con contenedores que implementen EJB versión 2.0 o superior para recibir mensajes de forma asíncrona. Si nuestro contenedor de EJBs implementa versiones anteriores, entonces tendremos que utilizar nuestra propia solución basada en JMS.

Además, los MDBs son objetos sin estado. Los EJB de entidad o de sesión con estado no pueden ser accedidos.

La filosofía del Service Activator es que todos los clientes que requieran llamar a nuestros servicios de negocio (EJB, POJOs,…) de manera asíncrona deberán hacerlo enviando un mensaje JMS al Service Activator. Éste será el encargado de interpretar el mensaje para identificar el servicio que el cliente requiere, localizarlo y posteriormente invocarlo para que procese la petición del cliente de forma asíncrona. Posteriormente, el Service

Activator puede generar, si así lo solicita el cliente, un nuevo mensaje JMS para comunicarle que el proceso ha finalizado y los resultados del mismo.

Estructura

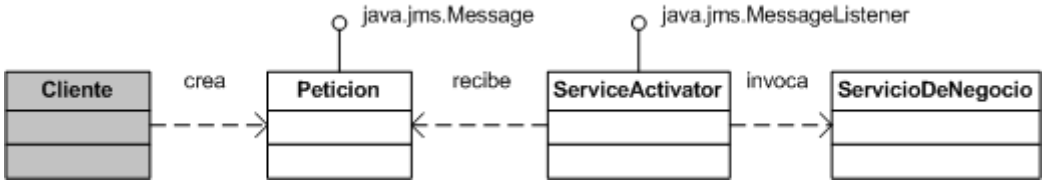
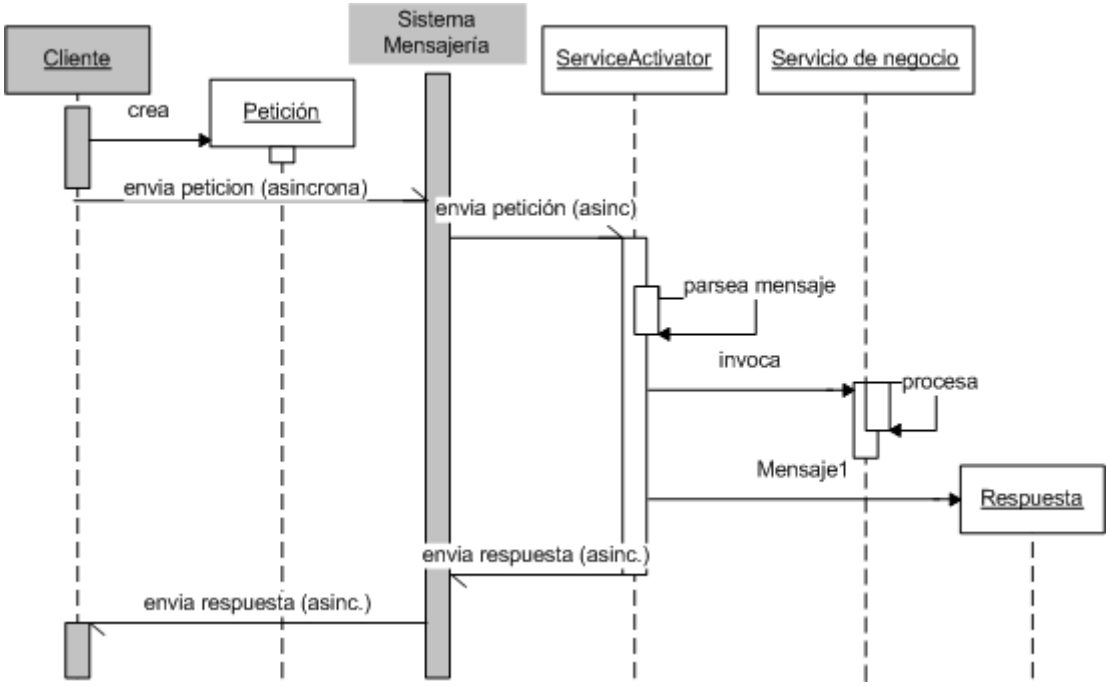


Diagrama de clases del patrón *Service Activator.*

Participantes y responsabilidades



Posible diagrama de secuencia del patrón *ServiceActivator*

Cliente

Solicita el proceso de negocio al Service Activator vía un mensaje JMS

Petición

Mensaje JMS (javax.jms.Message) que envía el cliente con la petición de servicio.

ServiceActivator

Recibe el mensaje de petición y se encarga de localizar e invocar los objetos de negocio. Una vez terminado el procesamiento genera un nuevo mensaje Respuesta que será enviado como los resultados (si este los solicito).

Servicio de negocio

Servicio de negocio que procesara la petición

Respuesta

Mensaje JMS (javax.jms.Message). de retorno que indica el fin del proceso y los resultados al cliente

Estrategias

Estrategia POJO para Service Activator

La estrategia más directa consiste en implementar un JMS Listener en un servlet que escuche y procese los mensajes recibidos.

A continuación se desarrolla un ejemplo con dicha estrategia consistente en un JMS Listener que esperará mensajes con órdenes de pedidos y un cliente que las enviará para que sean procesadas de forma asíncrona.

Ejemplo de Service Activator específico - OrderServiceActivator.java

```
public class OrderServiceActivator implements
  javax.jms.MessageListener{

  // Encola las sesiones y los receptores
  // Vea la API de JMS para más información
  private QueueSession orderQueueSession;
  private QueueReceiver orderQueueReceiver;


  private String connFactoryName =
    "PendingOrdersQueueFactory";
  private String queueName = "PendingOrders";

  // Usamos un Service Locator para localizar
  // componentes JMS administrados
  private JMSServiceLocator serviceLocator;

  public OrderServiceActivator(String connFactoryName,
      String queueName) {
    super();
    this.connFactoryName = connFactoryName;
    this.queueName = queueName;
    startListener();
  }

  private void startListener() {
    try {
      serviceLocator = new JMSServiceLocator
            (connFactoryName);
      qConnFactory =
```

```
            serviceLocator.getQueueConnectionFactory();
        qConn = qConnFactory.createQueueConnection();


        orderQueueSession = qConn.createQueueSession (...);
        Queue ordersQueue =
                serviceLocator.getQueue(queueName);
        orderQueueReceiver =
          orderQueueSession.createReceiver(ordersQueue);
        orderQueueReceiver.setMessageListener(this);
      }
      catch (JMSException excp) {
           // manejo errores
      }
    }

    // Este es el método que será invocado asíncronamente cuando un mensaje
    // llegue a la cola asignada en el constructor del Service Activator
    // Para más información vea la API de JMS.

public void onMessage(Message msg) {
      try {
          // parseamos el mensaje msg. Vea la API JMS Message.
          ...

          // Invocamos los métodos de nuestro proceso de negocio
          // vía un Bussiness Delegatte
          OrderProcessorDelegate orderProcDeleg =
              new OrderProcessorDelegate();

          // Usamos los datos que traía el mensaje para invocar
          // el método vía el delegado
```

```
        orderProcDeleg.fulfillOrder(...);

        // Enviamos respuesta...
    }
    catch (JMSException jmsexcp) {
      // Manejamos las excepciones JMSExceptions
    }
    catch (Exception excp) {
      // Manejamos el resto de excepciones
    }
  }

  public void close() {
    try {
      // Limpiamos antes de finalizar
      orderQueueReceiver.setMessageListener (null);
      orderQueueSession.close();
    }
    catch(Exception excp) {
      // Manejamos excepciones que se produzcan al cerrar
    }
  }
}
```

Ejemplo de cliente - OrderDispatcherFacade.java

```
// importaciones...
public class OrderDispatcherFacade
  implements javax.ejb.SessionBean {
```

```
  ...
  // Métodos para crear nuevos pedidos
  public int createOrder(...) throws OrderException {

    // creamos nuevos beans de entidad con los pedidos
    ... order = ...

    // Una vez creado el pedido lo procesamos
    // de modo asíncrono en backend
    OrderSender orderSender = new OrderSender();
    orderSender.sendOrder(order);

    // Cerramos el OrderSender si todo se ha hecho correctamente...
    orderSender.close();
    ...
  }
}
```

Como puede ver el cliente se ha apoyado en un objeto OrderSender para enviar las peticiones. Esto se ha hecho en aras de una mayor reutilización. Sacar fuera la funcionalidad de envío de peticiones permite que varios clientes la reutilizen sin tener que duplicar código.

```
// importaciones
public class OrderSender {
  // Colas de sesiones y emisores
  private QueueSession orderQueueSession;
  private QueueSender orderQueueSender;

  private String connFactoryName =
    "PendingOrdersQueueFactory";
```

```
private String queueName = "PendingOrders";

// usamos un Service Locator para localizar
// componentes JMS administrados, por ejemplo, colas
// o factorías de ellas
private JMSServiceLocator serviceLocator;
...
// método para crear e inicializar la cola emisora
private void createSender() {
  try {
    serviceLocator = new JMSServiceLocator
          (connFactoryName);
    qConnFactory =
        serviceLocator.getQueueConnectionFactory();
    qConn = qConnFactory.createQueueConnection();

    // See JMS API for method usage and arguments
    orderQueueSession = qConn.createQueueSession
        (...);
    Queue ordersQueue =
            serviceLocator.getQueue(queueName);
    orderQueueSender =
        orderQueueSession.createSender(ordersQueue);
  catch(Exception excp) {
    // Handle exception - Failure to create sender
  }
}

// método para despachar de forma asíncrona
// los pedidos a los procesos de negocio
public void sendOrder(Order newOrder) {
```

```
      // Creamos un nuevo mensaje para enviar los pedidos
      ObjectMessage objMessage =
        queueSession.createObjectMessage(order);

      // Tenemos que especificar las propiedades del Message
      // así como el modo de envío. Más info en la API JMS
      ...

      // Registramos el pedido en el mensaje
      objMessage.setObject(order);

      // enviamos el mensaje a la cola
      orderQueueSender.send(objMessage);

      ...
    } catch (Exception e) {
      // Handle exceptions
}
    ...
  }
  ...
  public void close() {
    try {
      // limpiamos antes de cerrar
      orderQueueReceiver.setMessageListener (null);
      orderQueueSession.close();
    }
    catch(Exception excp) {
      // Manejo de excepciones al cerrar
    }
  }
```

```
}
```

Estrategia MDB para Service Activator

Como comentábamos al principio de este apartado, con versiones EJBs 2.x o superiores podemos abordar la tarea de implementar un *Service Activator* de una forma más sencilla gracias a los beans dirigidos a mensaje conocidos como MDB (message-driven bean).

Los MDB simplifican la tarea de codificación ya que tan sólo deberemos implementar el método `onMessage()` que se encargará de atender los mensajes que lleguen. Por otro lado, la gestión de colas de mensajes la realizará el contenedor de EJBs de manera transparente haciendo uso de información declarativa que incluiremos en nuestros descriptores de despliegue.

Consecuencias

- ✓ Integra JMS en implementaciones anteriores a EJB 2.0
- ✓ Provee acceso asíncrono para cualquier componente de negocio

Patrones J2EE relacionados

- ✓ *Session Façade, Bussines Delegate, Service Locator* toman frecuentemente el rol de Componentes.