Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

ARQUITECTURA DE APLICACIONES J2EE BASADAS EN EL PATRÓN MVC UTILIZANDO ORACLE ADF

Daniel Caciá Rivas

Asesorado por el Ing. Victor Eduardo Quan Castañeda

Guatemala, febrero de 2007

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

ARQUITECTURA DE APLICACIONES J2EE BASADAS EN EL PATRÓN MVC UTILIZANDO ORACLE ADF

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

DANIEL CACIÁ RIVAS

ASESORADO POR EL ING. VICTOR QUAN CASTAÑEDA
AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, FEBRERO DE 2007

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Murphy Olympo Paiz Recinos
VOCAL I	Inga. Glenda Patricia García Soria
VOCAL II	Lic. Amahán Sánchez Álvarez
VOCAL III	Ing. Miguel Ángel Dávila Calderón
VOCAL IV	Br. Kenneth Issur Estrada Ruiz
VOCAL V	Br. Elisa Yazminda Vides Leiva
SECRETARIA	Inga. Marcia Ivonne Véliz Vargas

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANO	Ing. Murphy Olympo Paiz Recinos
EXAMINADOR	Inga. Virginia Victoria Tala Ayerdi
EXAMINADOR	Ing. Fredy Javier Gramajo López
EXAMINADOR	Ing. César Augusto Fernández Cáceres
SECRETARIO	Ing. Pedro Antonio Aguilar Polanco

HONORABLE TRIBUNAL EXAMINADOR

Cumpliendo con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

ARQUITECTURA DE APLICACIONES J2EE BASADAS EN EL PATRÓN MVC UTILIZANDO ORACLE ADF,

tema que me fuera asignado por la Dirección de la Escuela de Ingeniería en Ciencias y Sistemas, en agosto de 2005.

DANIEL CACIÁ RIVAS

AGRADECIMIENTOS A:

MIS PADRES	Por ofrecerme las herramientas necesarias para perseverar, esforzarme y culminar con éxito esta etapa de mi vida.
DATUM, S.A.	Empresa donde laboro, por haberme brindado los recursos necesarios para culminar este trabajo de investigación
MIS COMPAÑEROS DE TRABAJO	Ya que, siempre me apoyaron y tomarón su tiempo para compartirme un poco de su sabiduría
LA ESCUELA DE CIENCIAS Y SISTEMAS	Especialmente a la Inga. Elizabeth Domínguez por facilitarmente los requerimientos necesarios para poder elaborar el ejemplo práctico que se detalla en el presente trabajo

DEDICATORIA A:

MIS PADRES	Daniel, por sembrar en mí la semilla de la sabiduría y brindarme siempre sabios consejos y un ejemplo que poder imitar. "Abue", por compartir conmingo tu forma de ser y hacerme sentir siempre especial
MI ESPOSA	Glenda, te amo y me completas. Gracias por estar a mi lado siempre.
MIS HIJOS	Jorge Daniel y Gabriel Alejandro por ser mi inspiración y llenarme de felicidad
MIS HERMANOS	Jorge, te admiro por enseñarne a sobrepasar cada altibajo y hacer de cada obstáculo una experiencia para hacerte más grande. Vicky, por que el amor que desbordas y tu inocencia son contagiosos.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

FIGURAS

TABLA

GLOSARIO

Ámbito Contorno de un espacio. Espacio entre límites. En programación, alcance que tenga la definición de una variable.

API Application programming interface, por sus siglas en inglés. Un a API consiste en un conjunto de definiciones de la forma en la que un componente de software se comunica con otro. Es un método utilizado usualmente para implementar abstracción entre un nivel bajo de programación y un nivel complejo.

Bean Componentes de software que cumple con las especificaciones de la especificación J2EE de Enterprise Java Beans y que se ejecutan en un servidor J2EE.

Cache En términos de la ciencia de la computación, cache se refiere al proceso de duplicar los datos originales que se encuentran en una ubicación difícil de acceder -usualmente en términos de tiempo- de manera que puedan ser accedidos de una forma más rápida.

Castear En términos de la ciencia de la computación, se refiere a la operación que permite transformar un tipo de dato a otro bajo ciertas condiciones.

Commit	En términos de la ciencia de la computación y manejo de almacenamiento de datos, se refiere a la idea de hacer permanentes un conjunto de cambios alternativos, tal es el caso de el fin de una transacción. Ejecutar un commit es hacer los cambios temporales permanentes. Commit es un acto de compromiso.
Data streams	En telecomunicaciones, un data stream es una secuencia de señales coherentes codificadas digitalmente –paquetes- utilizadas para transmitir o recibir información en una transmisión
Herencia	Forma de obtener o extender la funcionalidad de una clase existente en una nueva clase.
Host	Ordenador/computadora conectado(a) a la Internet
Instanciar	En términos de programación orientada a objetos, proceso mediante el cual se obtiene una instancia de una clase. Creación de un objeto a partir de una clase.
Modularizar	En términos de la ciencia de la computación, proceso mediante el cuál la construcción de una solución se divide en módulos de tal forma que la implementación sea escalable.

SQL	Lenguaje de computación utilizado para crear, modificar y recuperar datos de un sistema de base de datos relacional.
Transacción	En el manejo de almacenamiento de datos, una transacción se refiere a una unidad de trabajo que debe ser realizada de forma atómica, consistente, aislada y durable.
Transportabilidad	En la ciencias de la computación, propiedad de las aplicaciones que pueden ser transportadas entre diferentes sistemas operativos o plataformas sin necesidad de modificarlas o recompilarlas.

RESUMEN

Las aplicaciones Web pueden desarrollarse utilizando cualquier arquitectura posible. Es por tal razón que existe una gran variedad de patrones de diseño y construcción de software. Uno de estos patrones de diseño es la arquitectura Modelo-Vista-Controlador el cual es una paradigma de programación que se puede aplicar tanto a el desarrollo de aplicaciones con interfaz gráfica (GUI) y al desarrollo de aplicaciones para Internet.

El principal objetivo de la arquitectura MVC es aislar tanto los datos de la aplicación como el estado (modelo) de la misma, del mecanismo utilizado para representar (vista) dicho estado, así como para modularizar esta vista y modelar la transición entre estados del modelo (controlador). Existe una gran cantidad de plataforma en las cuales se puede implementar el desarrollo de aplicaciones. Una de estas plataformas es J2EE o Java 2 Enterprise Edition, la cual es un conjunto de estándares para desarrollar e implementar aplicaciones de tipo empresarial.

Muchas veces, el desarrollo de aplicaciones para Internet utilizando la plataforma J2EE puede ser muy tedioso y complicado, es allí donde se hace necesario utilizar un framework de desarrollo de aplicaciones y para nuestro interés se utilizará Oracle ADF. Oracle ADF ayuda a obtener mayor productividad al permitir que los desarrolladores se concentren en definir la lógica del negocio para su aplicación, en contraposición con la escritura manual del código de nivel bajo para implementar la aplicación. Adicionalmente, genera en forma automática el código de infraestructura e implementa las mejores prácticas para ayudar a los desarrolladores a diseñar y crear aplicaciones J2EE optimizadas y seguras para la empresa.

OBJETIVOS

GENERALES

1. Proveer un documento en el cual se pueda encontrar una definición y explicación del desarrollo de aplicaciones para Internet utilizando la plataforma J2EE.

2. Mostrar los beneficios de utilizar un framework de desarrollo para facilitar el desarrollo de aplicaciones para el Web utilizando la tecnología J2EE.

ESPECÍFICOS

1. Indicar que son patrones de diseño y las ventajas de utilizarlos en el desarrollo de aplicaciones para Internet.

2. Describir las diferentes capas del modelo MVC -Modelo-Vista-Controlador- y sus principales componentes.

3. Identificar las mejores prácticas de implementación del modelo MVC.

4. Desarrollar para la Escuela de Ciencias y Sistemas de la Facultad de Ingeniería una aplicación Web basada en el patrón de diseño MVC, utilizando un *framework* de desarrollo de aplicaciones.

5. Redactar un informe final.

INTRODUCCIÓN

Las aplicaciones Web pueden desarrollarse utilizando cualquier arquitectura posible. Es por tal razón que existe una gran variedad de patrones de diseño y construcción de software. Los patrones de diseño de software son soluciones reutilizables a los problemas comunes que ocurren durante el desarrollo de un sistema de software o aplicaciones.

Los patrones de diseño de software proporcionan un proceso consistente o diseño que uno o más desarrolladores pueden utilizar para alcanzar sus objetivos. También, proporciona una arquitectura uniforme que permite una fácil expansión, mantenimiento y modificación de una aplicación.

La arquitectura del patrón Modelo-Vista-Controlador es una paradigma de programación que se puede aplicar tanto a el desarrollo de aplicaciones con interfaz gráfica (GUI) como al desarrollo de aplicaciones para Internet.

El principal objetivo de la arquitectura MVC es aislar tanto los datos de la aplicación como el estado, modelo, de la misma, del mecanismo utilizado para representar, vista, dicho estado, así como para modularizar esta vista y modelar la transición entre estados del modelo, controlador. Las aplicaciones construidas bajo este patrón se dividen en tres grandes áreas funcionales.

- Vista: la presentación de los datos
- Controlador: el que atenderá las peticiones y componentes para toma de decisiones de la aplicación
- Modelo: la lógica del negocio o servicio y los datos asociados con la aplicación

El propósito del patrón de diseño MVC es aislar los cambios. Es una arquitectura preparada para los cambios, la cual desacopla datos y lógica del negocio de la lógica de presentación, permitiendo la actualización y desarrollo independiente de cada uno de los citados componentes. El patrón de diseño Modelo-Vista-Controlador consta de: una o más vista de datos, un modelo, el cual representa los datos y su comportamiento y un controlador que controla la transición entre el procesamiento de los datos y su visualización.

En una aplicación que utiliza el patrón de diseño MVC, el cliente es la entidad o proceso que realiza una petición a la aplicación, está solicitud es enviada al controlador, el cual decide quien puede responder a dicha solicitud de mejor forma. El modelo implica la lógica del negocio y es controlada por el controlador que por medio de envío y recepción de parámetros devuelven los datos necesarios para satisfacer la solicitud del cliente.

La respuesta que da el modelo es enviada a través del controlador a la vista y es este componente el que se encarga de presentar los datos de respuesta al cliente de la manera más adecuada, por ejemplo, si el cliente es un teléfono móvil o un explorador de Internet, o bien una aplicación con interfaz gráfica.

Existe una gran cantidad de plataformas en las cuales se puede implementar el desarrollo de aplicaciones. Una de estas plataformas es J2EE o Java 2 Enterprise Edition, la cual es un conjunto de estándares para desarrollar e implementar aplicaciones de tipo empresarial.

J2EE hace énfasis en la portabilidad y aprovechamiento de la creación de aplicaciones basadas en componentes, con el fin de permitir la mejor

administración de las aplicaciones. J2EE soporta componente para cuatro capas: cliente, Web, negocio y Sistemas de Información Empresarial -EIS, por sus siglas en inglés- o de Acceso a Datos.

J2EE define una plataforma para desarrollar, publicar y ejecutar aplicaciones basada en un modelo de aplicación multicapas y distribuido. De lo anterior se deriva el hecho de que la lógica de una aplicación J2EE puede ser dividida en componentes basados en su funcionalidad y distribuidas en la capa apropiada en la arquitectura multicapas.

Por lo anterior y por otras características que implementa J2EE, se propone un modelo MVC para construir aplicaciones con el fin de obtener el máximo provecho de esta plataforma y, sobre todo, facilitar la implementación de la solución así como su administración y sobre todo su mantenimiento.

En primera instancia, las aplicaciones para Internet basadas en el patrón de desarrollo de MVC puedes ser implementadas con J2EE utilizando JSP para las vistas, servlets como controladores y JDBC para el modelo.

Muchas veces, el desarrollo de aplicaciones para Internet utilizando la plataforma J2EE puede ser muy trabajoso y complicado, es allí donde se hace necesario utilizar un framework de desarrollo de aplicaciones como Oracle ADF. Oracle ADF ayuda a obtener mayor productividad al permitir que los desarrolladores se concentren en definir la lógica del negocio para su aplicación, en contraposición con la escritura manual del código de nivel bajo para implementarla. Adicionalmente, genera en forma automática el código de infraestructura e implementa las mejores prácticas para ayudar a los desarrolladores a diseñar y crear aplicaciones J2EE y servicios Web optimizados y seguros para la empresa.

La arquitectura de Oracle ADF se diseño con la intención de permitir a desarrolladores de aplicaciones J2EE utilizarla para mejorar la productividad junto con otros componentes y marcos de trabajo J2EE y servicios Web, como por ejemplo, Enterprise JavaBeans, JavaServer Pages, Business Components for Java (BC4J), Struts, JavaServer Faces entre otras. El modelo de datos en una aplicación ADF incorpora archivos XML, que describen la estructura de la aplicación y captan su comportamiento y reglas comerciales. Estos archivos XML de metadatos pueden ser adaptamos para satisfacer requerimientos propios de cada empresa, permitiendo ahorrar tiempo de programación al implementar componentes reutilizables y adaptablas para cada nueva aplicación.

El presente trabajo de graduación pretende dar a conocer el patrón de diseño MVC implementado con Oracle ADF como alternativa para todas aquellas empresas que desean construir aplicación utilizando la especificación J2EE.

1. ORACLE ADF, *FRAMEWORK* PARA DESARROLLO DE APLICACIONES J2EE

1.1. Introducción

1.1.1. Definición de *framework*

Un *framework* o marco de trabajo, es una infraestructura que soporta un conjunto de conceptos, valores y prácticas que facilitan la construcción de aplicaciones. Un *framework* provee una serie de herramientas y componentes que permiten modelar e implementar de manera natural la realidad.

En el desarrollo de software, un *framework* o marco de trabajo es definido como una infraestructura de soporte en la cual un proyecto de software puede ser organizado y desarrollado. Regularmente, un *framework* puede incluir soporte de programas, librerías y código preescrito para ayudar a desarrollar y elaborar los diferentes componentes del proyecto de software que se construye con dicho *framework*.

Un marco de trabajo para desarrollo de aplicaciones debe cumplir con las siguientes características:

- Ser una capa productiva para construir aplicaciones
- Contener un conjunto de componentes inteligentes de software que cooperan entre sí
- Estar diseñado para especializarse en la lógica del negocio

- Manejar el mayor número de tareas comunes con comportamiento crítico para la aplicación
- Permitir personalizar de forma sencilla el comportamiento por defecto
- Utilizar estándares, proveer técnicas y patrones de diseño

Un marco de trabajo para desarrollo de aplicaciones provee una serie de componentes de software, que están diseñados para ayudar a la construir y modularizar una aplicación, de manera sencilla y probablemente de forma automática. Un marco de trabajo provee componentes definidos para ocupar lugares específicos en el proceso de desarrollo de soluciones de software, tal como consultas, lógica del negocio y validaciones y manejo de la persistencia de datos.

Los componentes de software deben ser inteligentes y deben integrarse con todos los componentes del marco de trabajo. Los componentes deben permitir la personalización de su comportamiento de forma simple para adecuarse a las necesidades específicas del negocio. El marco de trabajo debe también manejar la mayoría de tareas comunes que tengan especial incidencia en el desarrollo de aplicaciones y que sean comportamientos estándares.

1.1.2. Entiendo los *framework* para desarrollo de aplicaciones

Un marco de trabajo provee código “detrás de escena” para manejo de la funcionalidad básica de la aplicación. Estas funciones incluyen comportamiento estándar, tal como validación de datos y lógica del negocio. También incluyen métodos para acceder a los datos y manejo de transacciones. Con estas funciones y métodos preconstruidas, los desarrolladores de software pueden dedicarse a la construcción del software

específico para el negocio, en lugar de escribir programas de bajo nivel para manejar la infraestructura de la aplicación.

Además de la funcionalidad básica que el marco de trabajo provee, este debe ser fácil de personalizar. De ser necesario, se puede aumentar o inclusive modificar toda la funcionalidad por defecto según las reglas del negocio.

Otro aspecto importante de un marco de trabajo es, que los objetos que se crean, deben contener únicamente el código del desarrollador y no código que genera el *framework*. Esto permite que el código del desarrollador no se mezcle con el código generado por el *framework*, de esa manera, ninguna funcionalidad que el desarrollador agregue, modificará el código generado. En el desarrollo de software utilizando *frameworks*, no hay código generado que deba ser modificado para personalizar el comportamiento por defecto.

1.2. Oracle ADF, *framework* para desarrollo de aplicaciones J2EE

El marco de trabajo para desarrollo de aplicaciones J2EE Oracle ADF es un conjunto de componentes, estándares y patrones de diseño que facilitan el desarrollo de aplicaciones J2EE, minimizando la necesidad de escribir código que implemente patrones de diseño y la infraestructura de la aplicación.

Las principales características de Oracle ADF, que lo hacen único en comparación con otros *frameworks* de desarrollo J2EE, son las siguientes:

- **Ambiente de desarrollo:** Otros marcos de trabajo para desarrollo de aplicaciones J2EE no proveen de una herramienta de desarrollo

que permite desarrollar fácilmente con ellos. En el caso de Oracle ADF, Oracle JDeveloper es una herramienta poderosa para la construcción de aplicaciones utilizando este marco de trabajo, de forma visual y declarativa, por lo tanto reduce la necesidad de escribir código.

- **Independiente de plataforma:** Las librerías de Oracle ADF pueden ser instaladas en cualquier servidor de aplicaciones que cumpla al 100% con los estándares J2EE.

- **Elección de tecnología:** Los desarrolladores puede elegir entre una diversidad de componentes para implementar cada una de las capas de la aplicación. No obliga a utilizar una sola tecnología, como es el caso de otros *frameworks*.

- **Soluciones de principio a fin:** Oracle ADF es una solución para una capa. Provee soluciones completas, para cada una de las capas de que J2EE especifica y para todas las fases del ciclo de vida del desarrollo de software, desde el diseño hasta la construcción.

1.3. La arquitectura de Oracle ADF

Oracle ADF esta basado en el patrón de diseño Modelo-Vista-Controlador (MVC) y en las mejores prácticas de J2EE. El patrón de diseño MVC separa la arquitectura de la aplicación en tres capas:

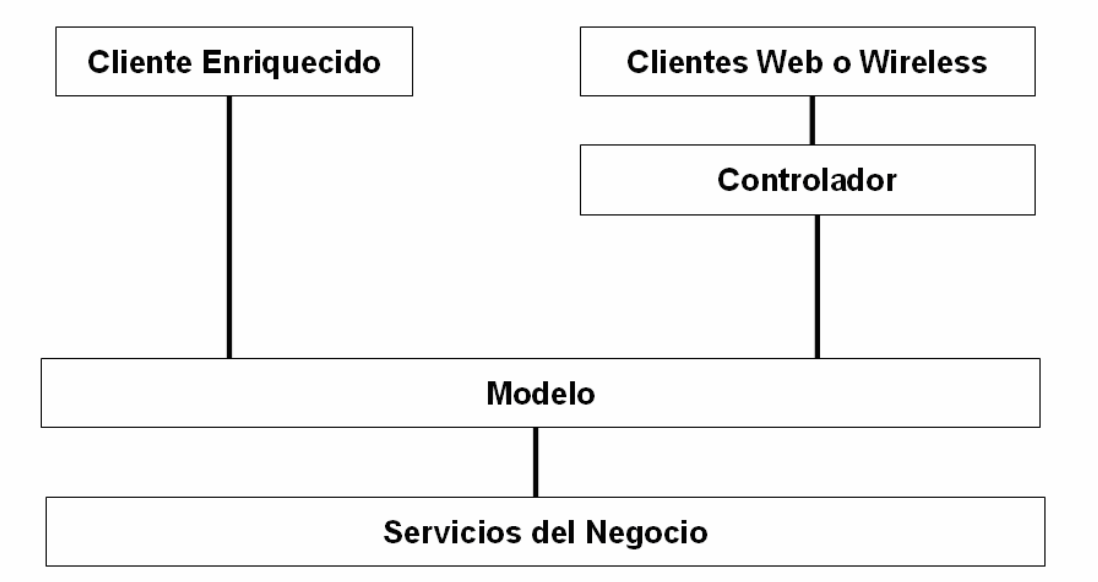
- Modelo, maneja la interacción con las fuentes de datos y ejecuta la lógica del negocio.
- Vista, la cual maneja la interfaz del usuario de la aplicación

- Controlador, en la cual se maneja el flujo de la aplicación y actúa como interfaz entre la capa del Modelo y la Vista

Al Separar las aplicaciones en estas tres capas se simplifica el mantenimiento y se explota la reutilización de componentes entre aplicaciones. Oracle ADF provee una implementación poderosa de MVC, que incrementa la productividad del desarrollo de sistemas.

1.3.1. Las capas de Oracle ADF

Figura 1. Arquitectura general de Oracle ADF



Oracle ADF esta basado en cuatro capas, como se muestra en la figura 1:

- La capa de servicios del negocio, la cual provee de acceso a datos desde diferentes fuentes y maneja la lógica de la aplicación.
- La capa del modelo, provee una capa de abstracción, por encima de la capa de los servicios del negocio, permitiendo que la capa de la vista

y la capa del controlador trabajen con diferentes implementaciones de los servicios del negocio de forma consistente.

- La capa del controlador, provee mecanismos para controlar el flujo de la aplicación.
- La capa de la vista, provee la interfaz del usuario de la aplicación. En la figura 1, se pueden apreciar los diferentes tipos de clientes.

Oracle ADF provee a los desarrolladores la posibilidad de elegir la tecnología con la cual puede implementar cada una de las capas. Sin importar que tecnología se seleccione, el *framework* proveerá las mismas facilidades para la construcción de aplicaciones.

1.4. Fundamentos de Oracle ADF

1.4.1. Conceptos y fundamentos de J2EE

1.4.1.1. Introducción

J2EE es el acrónimo de *Java 2 Enterprise Edition*[1], que traducido literalmente al español quiere decir Java 2 Edición Empresarial.

J2EE es una plataforma que define un estándar para el desarrollo de aplicaciones empresariales multicapa. J2EE simplifica las aplicaciones empresariales basándolas en componentes modulares y estandarizados, proveyendo un completo conjunto de servicios a estos componentes y manejando una parte significativa de la funcionalidad de la aplicación de forma automática, sin necesidad de programación compleja.

La plataforma J2EE aprovecha las características del lenguaje Java y la tecnología involucrada en el desarrollo de aplicaciones utilizando este lenguaje. Por ejemplo, "*Write Once, Run Anywhere*" (WORA), que traducido literalmente quiere decir "Escribir una vez, correrlo en Cualquier Parte", expresión que denota la transportabilidad del lenguaje. Aprovecha la forma en que un programa construido en lenguaje Java puede acceder a una base de datos por medio del JDBC API, la implementación de la tecnología CORBA para la interacción entre los recursos empresariales existentes y los modelos de seguridad que protegen los datos, incluso en aplicaciones Web. Construida sobre esta base, la plataforma *Java 2, Enterprise Edition* provee además, soporte completo a los componentes *Enterprise JavaBeans*, *Java Servlets*, *JavaServer Pages* y tecnología XML.

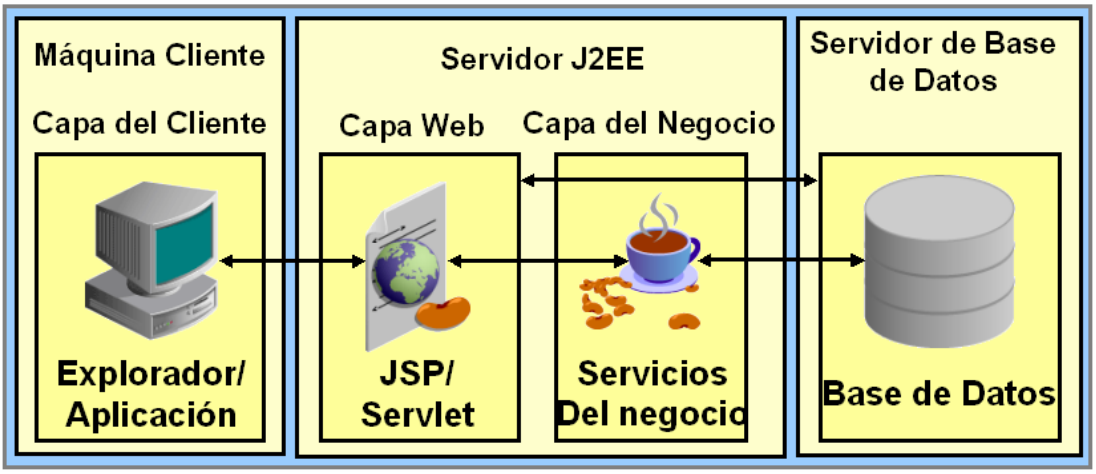
El estándar J2EE incluye una especificación completa que asegura la transportabilidad de la aplicación a través de una amplia gama de sistemas capaces de soportar la plataforma J2EE.

1.4.1.2. Plataforma *Java 2, Enterprise Edition*

J2EE define una plataforma para el desarrollo, publicación y ejecución de aplicaciones en una arquitectura multicapas basada en un modelo distribuido. Esto se refiere a que la aplicación J2EE puede ser divida en componentes, organizados por su funcionalidad y distribuidos en la apropiada capa de la arquitectura multicapa.

La figura 2 muestra el modelo estándar de una aplicación J2EE multicapa, donde la lógica de la aplicación se encuentra distribuida en las siguientes capas:

Figura 2. Arquitectura de la plataforma J2EE



Fuente: Oracle JDeveloper 10g: Build Applicacions with ADF. Oracle University. 2005

- Componentes de la Capa del Cliente, tales como el Explorador Web, ejecutándose en la máquina del cliente.
- La lógica de la presentación es construida en la Capa de los Componentes Web. La presentación puede ser construida con JavaServer Pages (JSP) y Java Servlets se ejecutan en el servidor J2EE. Existe también la posibilidad de implementar la lógica de la presentación a través de clientes Java (Java *clients*), cuando la aplicación que se construye tiene una arquitectura cliente servidor, en este caso el Java *client* se ejecutará en la máquina del cliente.
- La lógica del negocio es implementada en la Capa del Negocio y se ejecuta en el servidor J2EE. Un ejemplo de un componente que reside en esta capa son los Enterprise JavaBeans (EJB).
- Los datos son almacenados en un sistema empresarial de información (EIS, por sus siglas en inglés) que reside en un servidor de base de datos.

1.4.1.3. Beneficios de la plataforma J2EE

Como se mencionó en un principio, J2EE aprovecha las características del lenguaje Java, como lo es la transportabilidad. Esta transportabilidad es asegurada debido a que existe un conjunto de proveedores que soportan las especificaciones J2EE, facilitando la integración de aplicaciones distribuidas.

J2EE separa la lógica de la presentación de la lógica del negocio, permitiendo que las aplicaciones sean más fáciles de administrar y en consecuencia el mantenimiento de dichas aplicaciones es mucho más simple. Existe una suculenta gama de arquitecturas de diseño de donde pueden seleccionar los programadores J2EE para construir sus aplicaciones.

La plataforma J2EE permite la especialización de los desarrolladores por tipo de componente y funcionalidad. Por ejemplo, un desarrollador de EJB puede crear la lógica del negocio, enfocándose en la reglas del negocio para la empresa. Estos EJB son utilizados por los desarrolladores de los componentes Web, quienes se enfocarán en la presentación de los datos y la interfase del usuario. Estos componentes serán integrados por el "Integrador de Aplicaciones", quién será responsable de publicar y administrar la aplicación.

Un "Integrador de Aplicaciones" es una compañía o persona que configura y publica las aplicaciones J2EE. También es una persona que administra la infraestructura de red de cómputo donde las aplicaciones J2EE están ejecutándose.

La especialización de tareas se logra gracias a que la arquitectura J2EE esta divida en múltiples capas y cada una de ellas es independiente de las otras:

- Los diseñadores Web crean los componentes JSP, servlets o JSF
- El comportamiento de aplicación es creado por programadores Java, a través de componentes como los ofrecidos por el *framework* struts
- La lógica del negocio puede ser creada por programadores Java y expertos del negocio.

A continuación se presenta un listado de los beneficios más destacados de utilizar la plataforma J2EE:

- Transportabilidad
- Diferentes proveedores de software que soportan los estándares de la plataforma.
- Integración con otros sistemas a través de estándares.
- Separación de los requerimientos del cliente de la lógica del negocio.
- Diversidad de patrones de diseño.
- Separación de las tareas de desarrollo en habilidades específicas.

1.4.1.4. Arquitectura J2EE

La plataforma J2EE es una arquitectura basada en componentes para el desarrollo y publicación de aplicaciones empresariales. Un componente es una unidad de software reutilizable que se encuentra en un nivel de la aplicación, estos componentes pueden ser fácilmente actualizados conforme el negocio cambie.

Existen diferentes tipos de componentes J2EE, en el presente documento se concentra únicamente en los más comúnmente utilizados,

aquellos que pertenezcan a la capa Web y la capa del Negocio. Estos componentes son reutilizables, esto es, están disponibles para que sean utilizados por otros componentes o aplicaciones.

La arquitectura J2EE provee un ambiente basado en componentes e independiente de plataforma, en donde cada uno de estos componentes es ensamblado y publicado en su correspondiente contenedor. El servidor J2EE provee servicios para manejar transacciones, controlar el estado de sesiones, ejecución en paralelo (*multithreading*), pool de recursos, y otros detalles complejos de bajo nivel, en la forma de contenedores. Un contendedor es un ambiente de ejecución que implementa la especificación J2EE para la ejecución de aplicaciones.

Debido a que los servicios subyacentes que forman parte de la ejecución de una aplicación J2EE son provistos por el contenedor del servidor, los desarrolladores pueden concentrarse en resolver los problemas del negocio.

Los componentes J2EE se comunican el uno con el otro utilizando diferentes APIs. La arquitectura J2EE provee los siguientes APIs:

- **Java Naming and Directory Interface (JNDI):** Provee la funcionalidad de ubicar recursos a través de un directorio de nombres.
- **Remote Method Invocation (RMI):** Permite crear una aplicación Java distribuida, en la cual los métodos de un componente remoto pueden ser invocado por otros contenedores J2EE, posiblemente en diferentes máquinas.
- **Java Database Connectivity (JDBC):** Permite acceder cualquier fuente de datos, tal como una base de datos, hojas de cálculo o archivos planos desde una aplicación construida en lenguaje Java.

- **Java Transaction API (JTA):** Especifica una interfase Java estándar entre un administrador de transacciones y los participantes involucrados en un sistema de transacción distribuida: El administrador de recursos, el servidor de aplicaciones y la aplicación transaccional.
- **Java Messaging Services (JMS):** Permite a las aplicaciones crear, enviar, recibir y leer mensajes, al definir un conjunto común de interfases y semánticas asociadas que permiten a los programas escritos en el lenguaje de programación Java, comunicarse con otras implementaciones de mensajería.
- **JavaMail:** Provee un conjunto de clases abstractas que modelan un sistema de correo electrónico. Este API provee un framework para envió de correo electrónico independiente de plataforma y protocolo, basado en tecnología Java y aplicaciones de mensajería.
- **JavaBeans Activation Framework (JAF):** Provee servicios estándares para determinar el tipo de una corriente de datos (*data streams*), para encapsular el acceso a la misma, para descubrir las operaciones disponibles sobre esta y para instanciar el apropiado objeto que realiza dichas operaciones. Por ejemplo, el explorador Web obtiene una imagen JPEG, este *framework* permite al explorador identificar esa corriente de datos como una imagen JPEG. Luego el explorador Web puede localizar e instanciar un objeto que puede manipular o presentar la imagen.

1.4.1.4.1. Componentes de la capa del cliente

1.4.1.4.1.1. Cliente *Web*

Para una aplicación J2EE para Internet, el Explorador Web del usuario es el componente de la capa del cliente. El Explorador Web transfiere las páginas Web estáticas o dinámicas desde la capa Web del servidor J2EE a la máquina del cliente.

Los clientes Web típicamente son clientes livianos, los cuales no realizan operaciones como ejecutar reglas del negocio complejas, conectarse ó consultar la base de datos. Estas operaciones son realizadas generalmente por los componentes de la Capa del Negocio.

1.4.1.4.1.2. Aplicación cliente

Es una aplicación cliente ejecutada desde la máquina cliente para una aplicación no basada en Web. Puede contener una interfaz gráfica para el usuario (GUI, por sus siglas en inglés) creada con tecnología Swing, AWT o ser simplemente una interfaz de línea de comandos. Las aplicaciones cliente pueden acceder directamente los componentes de la capa de la lógica del negocio o pueden acceder servlets que estén ejecutándose en la capa Web a través de una conexión http.

1.4.1.4.2. Componentes de la capa del *Web*

Los componentes J2EE de la capa del cliente pueden ser tanto servlets como JSPs que pueden generar páginas HTML estáticas o dinámicas, páginas

WML (Lenguaje de Etiquetas Wireless) ó páginas XML (Lenguaje de Etiquetas Extendido).

Los JavaServlets proveen un API simple y poderoso para generar páginas Web al procesar dinámicamente las solicitudes de un cliente y construyendo una respuesta. Las páginas JSPs, simplifican el proceso de generar contenido dinámico, utilizando Java, como un lenguaje embebido, en páginas HTML. Las páginas JSPs son traducidas y compiladas en Java servlets que entonces son ejecutadas en el servidor Web como cualquier otro servlet.

Algunas de las ventajas de utilizar componentes Web son:

- La interfaz HTML que es generada por los componentes Web es liviana cuando se compara con los applets, los cuales requieren descargas pesadas hacia la máquina de los clientes para poder ser ejecutadas.
- La interfaz HTML no requiere ninguna preinstalación en las máquinas del cliente.
- El protocolo HTTP, por medio del cuál el cliente hace una solicitud por una página Web, cumple con las restricciones de casi todos los firewalls.

1.4.1.4.2.1. *Servlets*

Un servlet es un programa en Java que reside en un servidor de aplicaciones y que produce páginas dinámicas, en respuesta a una solicitud del cliente. Las páginas luego son enviadas de vuelta al Explorador Web del cliente. Regularmente, el formato de salida para los servlets son páginas

HTML, pero cualquier otro formato de salida, incluyendo XML, puede ser utilizado.

1.4.1.4.2.2. *JavaServer Pages*

Las páginas JSPs son servlet escritos de forma distinta. La tecnología JavaServer Pages separa la interfaz del usuario del contenido generado dinámicamente, de tal forma que los diseñadores pueden cambiar fácilmente la distribución y estilo de la página sin alterar el contenido generado dinámicamente. La tecnología JSP soporta un diseño basado en componentes reutilizable, facilitando y acelerando la construcción de aplicaciones Web.

Una característica de la páginas JSPs es simplificar la generación de código HTML a través del uso de etiquetas similares a las de HTML (JSP tag libs). Al separar el diseño de la presentación, de la lógica de la aplicación que genera los datos, las páginas JSPs permiten a las organizaciones reutilizar y compartir la lógica de las aplicaciones a través de etiquetas personalizadas y componentes. Esto también separa las responsabilidades de trabajo de los diseñadores Web de las responsabilidades de los programadores Java.

1.4.1.4.3. Componentes de la capa del negocio

Los Enterprise JavaBeans (EJB) son los componentes que se ejecutan en la Capa del Negocio. Estos componentes distribuidos contienen las reglas del negocio que conocen las necesidades de un dominio particular del mismo. Los componentes de la Capa del Negocio pueden recibir datos desde programas clientes, procesarlos y enviarlos procesados hacia la base de datos para que sean almacenados. Los componentes del la Capa del

Negocio pueden también obtener datos desde la base de datos, procesarlos y enviarlos a los programas clientes.

1.4.1.4.3.1. *Enterprise JavaBeans* (EJB)

Enterprise JavaBeans (EJB) es una arquitectura para desarrollar aplicaciones transaccionales como componentes distribuidos en Java. Cuando una aplicación es desarrollada con enterprise beans, el desarrollador de los beans y el programador de las aplicaciones clientes no tienen que preocuparse por detalles de bajo nivel, tal como soporte de transacciones, seguridad, acceso remoto a objetos. Estos detalles son provistos de forma transparente para los desarrolladores por el contendor de EJB, mismo que reside en el contenedor J2EE.

Los EJBs ofrecen transportabilidad. Un bean que es desarrollado en un contenedor EJB puede ejecutarse en otro contenedor EJB que siga las especificaciones EJB. La arquitectura EJB especifica *Remote Method Invocation* (RMI) como el protocolo de transporte. Los EJBs acceden a la base de datos a través de JDBC.

Los EJBs son accedidos utilizando el framework RMI de Java, que puede ser implementado con diferentes protocolos de red.

1.4.2. Patrones de diseño

1.4.2.1. Introducción a patrones de diseño

Como analistas y programadores se desarrollan a diario las habilidades para resolver problemas usuales que se presentan en el desarrollo del software. Por cada problema que se presenta el desarrollador

debe pensar distintas formas de resolverlo, incluyendo soluciones exitosas que ya se han utilizado anteriormente en problemas similares. Es así que a mayor experiencia que un desarrollador tenga, el abanico de posibilidades para resolver un problema crece, pero al final siempre habrá una sola solución que mejor se adapte a la aplicación que se está tratando de construir. Si se documentan estas soluciones, se pueden reutilizarlas y compartir esa información que se ha aprendido para resolver de la mejor manera un problema específico.

Los patrones del diseño tratan los problemas del diseño que se repiten y que se presentan en situaciones particulares del diseño, con el fin de proponer soluciones a ellas. Por lo tanto, los patrones de diseño son soluciones exitosas a problemas comunes. Existen muchas formas de implementar patrones de diseño. Los detalles de las implementaciones son llamadas estrategias.

Un patrón de diseño es una solución de diseño de software a un problema, aceptada como correcta, a la que se ha dado un nombre y que puede ser aplicada en otros contextos.

Un patrón de diseño es una abstracción de una solución en un alto nivel. Los patrones solucionan problemas que existen en muchos niveles de abstracción. Hay patrones que abarcan las distintas etapas del desarrollo; desde el análisis hasta el diseño y desde la arquitectura hasta la implementación.

Muchos diseñadores y arquitectos de software han definido el término de patrón de diseño de varias formas que corresponden al ámbito a la cual se aplican los patrones. Luego, se dividieron los patrones en diferentes categorías de acuerdo a su uso.

Los diseñadores de software extendieron la idea de patrones de diseño al proceso de desarrollo de software. Debido a las características que proporcionaron los lenguajes orientados a objetos (como herencia, abstracción y encapsulamiento) les permitieron relacionar entidades de los lenguajes de programación a entidades del mundo real fácilmente. Los diseñadores empezaron a aplicar esas características para crear soluciones comunes y reutilizables para problemas frecuentes que exhibían patrones similares.

1.4.2.2. Estructura de los patrones de diseño

Usualmente los patrones de diseño se describen con la siguiente información:

- Descripción del problema: Que permitirá el patrón y ejemplos de situaciones del mundo real.
- Consideraciones: Que aspectos fueron considerados para que tomara forma la solución.
- Solución general: Una descripción básica de la solución en si misma.
- Consecuencias: Cuales son las ventajas y desventajas de utilizar esta solución.
- Patrones relacionados: Otros patrones con uso similar que deben ser considerados como alternativas.

1.4.2.3. Tipos de patrones de diseño

Existen varias clasificaciones de los patrones de diseño, pero más que categorías, las diferentes clasificaciones se basan en subcategorías de la clasificación basada en su propósito: creacionales, estructurales y de comportamiento.

- **De creación:** Los patrones de creación tratan con las formas de crear instancias de objetos. El objetivo de estos patrones es de abstraer el proceso de instanciación y ocultar los detalles de cómo los objetos son creados o inicializados.
- **Estructurales:** Los patrones estructurales describen como las clases y objetos pueden ser combinados para formar grandes estructuras y proporcionan nuevas funcionalidades. Estos objetos adicionados pueden ser incluso objetos simples u objetos compuestos.
- **Comportamiento:** Los patrones de comportamiento nos ayudan a definir la comunicación e iteración entre los objetos de un sistema. El propósito de este tipo de patrón es reducir el acoplamiento entre los objetos.

El patrón MVC esta clasificado dentro de los patrones de diseño estructurales.

1.4.2.4. Patrón Modelo-Vista-Controlador (MVC)

1.4.2.4.1. Introducción

El MVC es un patrón de diseño compuesto debido a que tiene sus bases en varios patrones. El resultado es un framework que identifica componentes en una aplicación en base a sus diferentes funciones. Si el

cliente es un Explorador en una aplicación J2EE, la arquitectura Modelo Vista Controlador ayuda a separar la lógica del negocio de la lógica de la presentación dentro de la aplicación.

Al separar la funcionalidad de la aplicación de esta manera, se minimiza la necesidad de modificar la lógica del negocio, por ejemplo, si una columna de una tabla almacenada en la base de datos cambia o si la interfaz del usuario debe ser adaptada para un Explorador, un PDA o un celular. Esto reduce la duplicación de código y minimiza el mantenimiento.

En casi cualquier programa que se construya se pueden encontrar tres partes bien diferenciadas. Por un lado se tiene el problema que se trata de resolver. Este problema suele ser independiente de cómo se quiere que el programa recoja los resultados o cómo se requiere que los presente. Por ejemplo, si se quiere hacer un juego de ajedrez, todas las reglas del ajedrez son totalmente independientes de si se va a dibujar el tablero en 3D o plano, con figuras blancas y negras tradicionales o figuras modernas de cómo robots plateados y personajes famosos. Este código constituiría el modelo.

En el juego del ajedrez el modelo podría ser una clase (o conjunto de clases) que mantengan un arreglo de 8x8 con las piezas, que permita mover dichas piezas verificando que los movimientos son legales, que detecte los jaques, jaque mate, tablas, etc. De hecho, las metodologías orientadas a objetos introducen este tipo de clases, a las que llaman clases del negocio.

Otra parte clara, es la presentación visual que se quiere hacer del juego. En el ejemplo del ajedrez serían las posibles interfaces gráficas mencionados en el punto anterior. Esta parte del código es la vista. Se denomina interfaz gráfica, por ser lo más común, pero podría ser de texto, de comunicaciones con otro programa externo, con la impresora, etc. Se

representaría, por ejemplo, la ventana que dibuja el tablero con las figuras de las piezas, la posibilidad de arrastrar con el ratón una pieza para moverla, botones, lista visual con los movimientos realizados, etc.

La tercera parte de código es aquel código que toma decisiones, algoritmos, etc. Es código que no tiene que ver con las ventanas visuales ni con las reglas del modelo. Esta parte del código es el controlador. En el código de ajedrez formaría parte del controlador el algoritmo para pensar las jugadas (el más complejo de todo el juego).

1.4.2.4.2. Arquitectura Modelo-Vista-Controlador

La arquitectura MVC es el patrón de diseño más popular para aplicaciones para Internet. MVC separa la aplicación en agrupaciones lógicas de acciones o actividades. La siguiente separación de actividades, permite que el patrón de diseño MVC, sea aprovechado al construir aplicaciones para Internet:

- **Modelo:** El Modelo maneja toda la persistencia de los datos y la lógica del negocio. Esto asegura que, si la capa de persistencia o la lógica del negocio cambian, las otras partes de la aplicación pueden continuar sin sufrir cambios.

- **Vista:** La Vista maneja toda la presentación de los datos obtenidos por el Modelo. Debido a que la vista se comunica con el modelo de una manera específica, la vista puede ser cualquier vista J2EE (JSPs, aparatos móviles y otros).

- **Controlador:** El controlador maneja el flujo de la aplicación. Es independiente de la Vista, de tal forma que si el flujo de la aplicación

necesita cambiar, las vistas no sufrirán cambios. El controlador también interpreta los eventos de la aplicación y puede enviar a un usuario a una Vista específica basado en dicho evento.

1.4.2.4.3. Estructura MVC

Como se ha descrito en la sección anterior, el patrón de diseño MVC está compuesta por tres partes que trabajan en conjunto, cada una de estas partes maneja una distinta actividad dentro de la aplicación.

En la figura 3, se puede apreciar el flujo de trabajo de una aplicación construida sobre el modelo MVC. Se puede apreciar al cliente como entidad o proceso que realiza una petición a la aplicación. Está solicitud es enviada al controlador, el cual decide quien puede responder a dicha solicitud de mejor forma. El modelo implica la lógica del negocio y es administrada por el controlador, quien por medio de envío y recepción de parámetros intercambia los datos necesarios para satisfacer la solicitud del cliente.

La respuesta que da el modelo es enviada a través del controlador a la vista y es este componente el que se encarga de presentar los datos de respuesta al cliente de la manera más adecuada.

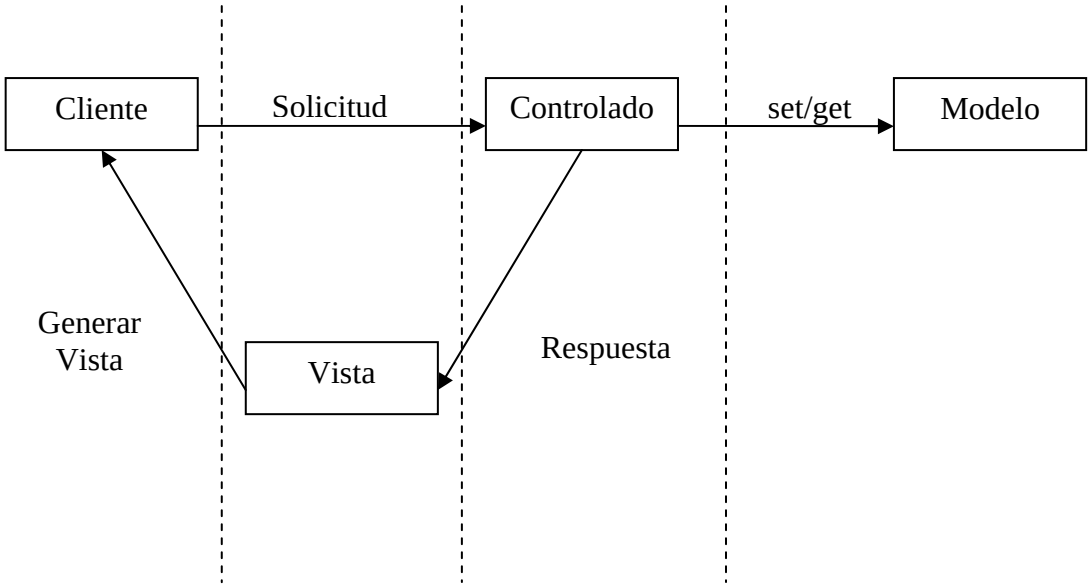
Las actividades que realizan cada uno de los componentes del modelo MVC son listadas continuación:

- **Controlador:**
  - Manejar el ruteo a la página correcta (flujo de la aplicación)
  - Mapear los cambios que se hagan a los datos en la interfaz del usuario hacia el modelo

- Transferir datos desde y hacia el modelo

- **Vista**
  - Administrar la interfaz con el usuario
  - Solicitar datos al modelo
  - Enviar eventos al modelo
  - Permitir al controlador seleccionar la siguiente vista
- **Modelo**
  - Almacenar el estado de la aplicación
  - Responder a las solicitudes de datos
  - Encapsular la lógica del negocio

Figura 3. Estructura del Modelo MVC



2. CAPA DE PRESENTACIÓN (VISTA)

2.1. Definición de la vista

La Vista es parte de la arquitectura MVC, corresponde a la interfaz del usuario. Es la parte que el usuario ve y por medio de la cual interactúa con la aplicación. Los componentes que corresponden a la Vista, interactúan con el controlador para obtener los datos del Modelo. Es importante notar que la vista no contiene ninguna lógica de la aplicación; simplemente muestra los resultados de las operaciones que se han realizado en la capa del Modelo y responde a las solicitudes que haga al controlador.

En la programación tradicional, es decir, aquella en la que este patrón de diseño no se ha implementado, la Vista contenía casi toda la aplicación. La Vista administraba el acceso a los datos, controlaba las solicitudes de los clientes, manejaba la navegación de la aplicación, y realizaba cualquier tarea que fuera necesaria como parte de la aplicación. Lo anterior prueba lo difícil que era escribir y dar mantenimiento, implicando que la aplicación fuese poco flexible, de tal forma que, cuando las necesidades del negocio cambiaban, era necesario reescribir el código.

Los principales conceptos que se deben tener en cuenta sobre la Vista, se resumen a continuación:

- La Vista no contiene código de la aplicación; únicamente contiene código para representar la interfaz del usuario y trasladar eventos al Controlador.
- La Vista es intercambiable sin necesidad de sobrescribir el código que corresponde al controlador y a la lógica del modelo.

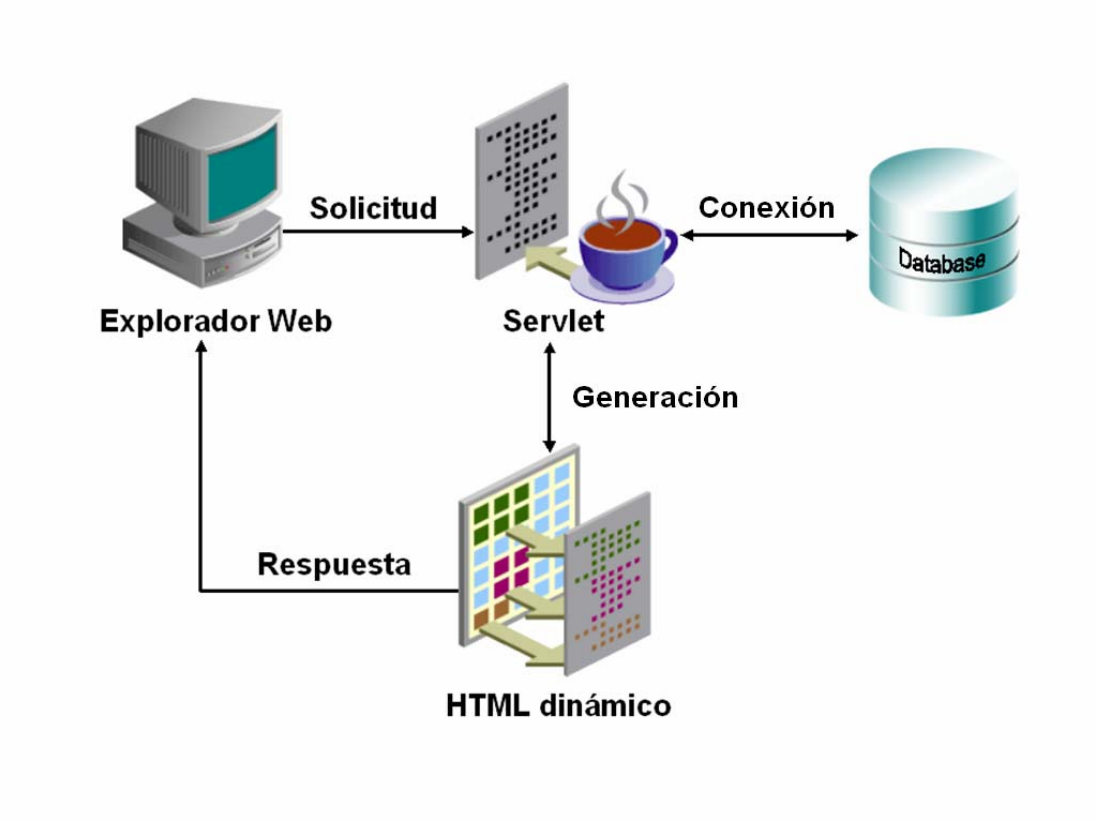
- Una simple aplicación puede tener diferentes Vistas que sean compatibles con diferentes tipos de aparatos electrónicos (exploradores HTML, celulares, etc.)

2.2. Componentes J2EE que corresponden a la capa de la vista

2.2.1. *Servlets*

Los conceptos básicos de los servlets son simples. Un explorador Web invoca un URL, que invoca al servlet. El servlet genera dinámicamente HTML, XML u otro tipo de contenido. El texto dinámico puede generarse después de conectarse a la base de datos (ver figura 4).

Figura 4. Generación del contenido dinámico de un *servlet*



Fuente: Oracle JDeveloper 10g: Build Applicacions with ADF. Oracle University. 2005

Un servlet es un programa de Java que implementa el API servlet. Estas clases se encuentran en el paquete *javax.servlet*. Cuando se utiliza el API servlet, este se encarga de una serie de tareas comunes que se deben realizar cuando se atiende una solicitud por parte de un cliente. Por ejemplo, el API soporta preinstanciación de objetos en la Máquina virtual de Java si múltiples clientes deben acceder simultáneamente a una función especial que se provee en el servlet. Esto es conocido como el ciclo de vida del servlet.

Un servlet se ejecuta en el contexto de un proceso especial que se conoce como motor servlet (*servlet engine*). Un servlet puede ser invocado simultáneamente por múltiples clientes, debido a que los métodos del servlet corren en *threads* (hilos). Es decir, un servlet implementa la ejecución en paralelo, por lo que las instancias de los servlets pueden atender múltiples solicitudes.

2.2.1.1. Ciclo de vida de un *servlet*

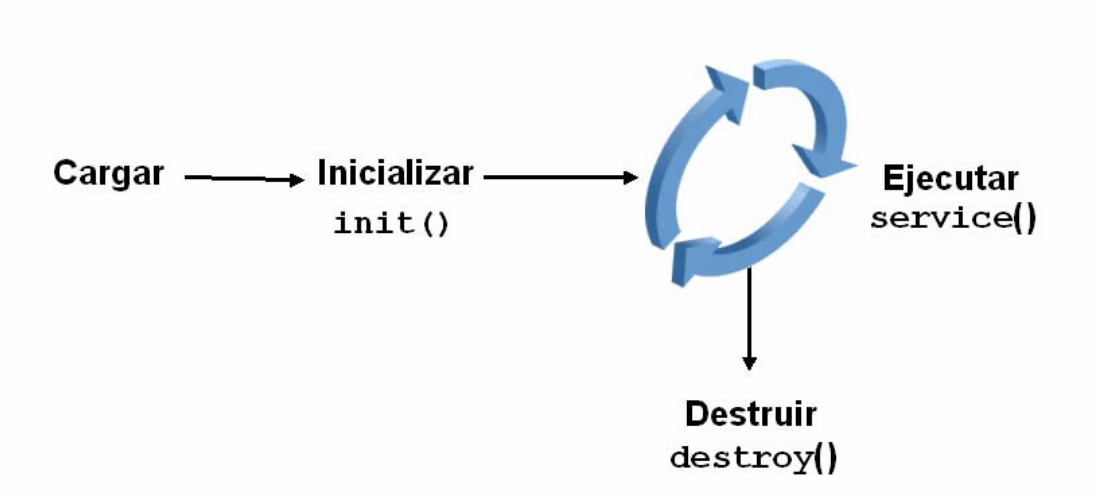
El ciclo de vida de los servlets corresponde a que la primera vez que se invoca al servlet, este se carga en memoria y se ejecuta el método init() que corresponde a la interfaz Servlet.

Luego de eso, el servlet queda instanciado en memoria para atender solicitudes que se hagan a través de la ejecución del método service() que corresponde a la misma interfaz.

Por último, cuando el servlet ha estado por un tiempo sin recibir solicitudes, el *servlet engine*, libera la memoria que correspondía al servlet por lo que se ejecuta el método destroy() que se haya implementado en el servlet.

La figura 5 muestra gráficamente el ciclo de vida descrito anteriormente.

Figura 5. Ciclo de Vida del *Servlet*



Fuente: Oracle JDeveloper 10g: Build Applicacions with ADF. Oracle University. 2005

El ciclo de vida del servlet tiene las siguientes características:

- Muchos motores de servlets pueden ejecutarse dentro de una misma Máquina Virtual de Java.
- Los servlets persisten entre solicitudes como una instancia de objetos. Si una instancia realiza una conexión a la base de datos, entonces no existe la necesidad de abrir otra conexión para una segunda solicitud.
- Se puede sobrescribir el método init(), que es invocado por el *servlet engine* al momento instanciar el servlet y el método destroy() que es invocado al momento de destruir la instancia del servlet para realizar tareas personalizadas.

2.2.1.2. HTTP *servlets*

Un HTTP servlet hereda de la clase javax.servlet.http.HttpServlet, que implementa la interfaz Servlet. Este tipo de servlet provee características que permiten mejorar la velocidad de respuesta al cliente. Por ejemplo, los parámetros GET que se transfieren, a través de un explorador Web hacia el servidor Web están disponibles para ser utilizados dentro del servlet API. De forma similar, los datos enviado a través de formas HTML son enviados al servlet utilizando el método doPost().

Cuando un HTTP servlet es invocado, el *servlet engine* invoca al método service() en el servlet. Esto es posible debido a que se ha implementado la interfaz Servlet. Si el Explorador Web ha invocado el método GET del protocolo HTTP, entonces el método service() invocará al método doGet() del servlet. La funcionalidad que la aplicación debe proveer se consigue sobrescribiendo este método. El método GET recibe dos parámetros que corresponden al cliente que ha efectuado la solicitud y a la salida donde se generará el código dinámico.

De manera similar, si el explorador invoca al método POST en el protocolo http (por ejemplo, cuando el usuario enviar una forma HTML), el método service() invoca al método doPost().

2.2.1.2.1. Ejemplo de un *servlet*

El código que se presenta a continuación en un simple servlet que imprime en texto sin formato el mensaje “Hola Mundo”.

```
import javax.servlet.*;
import javax.servlet.http.*;
```

```
import java.io.*;
public class SimplestServlet extends HttpServlet
{
        public void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
        {
                PrintWriter out = response.getWriter();
                out.println("Hello World");
        }
}
```

2.2.1.2.2. El método doGet()

El método doGet(), que recibe dos parámetros como entrada, es usualmente el primer método que se debe modificar de la clase HttpServlet. Es además, el más común método de solicitud del protocolo HTTP. El método doGet() recibe los parámetros HttpServletRequest y HttpServletResponse[2].

Los parámetros del objeto que ha realizado la solicitud del servlet son trasladados al método doGet(), agregándolos al URL con el que se está invocando el servlet, por ejemplo:

- http://www.misitio.com/servlet?param1=valor
- http://www.misitio.com/servlet?param1=valor&param2=valor

Una solicitud GET del protocolo HTTP es generada por el explorador cuando ocurre una de las siguientes situaciones:

- El usuario ingresa un URL en la línea de dirección de un Explorador.
- El usuario sigue un vinculo (HREF) desde otra página.
- El usuario envía una forma HTML que no especifica el método de solicitud.
- Cuando el método especificado para la solicitud del envío de una forma es GET (FORM METHOD=”GET”)

2.2.1.2.3. El método doPost()

El método doPost() es utilizado en colaboración con una forma HTML. Cuando el cliente presiona el botón de envío dentro de una forma, cualquier parámetro que esta incluido dentro de la forma es enviado al servlet que es especificado en la etiqueta *action* de dicha forma.

El método doPost(), al igual que el método doGet(), también recibe dos parámetros, un objeto de tipo HttpServletRequet y el HttpServletResponse.

Los parámetros son enviados al servidor Web en una cabecera adicional a la solicitud HTTP, y no son agregados al URL del servlet, como es el caso del método GET, las ventajas de utilizar POST como método de solicitud se listan a continuación:

- Los parámetros (como las contraseñas) no son visibles en el campo de la dirección URL del explorador.
- No se puede almacenar o copiar el URL conteniendo los parámetros.
- Los servidores Web usualmente limitan la cantidad de caracteres que pueden ser enviados agregados a un URL (de 2

a 4 kilobytes), pero no hay un límite teórico para el tamaño de los parámetros POST

El método doPost() es invocado en un servlet cuando una forma HTML especifica su etiqueta de la siguiente forma:

- <form method="post" action=....>

Los parámetros enviados al método doPost(), utilizan las etiquetas HTML que sirven para interactuar con el usuario, por ejemplo:

- <input type="text" name="param1">
- <input type="hidden" name="param1" value="opcion"/>
- <input type="checkbox" name="param1" value="opcion"/>
- <input type="password" name="param1"/>
- <input type="radio" name="param2" value="opcion1"/>
  <input type="radio" name="param2" value="opcion2"/>

2.2.1.2.4. El objeto HttpServletRequest

El objeto HttpServletRequest, encapsula la siguiente información acerca del cliente que invocó al servlet:

- Los parámetros del Servlet, tanto valores como nombre.
- El nombre del host que efectuó la solicitud del servlet.
- El nombre del servidor que recibió la solicitud

Los métodos[3] utilizados para acceder a la información antes descrita son:

- getParameter(String nombre), que permite obtener el valor del parámetro que es enviado por el cliente que solicitó el servlet y cuyo nombre se envía como parámetro a este método.
- getRemoteHost(), para obtener el nombre del Host remoto.
- getServerName(), para obtener el nombre del servidor que recibió la solicitud.
- getRequestURI(), para obtener el URL que provocó la solicitud del servlet, sin incluir el nombre del servidor, es decir, el directorio virtual raíz y el nombre del servlet.

La tabla I presenta el valor que retornaría una serie de métodos de este objeto, si se hiciera la solicitud a través del siguiente URL: http://mimaquina:80/midirectorio/serlvet/miservlet?param1=valor1

Tabla I. Valores de retorno de métodos de HttpServletRequest

Método de Solicitud	Valor de Retorno
getServerName()	mimaquina
getServerPort()	80
getPathInfo()	/midirectorio/servlet/miservlet.class
getContextPath()	/midirectorio
getRequestURI()	/midirectorio/servlet/miservlet

2.2.1.2.5. El Objeto HttpServletResponse

El segundo parámetro para los métodos doGet() y doPost() es el objeto HttpServletResponse[4]. Una respuesta HTTP consta de una línea de estado (sin importar si la llamada al servlet fue un éxito o no), uno más encabezados (tipo de contenido a generar), y la salida actual.

2.2.1.3. Un ejemplo de integración entre *servlets*

El siguiente código construye una página HTML, con una forma que invocará al servlet HolaMundo, y que envía un parámetro, cuyo nombre es "apellido" y su valor deberá ser proporcionado por el cliente.

```
<html><body>
   <form action="holamundo" method="POST">
     Por favor ingrese su apellido, gracias.
     <input type="text" name="apellido"/>
     <input type="submit" value="Enviar Apellido"/>
   </form>
 </body></html>
```

La página Web se vería como se muestra en la figura 6.

Figura 6. Forma HTML para ingreso de parámetros

El código del servlet HolaMundo es presentado a continuación:

```
public class HolaMundo extends HttpServlet
{
public void doPost(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException
 {
   res.setContentType("text/html");
   PrintWriter out = res.getWriter();
   out.println("<html>");
   out.println("<body>");
   String apellido = req.getParameter("apellido");
   if ((apellido != null)&&(apellido.length() > 0))
```

```
    {
      out.println("Hola: " + apellido + ", Como estas?");
    }
  else
    {
      out.println("Hola: Anónimo");
    }
  out.println("</body></html>");
  out.close();
 }
}
```

Este servlet será invocado desde la forma HTML antes descrita. Una vez ejecuta el método doPost() obtendrá el valor del elemento input que se incluyo dentro de la forma y evaluará el contenido de dicho parámetro para presentar una bienvenida con el apellido o una bienvenida a un usuario anónimo, si no se ingresaron valores como parámetros de la forma.

Salida con un valor para el parámetro apellido de “MiApellido”:

```
<html><body>
Hola: MiApellido, ¿Como estás?
</body></html>
```

Salida con un valor para el parámetro apellido con valor nulo:

```
<html><body>
Hola: Anónimo
</body></html>
```

2.2.2. *JavaServer Pages*

La idea detrás de la tecnología servlet y JavaServer Pages es separar la lógica del negocio de la presentación y de esa manera, proporcionar clientes livianos. Las páginas JSPs, están basadas en la tecnología servlet y son una extensión de los mismos. Las páginas JSP, pueden generar contenido dinámico al igual que los servlet, sin embargo, una página JSP tiene sus propias ventajas.

La ventaja principal del modelo JSP es, que los diseñadores Web, no necesitan estar familiarizados con el lenguaje de programación Java para crear las páginas JSP. Los programadores pueden proveer JavaBeans y etiquetas personalizadas para los diseñadores Web, que esta familiarizados con HTML. En éste modelo, se definen adecuadamente los roles de trabajo de los diseñadores de páginas Web y los programadores de la aplicación.

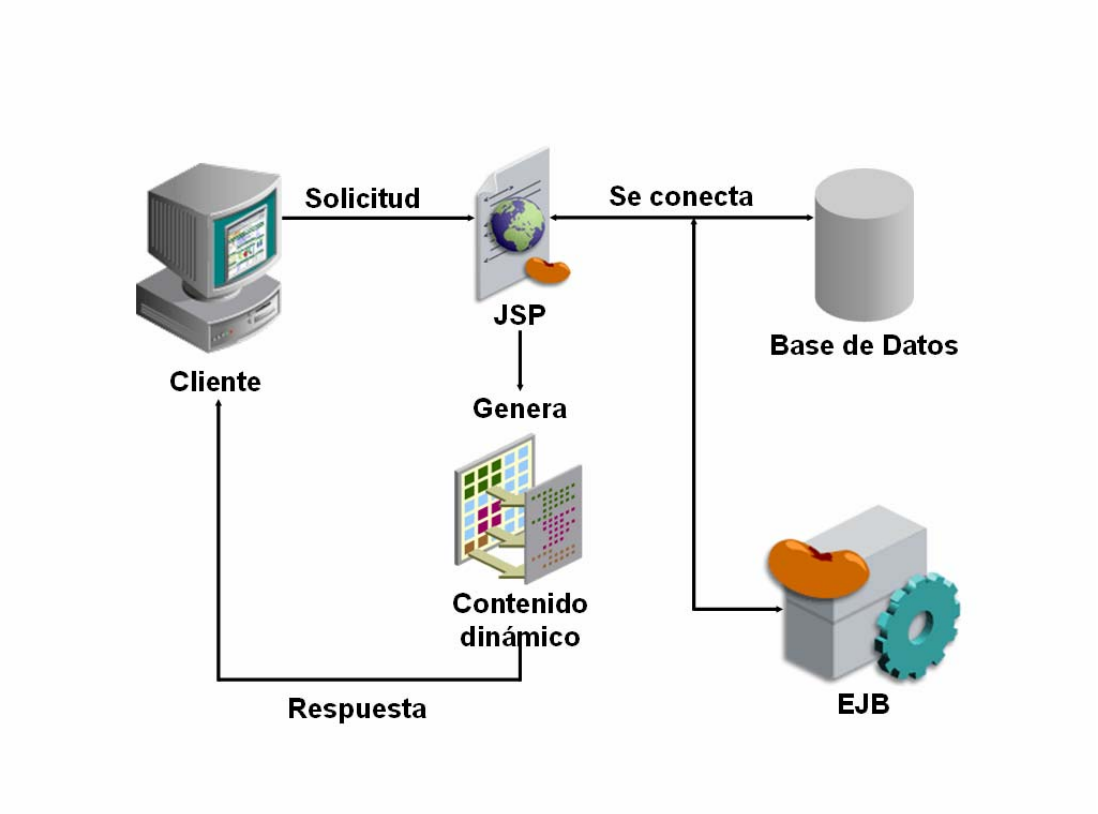
Siendo las páginas JSP construidas sobre la tecnología servlet, el principal objetivo de estos componentes es mejorar la productividad de los programadores.

Como se puede apreciar en la figura 7, el cliente invoca la página JSP desde un explorador Web a través de una dirección URL. Se debe notar que, las páginas JSP utilizan el mismo modelo de solicitud/respuesta de los serlvets. Dependiendo del tipo de solicitud por parte del cliente. La página JSP puede conectarse hacía la base de datos, o bien, invocar a un Enterprise JavaBean, el cual a su vez, tiene la posibilidad de conectarse hacía una base de datos. La página JSP, luego crear el contenido dinámico, utilizando información obtenida de la base de datos o del EJB, retorna el resultado hacia el cliente.

2.2.2.1. Comparación entre un *servlet* y una página JSP

Una página JSP es una página HTML con código Java embebido. Una página JSP toma una página HTML, le agrega etiquetas, código de Java y a continuación genera código dinámico. Una clase servlet, también genera contenido dinámico, pero no es una página HTML. Un servlet es un programa en Java que tiene embebido código HTML. Debido a que un servlet es un programa en Java, el programador debe tener cuidado de la sintaxis y semántica cuando esta desarrollando el servlet. El servlet debe se compilado antes de ser ejecutado.

Figura 7. Generación de contenido dinámico de una JSP



Fuente: Oracle JDeveloper 10g: Build Applicacions with ADF. Oracle University. 2005

Mientras que se debe tener un gran conocimiento de lenguaje de programación Java y del API para desarrollar servlets, no se necesita ser un experto en Java para desarrollar una página JSP. Un diseñador de páginas Web puede desarrollar una página JSP, debido a que una página JSP

principalmente contiene etiquetas HTML y algunas etiquetas JSP que contienen código de Java.

Debido a que un servlet es una programa en Java, el archivo .class debe ser creado antes de que se pueda invocar. Una página JSP es compilada automáticamente y transformada en un servlet cuando es invocada la primera vez. No se necesita que explícitamente se compile el código.

2.2.2.2. Invocando una página JSP

Se puede invocar una JSP de diferentes formas, dependiendo de las necesidades de la aplicación:

- Invocando la página JSP con una dirección URL, de la forma http://servidor:puerto/directorio-raiz/pagina.jsp, donde:
  - servidor, es el nombre o la dirección IP de la máquina en la cual la página JSP reside.
  - puerto, es el número de puerto en el cual el servidor está atendiendo solicitudes.
  - directorio-raiz, es el nombre con el cual el documento esta disponible para el cliente.
  - pagina.jsp, es el nombre del archivo JSP
- Se pude invocar la página JSP desde una forma HTML, un servlet u otra página JSP.

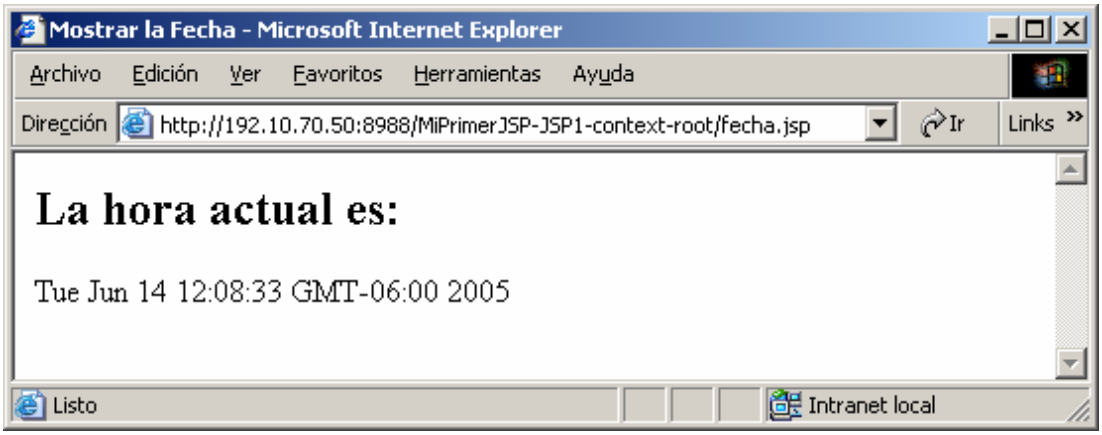
2.2.2.3. Ejemplo de una página JSP

El siguiente código es el de una página JSP que muestra la hora en la cuál fue invocada.

```
<%@ page contentType="text/html;charset=windows-1252"%>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
    <title>Mostrar la Fecha</title>
  </head>
  <body>
  <h2>  La hora actual es: </h2>
  <p> <%= new java.util.Date() %> </p>
  </body>
</html>
```

La salida del código anterior es presentada en la figura 8.

Figura 8. Salida de página JSP ejemplo



La misma funcionalidad, escrita con un servlet se vería de la siguiente forma:

```
...
public void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException
  {
    response.setContentType(CONTENT_TYPE);
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head><title>Mostrar Fecha</title></head><body><h2>La hora actual
                es:</h2><p>");
```

```
    out.println(new java.util.Date());
    out.println("</p></body></html>");
    out.close();
  }
...
```

2.2.2.4. Ciclo de vida de una página JSP

Cuando el servidor Web recibe una solicitud, el servidor descifra la solicitud para determinar si la solicitud es para una página JSP. El motor JSP verifica si el servlet que corresponde a la página JSP ya existe.

Si el servlet no existe, se establece que es la primera vez que se invoca a la página JSP. El ciclo de vida transcurre como se describe a continuación:

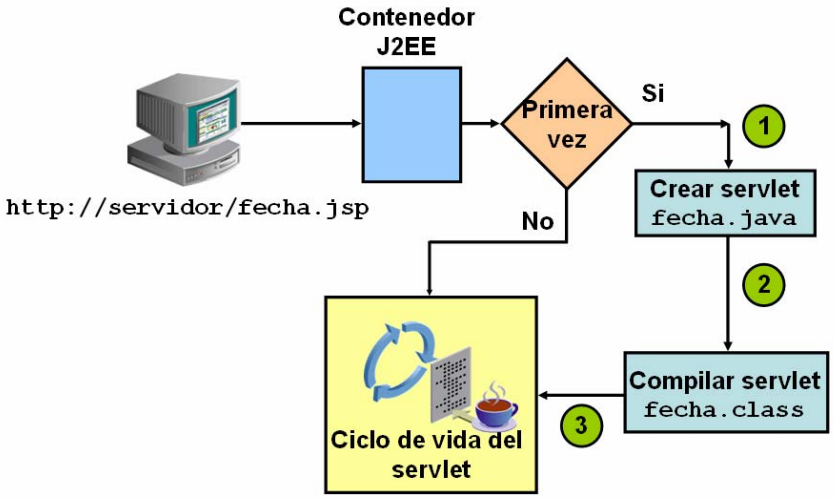
1. La página JSP es transformada en un servlet. Durante esta fase de transformación, cada una de las diferentes etiquetas JSP es manejada de diferente forma, ya que algunas etiquetas proveen instrucciones para el contenedor y otras son utilizadas para crear contenido dinámico.

2. Después de que la transformación es finalizada, el código fuente del servlet es compilado y se genera la clase servlet.

3. El servidor ejecuta el servlet siguiendo el ciclo de vida del servlet.

Si el servlet ya existe, entonces el servlet es ejecutado como se menciona en el paso 3. El ciclo de vida de una página JSP se puede apreciar gráficamente en la figura 9.

Figura 9. Ciclo de vida de una página JSP



Fuente: Oracle JDeveloper 10g: Build Applicacions with ADF. Oracle University. 2005

2.2.2.5. Elementos básicos de una página JSP

Una página JSP puede contener tres elementos principales:

- **Elementos de Texto:** Los elementos de texto representan la parte estática de la página, utilizan HTML para dar formato a la página.
- **Directivas:** Estos elementos proveen instrucciones para la página JSP en tiempo de ejecución. El contendor JSP procesa estas instrucciones cuando está compilando la página.
- **Elementos de *scripting*:** Estos elementos, representan la parte dinámica de la página y contienen código de Java. Son utilizados para realizar cálculos y para generar contenido dinámico que se necesite. Incluyen etiquetas llamadas scriptlets, expresiones y declaraciones.

2.2.2.5.1. Declaraciones

Como su nombre lo indica, las declaraciones JSP son utilizadas para declarar los métodos o las variables que se utilizarán a lo largo de la página.

Las declaraciones comienzan con la secuencia <%! y terminan con la secuencia %>. Son insertadas dentro del cuerpo del servlet durante la transformación.

A continuación se presentan dos ejemplos, en el primero se declara una variable de tipo entero y se inicializa con un valor de tres. En el segundo se declaran dos variables de tipo cadena de caracteres y se inicializan.

- <%! private int i=3; %>
- <%! private String a="Hola", b=" Mundo"; %>

Es obligatorio terminar la declaración con un punto y coma antes de cerrar la declaración utilizando %>. Este tipo de declaración no genera ningún tipo de salida en la página JSP. Las variables declaradas de esta forma son estáticas por defecto.

2.2.2.5.2. Expresiones

Las expresiones son utilizadas para insertar valores directamente hacia la salida de la página JSP. La expresión es evaluada, el resultado es convertido a una cadena de caracteres y la cadena resultante es escrita en la página de salida. Las expresiones son evaluadas en tiempo de ejecución.

Las expresiones comienzan con la secuencia <%=, contienen cualquier expresión que puede ser evaluada e insertada en el servlet de salida y finaliza con la secuencia %>. Las expresiones no deben contener un punto y coma al final.

A continuación se presentan tres ejemplos. El primero de ellos suma uno al valor de la variable "i" y el resultado es impreso en la página resultado. El segundo suma el valor de dos variables tipo cadena de caracteres y los muestra en la página generada y el último crea un nuevo objeto de tipo java.util.Date y utiliza el método toString() para generar una cadena de caracteres que muestre la fecha.

- <%= i +1 %>
- <%= a +b %>
- <%= new java.util.Date %>

2.2.2.5.3. *Scriptlets*

Un scriptlets permite al programador escribir bloques de código de Java dentro de la página JSP. Este código es ejecutado cada vez que la página JSP es invocada. Un scriptlet puede contener cualquier código válido de Java. Cualquier error en el código escrito dentro de un scriptlet lanzará una excepción, ya sea durante el tiempo de traducción o durante la compilación del servlet.

Los scriptlets comienzan con la secuencia <%, contienen un bloque código Java que es ejecutado cada vez que es invocada la página y terminan con la secuencia %>.

A continuación se presenta un ejemplo de un scriptlet que evalúa el valor de la variable i y dependiendo de ciertas condiciones imprime diferentes mensajes.

```
<% if (i<3)
      out.println("i<3");
   if (i==3)
      out.println("i==3");
   else
      out.println("i>3");
%>
```

El código descrito anteriormente, también puede ser escrito de la siguiente forma y obtener el mismo resultado.

```
<% if (i<3) %>
i<3
<% if (i==3) %>
i==3
<% else %>
i > 3
```

Este código, aunque no esta encerrado entre scriptlets es tratado como código HTML y es convertido a sentencias de impresión durante la fase de transformación. De tal forma, no es necesario incluir la sentencia de impresión repetidamente.

2.2.2.5.4. Directivas JSP

Las directivas JSP contienen instrucciones para el contenedor JSP en el que se publicarán las páginas. Son utilizadas para configurar valores globales tales como declaración de clases, implementación de métodos y otros. Todas estas directivas tienen como ámbito la página JSP que las contiene.

La directivas comienzan con la secuencia <%@ y terminan con la secuencia %>. Hay tres tipos de directivas *page*, *include*, *taglib*.

2.2.2.5.4.1. Directiva *page*

La directiva *page* es utilizada para definir atributos importantes para la página JSP. Esta directiva se puede incluir cuantas veces se necesite. Los atributos más relevantes que se pueden utilizar con esta directiva, se listan a continuación[5]:

- ***import*:** Este atributo permite especificar una lista de paquetes o clases, separadas por coma, que deben ser importadas por la clase servlet creada para la página JSP.
- ***contentType*:** Este atributo configura el valor del tipo MIME (*Multipurpose Internet Mail Extensions*) que define el tipo de contenido que será generado por la página JSP, el valor por defecto es text/HTML.
- ***isThreadSafe*:** Si el valor de este atributo es “false”, entonces se creará una instancia del servlet para atender cada una de las solicitudes que se hagan de la página. El valor por defecto es “true”.
- ***session*:** Si el valor de este atributo es falso, la solicitud que realice un cliente no es asociada con ninguna sesión. El valor por defecto es verdadero.

2.2.2.5.4.2. Directiva *include*

Esta directiva es utilizada para incluir archivos dentro de la actual página JSP. Típicamente se utiliza la directiva *include* para incluir archivos que cambian rara vez:

- <%@ include file="/encabezado.jsp" %>

2.2.2.5.4.3. Directiva *taglib*

La directiva *taglib* se utiliza para especificar librerías de etiquetas personalizadas.

2.2.2.6. Objetos implícitos de una página JSP

En una página JSP, existen objetos implícitos provistos para ser utilizados. Se pueden utilizar estas variables predefinidas sin necesidad de explícitamente declararlas. Los objetos implícitos, son creados por el contenedor en el cual esta publicada la página JSP y contienen información relacionada con una solicitud en particular, una página o una sesión.

Existen ochos objetos implícitos, también conocidos como variables predefinidas:

- ***request*:** Esta variable es el objeto HttpServletRequest que esta asociado con la solicitud. Se puede acceder a los parámetros de solicitud, así como al tipo de solicitud, las cabeceras HTTP y otros valores que contiene esta variable.
- ***response*:** Esta variable es el objeto HttpServletResponse que esta asociada con la respuesta del cliente.

- ***session*:** Esta variable es el objeto HttpSession que esta asociado con la solicitud. Las sesiones son creadas automáticamente por el servidor de aplicaciones. Se puede deshabilitar este comportamiento por defecto si no se quiere crear una sesión por cliente.
- ***out*:** Esta variable es el objeto PrintWriter que es utilizado para enviar mensajes de salida al cliente.
- ***application*:** Los servlets y las páginas JSP almacenan datos persistentes en el objeto ServletContext. La variable de aplicación es la representación del objeto ServleContext.
- ***config*:** Esta variable es utilizada para almacenar el objeto ServletContex para la página.
- ***pageContext*:** El objeto pageContext es utilizado para dar un único punto de acceso para los atributos de la página.
- ***page*:** El objeto page es el sinónimo de this.

3. CAPA DEL MODELO

3.1. Definición del modelo

La capa del modelo dentro del patrón de diseño MVC, provee un único punto de acceso para el mantenimiento de la información. Esto significa que la capa del modelo provee la persistencia de los datos, prepara y envía los datos que sean solicitados.

En Oracle ADF la capa del modelo es extendida y dividida en dos subcapas: la capa de los servicios del negocio y la capa del modelo.

3.2. Capa de servicios del negocio

La capa de servicios del negocio maneja toda la interacción con la capa de persistencia de los datos (fuente de datos). Provee servicios tales como persistencia de los datos, asociaciones objeto/relacional, manejo de transacciones y ejecución de la lógica del negoció.

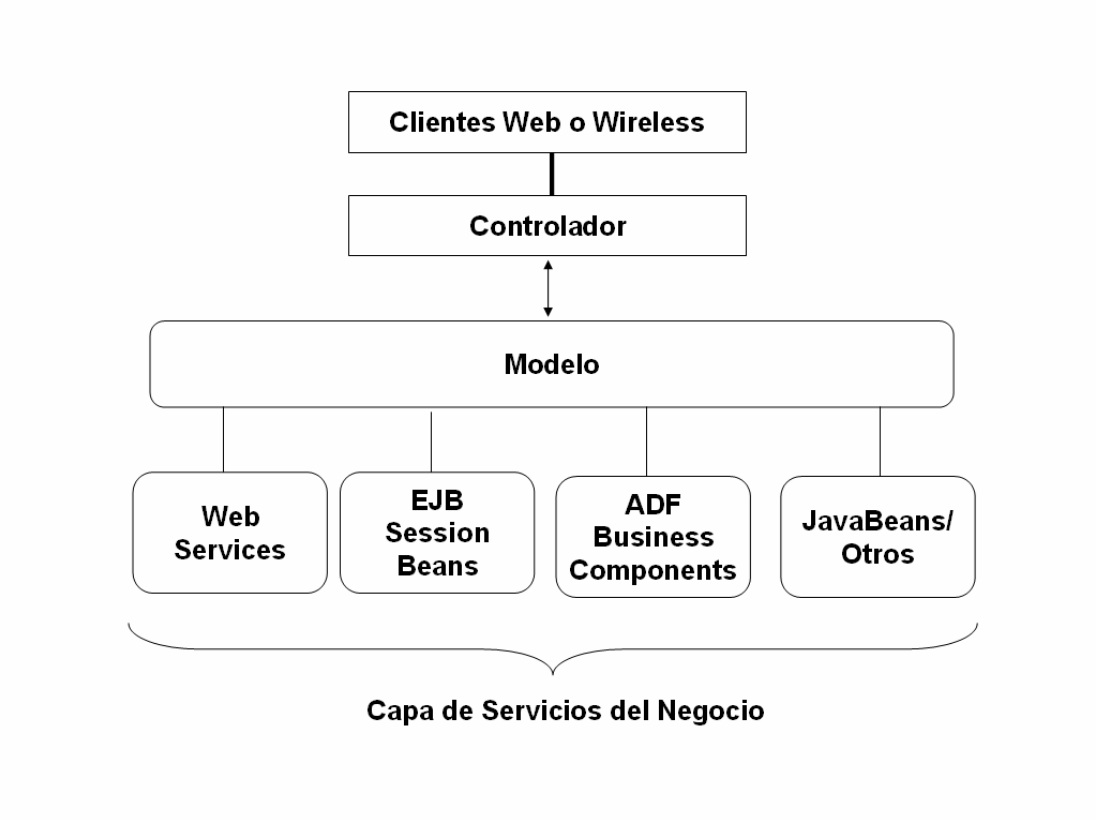
Esta capa es responsable de encapsular los objetos y funciones del negocio. Es en esta capa donde se asocian objetos y tablas relacionales, se ejecutan validaciones, funciones de negocio y se implementa el dominio de las reglas del negocio.

La capa de servicios de negocio puede ser implementada en Oracle ADF como clases simples de Java, EJBs, Web services, objetos TopLink o componentes del negocio Oracle ADF. Dentro del desarrollo de una aplicación ADF se puede elegir cualquiera de estas tecnologías o se puede utilizar una combinación de estas.

3.3. Capa del modelo

La capa del modelo, implementada a través de Oracle ADF, es una capa de abstracción por encima de la capa de servicios del negocio. Esta abstracción provee una única interfaz consistente para la capa de los servicios del negocio, de tal forma que, sin importar la tecnología con la que fue implementada la capa de servicios del negocio, la capa de la vista y la capa del controlador tendrán la misma percepción sobre la implementación del modelo (ver figura 10).

Figura 10. Capa del Modelo



La capa del modelo es la conexión o interfase entre la aplicación cliente y los servicios del negocio. Los servicios del negocio proveerán la persistencia y las reglas del negocio.

La arquitectura del modelo permite un diseño e implementación flexible, debido a que la tecnología por medio de la cual se implementan los servicios del negocio puede cambiar, sin necesidad de afectar las aplicaciones cliente.

3.4. Componentes de la capa del Modelo

3.4.1. *Data Bindings* y *Data Controls*

Como se ha descrito, para desarrollar una aplicación en base al modelo propuesto por Oracle ADF, se pueden utilizar diversas tecnologías e incluso pueden combinarse en la misma aplicación. Esto presenta un reto que añade complejidad al desarrollo, dado que desde la vista y el controlador es necesario consultar e interactuar con los objetos del negocio que se implementan en el modelo, provocando que los detalles de implementación del modelo trasciendan a las otras capas. Lo que significa que si el modelo está implementado con EJB y Web services, la complejidad de esta tecnología se expanda hasta la vista y el modelo.

Para alcanzar el reto antes descrito, la JSR 227[6] (*Java Specification Request* 227) ha estandarizado la "*Data Binding Facility*". Este estándar crea una abstracción sobre la capa de servicios del negocio. La capa del modelo, por ser implementada con JSR 227, es una abstracción sobre los procesos del negocio.

La capa del modelo consta de varias clases de Java que proporciona Oracle y archivos XML donde se describen los objetos y funciones de la capa de servicios de manera que el controlador y las vistas solo tienen acceso a los servicios del negocio utilizando *data bindings*.

Los ADF Data Bindings implementan interfases de los objetos de servicio a través de clases de Java y una descripción de los objetos de negocio en un archivo XML. Los Data Bindings presentan los servicios de negocio a las capas superiores en un formato neutro, independientemente de la tecnología en que han sido desarrollados. Mediante esta arquitectura se logra que los detalles de implementación de los servicios de negocio no se transmitan a las capas superiores.

Los Data Bindings son meta datos, que describen como los componentes de la interfaz del usuario utilizarán los valores y acciones provistas por la capa de servicios del negocio. Los Data Controls son meta datos que describen el modelo de datos retornado por la capa del servicios del negocio (formas de ingreso, listas, gráficas, etc.).

3.5. Componentes de la capa de servicios del negocio

Existen diversos tipos de componentes a través de los cuales se puede implementar la capa de servicios en Oracle ADF. Algunos se basan en tecnología estándar como EJBs, Web services y JavaBeans. Por el contrario, existen componentes desarrollados por Oracle, que generan tecnología estándar en tiempo de ejecución conocidos como Componentes de Negocio ADF o *ADF Business Components*.

3.5.1. ADF *Business Components*

Los *ADF Business Components* (ADF BC) están diseñados para proveer servicios del negocio para aplicaciones J2EE construidas con Oracle ADF. Estos proveen la persistencia de los datos y la lógica del negocio para

la capa del modelo. Además, proveen interacción entre los clientes y la fuente de datos en una aplicación J2EE y constituyen la infraestructura de la lógica de la aplicación.

Los ADF BC son una capa de negocios integral dentro del marco de trabajo para el desarrollo de aplicaciones. Sus componentes manejan la persistencia de datos, las reglas del negocio, las vistas del negocio y la seguridad. Los BC implementan soluciones basadas en los patrones de diseño y mejores prácticas de J2EE.

Los ADF BC están diseñados para ser escalables y para operar de forma óptima, e interactuar de manera eficiente con la fuente de datos.

3.5.1.1. Beneficios de los ADF *Business Components*

Los ADF BC incrementan la productividad de los desarrolladores al implementar componentes reutilizables. La mayoría de aplicaciones gira alrededor de acceso a los datos e implementaciones de reglas del negocio. Los ADF BC incluyen funciones sobre los datos "listas-para-usar" que administran todo el acceso hacia los datos. No existe necesidad de codificar manualmente rutinas estándar de acceso a la información.

Debido a que los requerimientos para cada negocio son diferentes, los ADF BC permiten aumentar o incluso cambiar el comportamiento estándar, para cumplir con estas actividades.

3.5.1.2. Componentes de dominio del negocio

Los componentes del dominio del negocio son la base fundamental de los ADF BC. Proveen funciones para acceder a los datos, manejar la persistencia, ejecutar reglas de negocio y validación de datos. También proporciona funciones para acceder a objetos relacionados a través de asociaciones. Por ejemplo, desde un objeto que representa un cliente, se puede acceder a los objetos que representen las órdenes que dicho cliente ha realizado y viceversa. Estos componentes también manejan el cache de los datos en la memoria del servidor de aplicaciones donde se encuentren publicados.

3.5.1.2.1. *Entity Objects*

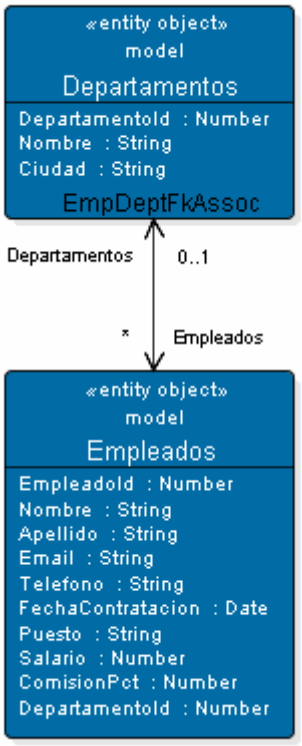
Los *entity objects* son los componentes medulares de los ADF BC. Los *entity objects* contienen los atributos, reglas del negocio e información de persistencia que aplica para una parte específica del modelo de negocios. Los *entity objects* son utilizados para definir la estructura lógica del negocio, tal como productos, departamentos, empleados y clientes entre otros.

Un entity object esta basado sobre una fuente de datos; por ejemplo, si existe una tabla a nivel de base de datos que represente la entidad empleados, podrá existir un entity object que represente a dicha tabla.

Los atributos de los entity objects representan cada una de las columnas de la fuente de datos. Asumiendo que la tabla empleados tiene como columnas el identificador del empleado, el nombre del empleado y la fecha de contratación, el entity object que represente dicha tabla tendría un atributo para representar cada una de estas características.

Se pueden agregar reglas de validación a nivel de los entity objects, por ejemplo, validar que los ingresos de un empleado no sean menores que el salario mínimo.

Figura 11. Diagrama UML de *Entity Objects*



La figura 11 muestra un diagrama UML de los Entity Objects que representan las tablas de departamentos y empleados descritas a continuación:

```
CREATE TABLE departamentos
   ( departamento_id    NUMBER(4)
   , nombre  VARCHAR2(30)
         CONSTRAINT  dept_name_nn  NOT NULL
   , ciudad      VARCHAR2(30)
   ) ;
```

```
CREATE UNIQUE INDEX dept_id_pk
ON departamentos (departamento_id) ;

ALTER TABLE departamentos
ADD ( CONSTRAINT dept_id_pk
                    PRIMARY KEY (departamento_id)
    ) ;

CREATE TABLE empleados
   ( empleado_id    NUMBER(6)
   , nombre     VARCHAR2(20)
   , apellido      VARCHAR2(25)
          CONSTRAINT     emp_last_name_nn  NOT NULL
   , email          VARCHAR2(25)
         CONSTRAINT     emp_email_nn  NOT NULL
   , telefono   VARCHAR2(20)
   , fecha_contratacion      DATE
         CONSTRAINT     emp_hire_date_nn  NOT NULL
   , puesto         VARCHAR2(10)
         CONSTRAINT     emp_job_nn  NOT NULL
   , salario         NUMBER(8,2)
   , comision_pct NUMBER(2,2)
   , departamento_id  NUMBER(4)
   , CONSTRAINT     emp_salary_min
                CHECK (salario > 0)
   , CONSTRAINT     emp_email_uk
                UNIQUE (email)
   ) ;

CREATE UNIQUE INDEX emp_emp_id_pk
ON empleados (empleado_id) ;

ALTER TABLE empleados
ADD ( CONSTRAINT     emp_emp_id_pk
                PRIMARY KEY (empleado_id)
   , CONSTRAINT     emp_dept_fk
```

```
                FOREIGN KEY (departamento_id)
                 REFERENCES departamentos
) ;
```

3.5.1.2.1.1. Estructura de un *Entity Object*

Un Entity Object esta estructurado por defecto por dos archivos. Un archivo con extensión XML y una archivo Java.

El archivo XML contiene información acerca del entity object que le servirá a otros componentes y al mismo contenedor J2EE, en donde se publique la aplicación, para determinar como se puede interactuar con este componente, que objeto relacional esta representando y con que otros entity objects esta relacionado. Este archivo XML debe tener el mismo nombre que el Entity Object con extensión xml. En el caso del Entity Object Departamentos del diagrama UML de la figura 12, el archivo XML asociado puede verse en el apéndice 1.

El archivo Java, es la clase Entity Object, la cual hereda su estructura de la clase *EntityImpl*. Esta clase representa un registro dentro de la tabla representada por el Entity Object. Contiene métodos *set* y *get* para cada uno de lo atributos del Entity Object. La clase de Java debe tener el nombre del Entity Object concatenado con el sufijo Impl, así, para el Entity Object Departamentos del diagrama UML de la figura 12, el archivo XML tiene como nombre DepartamentosImpl.java y el código puede apreciarse en el apéndice 2.

3.5.1.3. Componentes del Modelo de Datos del Negocio

Los componentes del modelo de datos del negocio o *Data Model Business Components* son la fachada que el cliente puede manipular. Proveen acceso a los datos para las aplicaciones cliente y restringen la información que estos pueden manipular.

El componente principal de estos componentes de fachada es el Application Module. El Application Module contiene y publica otros componentes de fachada. Es el contenedor global de todos los otros componentes del Modelo de Datos del Negocio.

Los *View Objects* y los *View Link* son components que perteneces a Oracle ADF. Los *View Objects* son colecciones de datos que proveen un punto de acceso a la información para las aplicaciones cliente. Los *View Link* son relaciones entre View Objects y proveen la funcionalidad necesaria para la coordinación de las relaciones maestro-detalle que existan.

3.5.1.3.1. *View Objects*

Los *View Object* son parte de la capa de los ADF BC, y actúan como una interfaz entre el módulo de la aplicación (*Application Module*) y la base de datos.

Un *View Object* es una clase que permite definir y trabajar con conjuntos de registros que a menudo forman parte de la interfaz del usuario. Regularmente, un view object contiene una consulta de SQL que selecciona datos de la fuente de información. Puede estar asociada con un Entity Object, de tal forma que se pueden modificar las entidades persistentes. Se deben crear los view objects según las necesidades del cliente.

La primera parte de construir una aplicación utilizando ADF BC consiste en diseñar y construir la lógica de la aplicación. Luego que se diseña y se prueba la aplicación, es necesario concentrarse en como presentar los datos de una manera útil y lógica. Existen un sin número de formas en las cuales se desee visualizar la información en la aplicación. Cada diferente forma, sin importar si es un subconjunto de atributos o un conjunto de registros, puede ser expuesta como un *view object*.

Un View Object puede estar relacionado con un entity object, por ejemplo, puede existir un View Object que representa la información del entity object empleados.

Un atributo del view object se sincroniza con los atributos del entity object en el cuál esta basado. El view object utiliza una consulta de SQL para ordenar y filtrar la información.

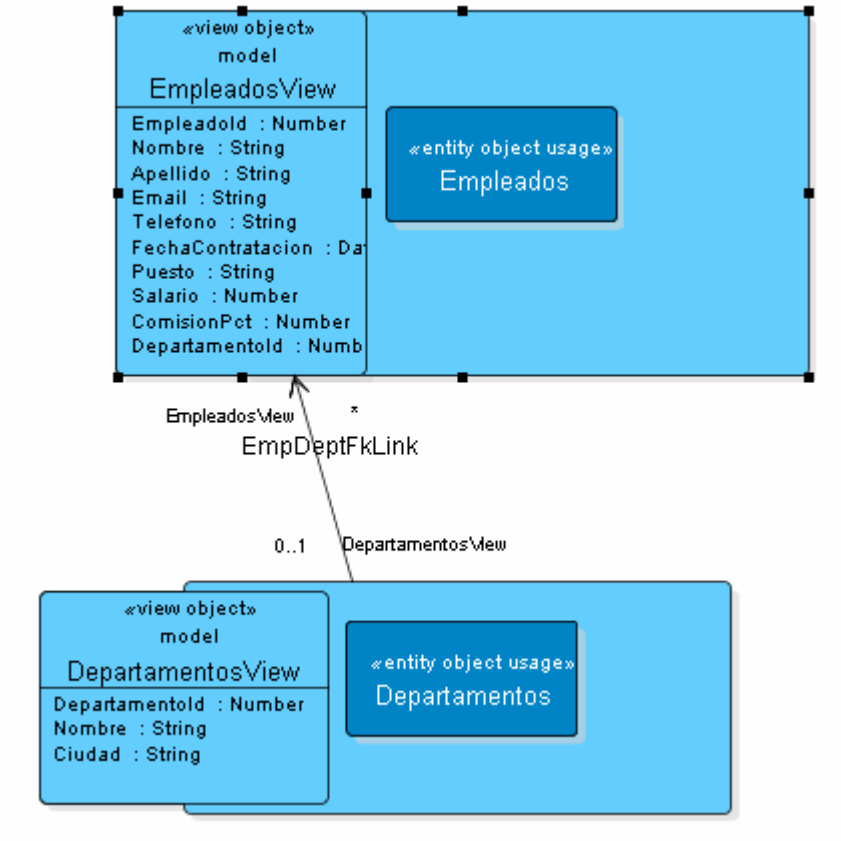
En la figura 12, se muestran dos view objects basados en los entity objects mostrados en la figura 11.

El view object EmpleadosView está basado en la siguiente consulta:

```
SELECT Empleados.EMPLEADO_ID,
    Empleados.NOMBRE,
    Empleados.APELLIDO,
    Empleados.EMAIL,
    Empleados.TELEFONO,
    Empleados.FECHA_CONTRATACION,
    Empleados.PUESTO,
    Empleados.SALARIO,
    Empleados.COMISION_PCT,
    Empleados.DEPARTAMENTO_ID
```

FROM EMPLEADOS Empleados

Figura 12. Diagrama UML *View Objects*



El view object DepartamentosView está basado en la siguiente consulta:

```
SELECT Departamentos.DEPARTAMENTO_ID,
    Departamentos.NOMBRE,
    Departamentos.CIUDAD
FROM DEPARTAMENTOS Departamentos
```

3.5.1.3.1.1. Interacción entre *View* y *Entity Objects*

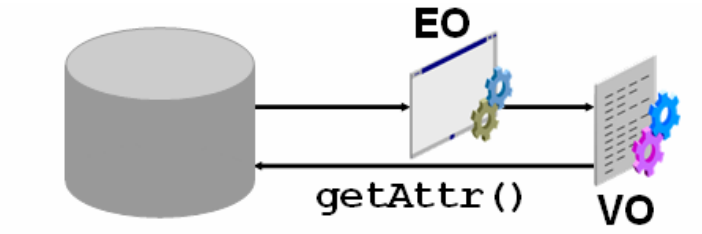
3.5.1.3.1.1.1. Consultando información

El view object obtiene sus datos utilizando la consulta de SQL en la cual esta basada, y está atado a un entity object. Los entity objects y los view objects trabajan en conjunto cuando un view object consultado datos:

- El view object consulta la base de datos directamente, asegurándose de los datos sean consistentes (no existen bloqueos en los registros o no han sido actualizados por otra transacción).
- Los datos obtenidos por la consulta son almacenados en el cache de los entity objecs y luego son enviado al view object.

La figura 13 muestra lo descrito anteriormente

Figura 13. Interacción entre VO y EO para consultar datos



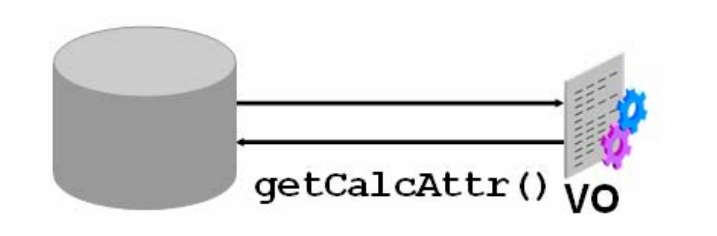
Fuente: Oracle JDeveloper 10g: Build Applicacions with ADF. Oracle University. 2005

3.5.1.3.1.1.1.1. Atributos calculados

Un atributo calculado se base únicamente en la consulta hacia la base de daos, no en un atributo de un entity object. Los atributos calculados

obtienen sus datos directamente de las columnas en la consulta de SQL. Los datos son almacenados en el cache del view object (figura 14).

Figura 14. Obtención de un atributo calculado por un VO



Fuente: Oracle JDeveloper 10g: Build Applicacions with ADF. Oracle University. 2005

3.5.1.3.1.1.2. *Entity Objects*

Los datos son enviados a los entity objects para validar las reglas del negocio almacenadas en el objeto. Los view objects determinan que información debe ser recuperada. El entity object garantiza que los datos sigan las reglas definidas del negocio de forma que, no se pueden consultar datos que no satisfagan la lógica del negocio almacenada en un entity object.

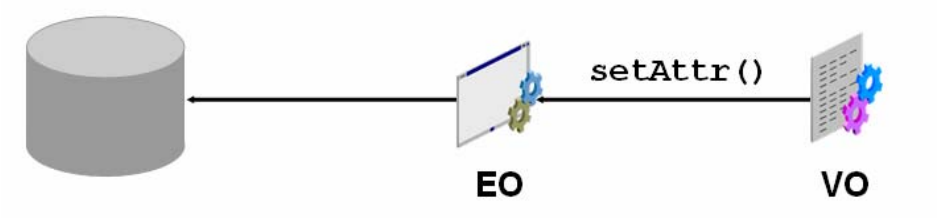
3.5.1.3.1.1.2. Actualizando datos

Cuando se actualiza la información en la base de datos, se ejecuta la siguiente secuencia de eventos:

- El view object actualiza el cache del entity object.
- Cuando una transacción ejecuta un *commit*, el entity object actualiza la base de datos.

La figura 15, muestra de forma gráfica lo descrito anteriormente

Figura 15. Interacción entre un VO y un EO para actualizar datos



Fuente: Oracle JDeveloper 10g: Build Applicacions with ADF. Oracle University. 2005

3.5.1.3.1.2. Estructura de un *View Object*

El view object está estructurado por un archivo XML que contiene meta datos acerca del view object y dos clases java que permiten manipular la información que este representa. Estas clases pueden adecuarse a la funcionalidad que requiera la aplicación en construcción.

La estructura de los archivos que componen un view object es:

- El archivo XML lleva el nombre del view object y contiene información que le servirá al contenedor y a otros componentes para determinar como se debe interactuar con él, que datos representa (contiene la consulta SQL en la cual se basa) y que entity objets interactuan con él. Para el caso del diagrama del view object EmpleadosView de la figura 12, el archivo se llama EmpleadosView.xml y puede ser visto en el apéndice 3.

- La primera clase de Java representa al view object y que permite modificar el comportamiento por defecto del mismo, este clase lleva el nombre del view object más el sufijo Impl, de forma que para el view object EmpleadosView, el archivo tendría el nombre de EmpleadosViewImpl.java y su estructura puede ser vista en el apéndice 4.

- La segunda clase de java representa los registros que se obtienen a través de un view object. Contiene métodos set y get para cada atributo del view object. Tiene el nombre del view object más el sufijo RowImpl. Para el view object EmpleadosView está clase llevaría el nombre de EmpleadosViewRowImpl.java y contendría los métodos getNombre(), getApellido(), setNombre(), etc. La estructura de este archivo puede ser vista en el apéndice 5.

3.5.1.3.2. *Application Module*

Un *application module* es un componente que forma parte de la capa de servicios del negocio de Oracle ADF. Es una clase que representa una tarea específica de la aplicación, además del código que implementa dicha tarea. El application module provee servicios que brindan soporte al cliente para cumplir con sus requerimientos. Un application module puede representar y asistir en tareas tal como:

- Actualizar información de un cliente
- Crear una nueva orden
- Procesar el incremento de salarios
- Invocar servicios Web para soportar integración B2B (bussiness-to-business)

Representa el modelo de datos que un cliente puede utilizar. Para crear el modelo de datos, el application module contiene instancias nombradas de los view objects y view links. Los view objects son componentes que permiten recuperan información y los view link permiten manipular las relaciones que existen entre las instancias de los view objects.

En conjunto, los view objets y los view link representan las vistas de los datos que son requeridas para una tarea específica. Por ejemplo, para cumplir con la tarea de crear una nueva orden se requieren:

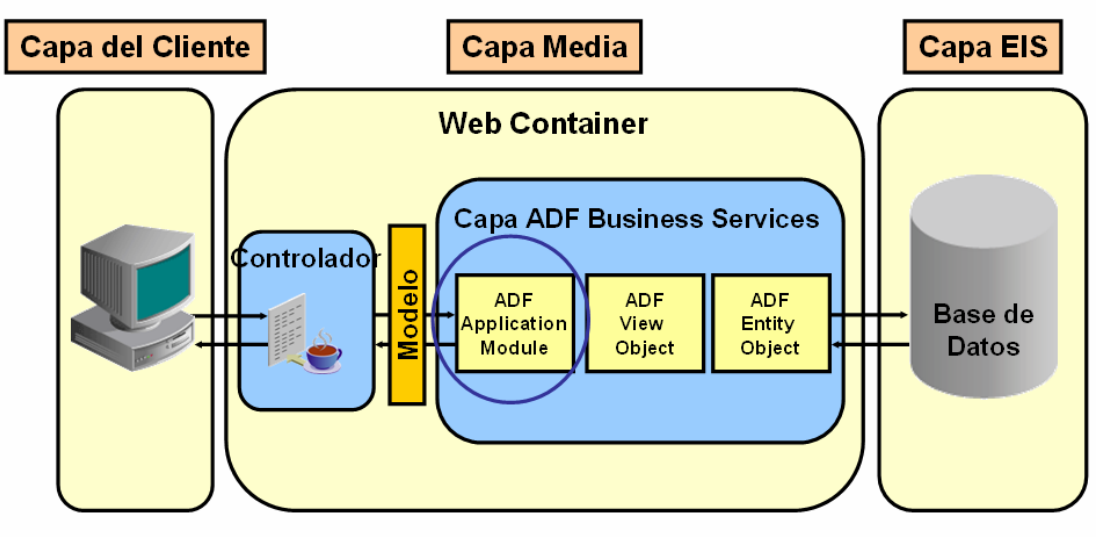
- Una vista de los detalles mínimos de un cliente (nombre, teléfono, código)
- Una vista de los productos disponibles que pueden ser ordenados (para crear una página de navegación)
- Una vista para la información del encabezado de la orden (en la cual se crearán las nuevas ordenes)
- Un view link que permita obtener los productos incluidos en una orden (en el cual se ingresarán los productos de la orden)

Diferentes formas o páginas en la aplicación cliente pueden compartir la instancia del view object y referirse a ella por su nombre debido a que se publican a través del application module.

La capa de la lógica del negocio esta compuesta por uno o más application modules con los cuales las otras capas puede interactuar (capa del cliente y capa de datos).

La figura 16 muestra la arquitectura de una aplicación construida a partir de componentes ADF BC y como estos componentes interactúan con las otras capas.

Figura 16. Arquitectura ADF BC



Fuente: Oracle JDeveloper 10g: Build Applicacions with ADF. Oracle University. 2005

Una application module es parte de la capa de la lógica del negocio, el cual es accedido por las otras capas. Las aplicaciones clientes y la base de datos interactúan con los application modules. El propósito y funciones principales de un application module se presentan a continuación:

- Representar el modelo de datos que el cliente utiliza y a la vez contiene componentes del negocio incluyendo instancias de los view objects y los view links.
- Monitorear todos los cambios que afecten a los datos en la fuente de datos y administrar las transacciones.
- Proveer conectividad hacia el sistema de información empresarial.
- Brindar métodos de acceso remoto a la funcionalidad implementada en el módulo.
- Proporcionar métodos personalizados que el desarrollador ha escrito para procesar requerimientos específicos de la aplicación.

Un application module provee un ambiente transaccional para las instancias de los view objects que contiene. La transacción mantiene control sobre todos los cambios que afectan a los datos en la fuente de datos. Todos los cambios realizados a los datos durante una transacción son registrados de forma permanente (commit) o restablecidos (rollback) juntos.

El application module consta de dos archivos, uno XML, que contiene meta datos que le indican al contenedor como debe interactuar con este componte y una clase de Java que contiene la implementación de este componente y es donde se pueden agregar métodos personalizados. En el apéndice 6 y 7 se presenta ejemplos de estos archivos.

El código necesario para realizar una tarea específica puede ser escrito dentro de la clase del application module, de forma que se encapsula la lógica de procesamiento y se simplifica el trabajo de utilizar esta funcionalidad en la capa del controlador y de presentación. Por ejemplo, un modulo que permite crear ordenes nuevas podría tener un método nombrado "crearNuevaOrdenParaCliente()" el cual encapsularía los detalle de buscar al cliente y asignarlo a la orden.

Cada componente de la capa de servicios de negocio puede tener código asociado que él para realizar un rol particular. Esta arquitectura permite que los componentes sean reutilizables. En general, se puede determinar donde se debe agregar cierta funcionalidad basándose en las siguientes guías:

- Un entity object contiene lógica del negocio que pertenece a una sola entidad del negocio. Todo los view object basados en un entity object comparte esta lógica. Al nivel de los entity, los cálculos se hacen en código de Java.

- Un view object contiene lógica que pertenece a una consulta de SQL que hace hacia la base de datos. Esta puede incluir expresiones SQL calculadas, reuniones, uniones, subconsultas, etc. Se puede agregar código de Java, por ejemplo, para propagar eventos generados en la capa de presentación, o crear un método que invoque a un método de un entity object que se desee exponer para esta vista.
- Los archivos fuente de una application module contienen lógica específica para la tarea que realiza; lógica que no sería apropiada colocar en un entity object o un view object que varios application modules que realicen diferentes tareas pueden utilizar. Si dos application modules utilizan la misma vista, se debe colocar la lógica específica para una de las aplicaciones en sus archivos fuente en lugar de colocarla en el entity object o en el view object.

Un application module provee métodos que implementan el comportamiento del mismo y que son accesibles para los clientes. Existen métodos predefinidos por el framework que el cliente puede utilizar para trabajar con la instancia del aplicación module. Por ejemplo, una application module puede permite obtener instancias de los view objects y ejecutar las consultas, manipular transacciones, y otras tareas.

Además, se pueden agregar métodos personalizados y selectivamente publicar estos métodos para que sean accedidos por los clientes. Por ejemplo, un proceso que incremente el salario de un empleado puede ser publicado de tal forma que desde la capa del controlador pueda ser invocado para encontrar al empleado y realizar el aumento.

Se puede publicar el mismo application module utilizando diferentes configuraciones, tal es el caso de publicación EJB, WAR o módulo web, etc., sin cambiar su código. Además, el mismo application module puede ser

utilizado para construir una aplicación de una, dos o tres capas sin ningún cambio.

En tiempo de ejecución, los clientes crean una instancia de una application module para utilizarla. Estas instancias no son compartidas entre clientes.

4. CAPA DEL CONTROLADOR

4.1. Definición del controlador

El controlador es responsable de administrar el flujo de la aplicación. La utilización de un controlador facilita el desarrollo e incrementa la productividad; alcanzando un desarrollo de aplicaciones más flexible y fácil de mantener. Desde el controlador se realiza la interacción programática con el modelo, para acceder a las funciones y datos de negocio.

El controlador es el encargado de redireccionar las solicitudes de los clientes a los componentes adecuados de la capa de la vista. Las decisiones de navegación entre páginas, a partir de una solicitud de un cliente, pueden estar basadas en función de una acción del mismo (presionando un botón, un link, etc.), o en una combinación de eventos sobre la información.

Un beneficio de utilizar un controlador en una aplicación Web es, que cada página individual no tiene necesidad de incluir lógica de navegación. La navegación de las páginas es manejada por el controlador, de forma que, si la navegación cambia, las páginas no son modificadas. Adicionalmente, las páginas no necesitan conocer de donde se originan las solicitudes que la han invocado. Las paginas únicamente debe responder a las solicitudes y el controlador manejará el resto. En esencia, separa la capa del modelo de la capa de la vista.

Para el desarrollo del controlador, ADF proporciona la opción de utilizar *Struts*.

4.2. Introducción a Struts

Oracle ADF utiliza tecnología del proyecto Apache Struts[7] para implementar el controlador. Struts es un proyecto de la fundación Apache (Apache Software Foundation) y el objetivo principal de dicho proyecto es proveer un *framework* para construir aplicaciones Web J2EE utilizando el patrón de diseño MVC.

El núcleo del framework de Struts es una capa flexible de control basada en tecnología estándar[8], como los Java Servlets, JavaBeans, ResourceBundles y XML, así como un conjunto de paquetes (Jakarta Commons[9]). Struts provee de sus propios componentes controladores e integra otras tecnologías para implementar el modelo y la vista. Para el modelo, Struts puede interactuar con tecnologías estándares para acceso a datos, como JDBC y EJB, así como la mayoría de implementaciones de terceros, tal como ADF BC. Para la vista, Struts trabaja de manera eficiente con páginas JavaServer Pages, incluyendo JSTL y JSF, así como otros sistemas de presentación (XSTL, etc.).

Los componentes que ofrece Struts nos permiten implementar la capa del controlador en una aplicación construida con Oracle ADF. Los componentes del controlador serán los encargados de coordinar las actividades de la aplicación, que van desde la recepción de datos del usuario, las verificaciones de forma hasta la selección de un componente del modelo a ser invocado. Por su parte, los componentes del modelo envían al controlador sus resultados y/o errores a manera de poder continuar con otros procesos de la aplicación.

Esta separación simplifica enormemente la construcción del modelo y garantiza la reutilización de cada capa de forma independiente.

Entre las características más importantes de Struts están:

- La configuración del controlador es centralizada.
- Las interrelaciones entre acciones y páginas u otras acciones se especifican en documentos XML en lugar de codificarlas en los programas o páginas.
- Los componentes permiten compartir información bidireccionalmente entre el usuario de la aplicación y las acciones del modelo.
- Contiene un conjunto de librerías que facilitan la construcción de páginas JSP con operaciones comunes que éstas realizan.
- Struts contiene herramientas para validación de campos de plantillas. Provee varios esquemas, que van desde validaciones locales en la página (en JavaScript) hasta las validaciones a nivel de las acciones.

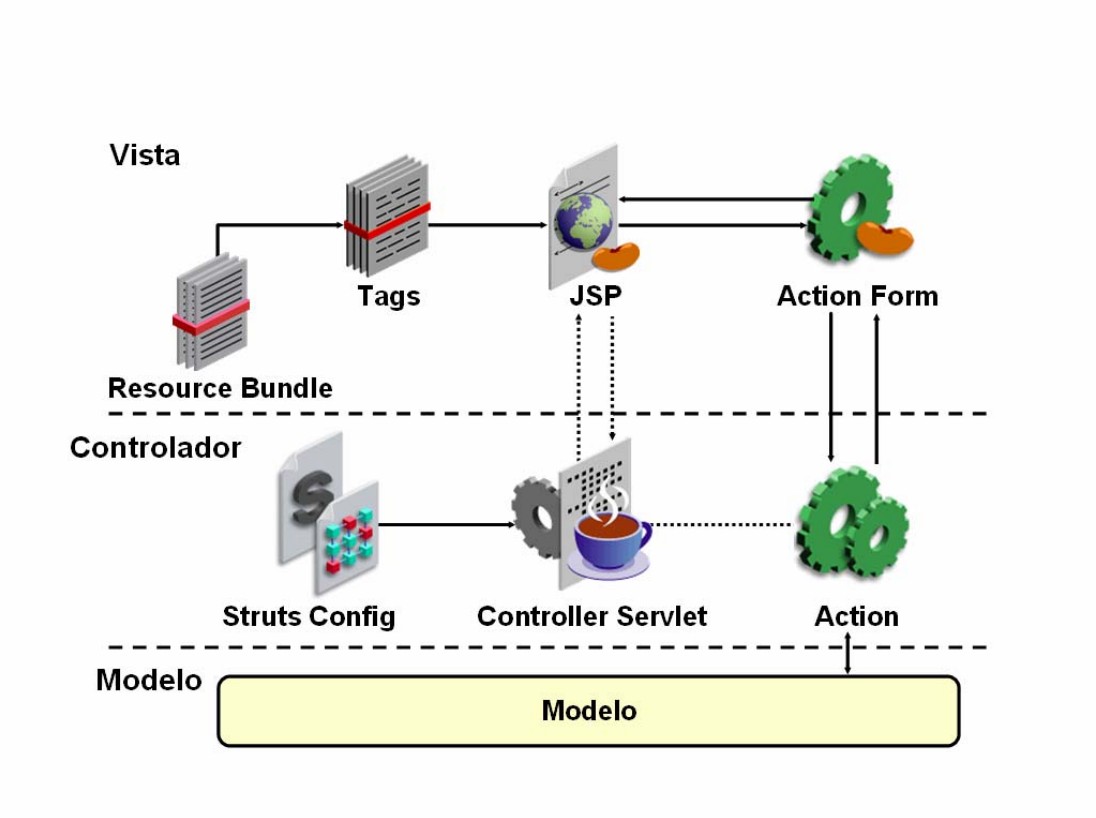
Struts permite que el desarrollador se concentre en el diseño de aplicaciones complejas como una serie simple de componentes del modelo y de la vista intercomunicados por un controlador centralizado. El resultado es una aplicación más consistente y más fácil de mantener.

4.3. Componentes de la capa del controlador en Oracle ADF

La figura 17 presenta la relación existente entre los componentes de una aplicación construida con el framework de Struts. El Controller Servlet o Servlet Controlador se enfoca en el flujo de la aplicación y las páginas JSP se enfocan en la presentación. La figura 17 muestra que el Controlador Servlet consulta el archivo de configuración de Struts para identificar las rutas y eventos que regirán el flujo de la aplicación. Utilizando componentes Action,

se comunica hacia el modelo para devolver una respuesta a la página JSP de donde provino la solicitud.

Figura 17. Componentes Struts



Fuente: Oracle JDeveloper 10g: Build Applicacions with ADF. Oracle University. 2005

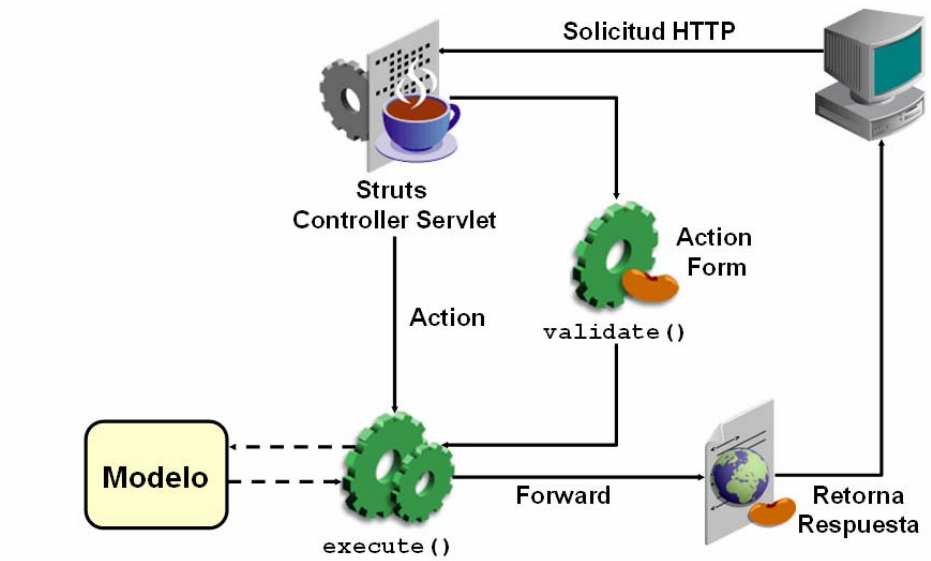
La siguiente lista presenta una descripción resumida de cada uno de los componentes Struts que pueden ser incluidos dentro de una aplicación Oracle ADF y el rol que desempeñan dentro de la aplicación:

- Controlador Servlet Struts: Maneja las solicitudes y devuelve resultados.
- Archivo de configuración Struts: Es mapa que utiliza el Servlet para regir el flujo de la aplicación.
- Acciones: Procesan una solicitud y dan como resultado un evento (action forward).

- ActionForms: Permiten transmitir información entre el cliente y el modelo.
- TagLibs: Conjunto de librerías de etiquetas Struts que se puede utilizar en la páginas JSPs.
- Resource Bundle: Archivo de texto para internacionalización de mensajes. Permite la construcción de aplicaciones multilenguaje.

4.3.1. Flujo de los componentes Struts

Figura 18. Flujo de los componentes Struts



Fuente: Oracle JDeveloper 10g: Build Applicacions with ADF. Oracle University. 2005

En la figura 18, muestra los principios generales del flujo de procesos en Struts:

1. La clase ActionServlet (Servlet Controlador Struts) recibe y enrruta el tráfico http al manejador apropiado dentro del framework.
2. Si una ActionForm está asociada a la acción, entonces el método validate() del ActionForm se ejecuta y realiza validaciones. En caso de que no exista un ActionForm asociado a la acción, esta continúa su ejecución normal.
3. Si la validación falla en el ActionForm, se retorna un error, de lo contrario la acción continúa su ejecución normal.
4. El método execute() de la acción envía una clase ActionForward al controlador. Este método es el encargado de invocar procesos de la lógica del negocio que residen en el modelo.
5. La clase ActionForward envía una respuesta de vuelta al cliente solicitante.

4.3.2. *Servlet* controlador

Es un servlet que controla todo el flujo de procesos de la aplicación. Este servlet se implementa a través de la clase ActionServlet, la cual es parte del paquete org.apache.struts.action y hereda su estructura de la clase HttpServlet.

En un aplicación construida con Oracle ADF, este clase no puede ser modificada, pero si debe existir una referencia a dicha clase en el archivo web.xml (archivo de configuración de la capa de presentación).

El servlet controlador es configurado a través de un archivo XML en el cual se indican las rutas del flujo de la aplicación y los eventos que puede ocurrir dentro del contexto de la misma. Este archivo XML es el archivo struts-config.xml.

4.3.3. El archivo de configuración de Struts

Identificado como struts-config.xml, es utilizado para guiar al servlet controlador. Este archivo provee información que define para la aplicación el mapa de las acciones, el flujo de las páginas, el flujo de los datos, manejo de excepciones y los recursos utilizados por la misma.

El archivo struts-config.xml es un mapa del flujo que debe seguir la aplicación en función de una serie de eventos que se van generando. Es en éste archivo en el cual se define el flujo lógico de la aplicación utilizando sintaxis XML. Además, describe las clases Form Beans, Actions, ActionMappings, resource bundles, conexiones data source y otros parámetros.

A continuación se presenta el ejemplo de un archivo struts-config.xml

```
<struts-config>
 <action-mappings>
  <action path="/action1"/>
   <forward name="success" path="/page1.do"/>
  </action>
 </action-mappings>
 <message-resources parameter="view.ApplicationResources"/>
</struts-config>
```

El ejemplo anterior, muestra el flujo de una aplicación simple, en el cual se indica que, si el proceso se encuentra en la clase action1 y sucede el evento “success” el flujo de la aplicación debe dirigirse hace la clase page1.do. Este flujo es conocido como un ActionMapping.

4.3.4. Clases *Action*

La clase org.apache.struts.action.Action actúa como un puente entre la solicitud del cliente y una operación del negocio. La clase Action procesa una solicitud HTTP y retorna un objeto ActionForward como respuesta. Una acción existe por cada solicitud lógica.

El controlador selecciona la acción apropiada para cada solicitud basado en los mapas de acciones (ActionMappings) definidos en el archivo struts-config.xml y luego invoca al método execute() del Action correspondiente.

La clase Action puede implementar lógica en su respectivo método execute(). Sin embargo, la lógica y el acceso a la base de datos debe ser realizado solicitando un método apropiado del Application Module.

El tipo de retorno de un objeto ActionForward le indica al controlador hacía donde debe fluir la aplicación y cuyo destino puede ser, tanto una página JSP como otra acción.

La definición XML de una acción en el archivo struts-config.xml es:

```
<action path="/action1"/ type="view.AuthUserAction"/>
```

Donde el atributo *path* es un nombre lógico con el cuál se puede invocar la acción, mientras que *type* indica en nombre físico de la acción en donde se implementa la lógica.

4.3.4.1. Ejemplo de una Clase *Action*

La clase Action tiene un único método que obligatoriamente debe ser implementado, el método execute(). Cuando se invoca la acción, éste es el método que se ejecuta y que recibe como parámetros un objeto de tipo ActionMapping, un objeto ActionForm, un objeto HttpServletRequest y un objeto HttpServletResponse.

El método execute() retorna un objeto ActionForward, también denominado *page forward*.

A continuación se presenta el código por defecto que tiene una clase Action generada por Oracle ADF:

```
public class AuthUserAction extends Action
{
/* Esta es la acción principal que se llama desde el framework Struts.*/
public ActionForward execute(
                ActionMapping mapping,
                ActionForm form,
                HttpServletRequest request,
                HttpServletResponse response)
        throws IOException, ServletException
  {
    return mapping.findForward("success");
  }
}
```

Generalmente, cuando se escribe una acción Struts, se hereda la estructura de la clase Action y se implementa un método execute()

personalizado a las necesidades de la aplicación. El *framework* transfiere todo lo necesario al método incluyendo:

- Una referencia al metadata de Struts (ActionMappings), de forma que se pueda resolver el nombre de los page forwards.
- Un form bean (estático o dinámico) que contiene información de la página solicitante.
- Una referencia al objeto *request* y a la sesión.
- Una referencia al objeto *response*, por medio del cual se puede registrar información para ser utilizada posteriormente, como datos ó errores y a la cual se tendrá acceso en la respuesta.

4.3.5. *Page Forwards*

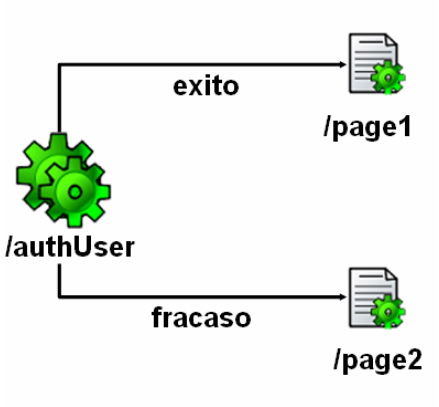
Los page forwards controlan el flujo de la aplicación. Uno o más forwards son definidos entre el elemento <action> en el archivo struts-config.xml.

A continuación se presenta una porción del archivo struts-config.xml donde se definen page forwards para un Action:

```
<action path="/authUser" type="view.AuthUserAction">
  <forward path="/page1.do" name="exito"/>
  <forward path="/page2.do" name="fracaso"/>
</action>
```

La representación gráfica puede apreciarse en la figura 19.

Figura 19. Diagrama de *Pages Forwards*



Si la clase authUser responde con un ActionForward con valor de "éxito", el flujo de la aplicación será redireccionado a la página page1. Si la clase authUser genera un ActionForward con valor de "fracaso", el flujo de la aplicación será redireccionado a la página page2.

4.3.6. *ActionForward* de la clase *Action*

Para ejecutar el redireccionamiento de uno a otro componente, el diseñador de la aplicación debe implementar el método execute() y en este generar un page forward. En Oracle ADF, esta tarea se consigue sobrecargando el método findForward(), que acepta como parámetro el nombre de page forward que se debe retornar. El parámetro de retorno de un ActionForward indica al controlador hacia donde debe enviar el flujo de la aplicación. El nombre por defecto de un page forward simple es *success*.

El método execute() puede ser personalizado, de forma que pueden generar diferentes page forwards como resultado, por ejemplo:

```
public ActionForward execute(
        ActionMapping mapping,
        ActionForm form,
```

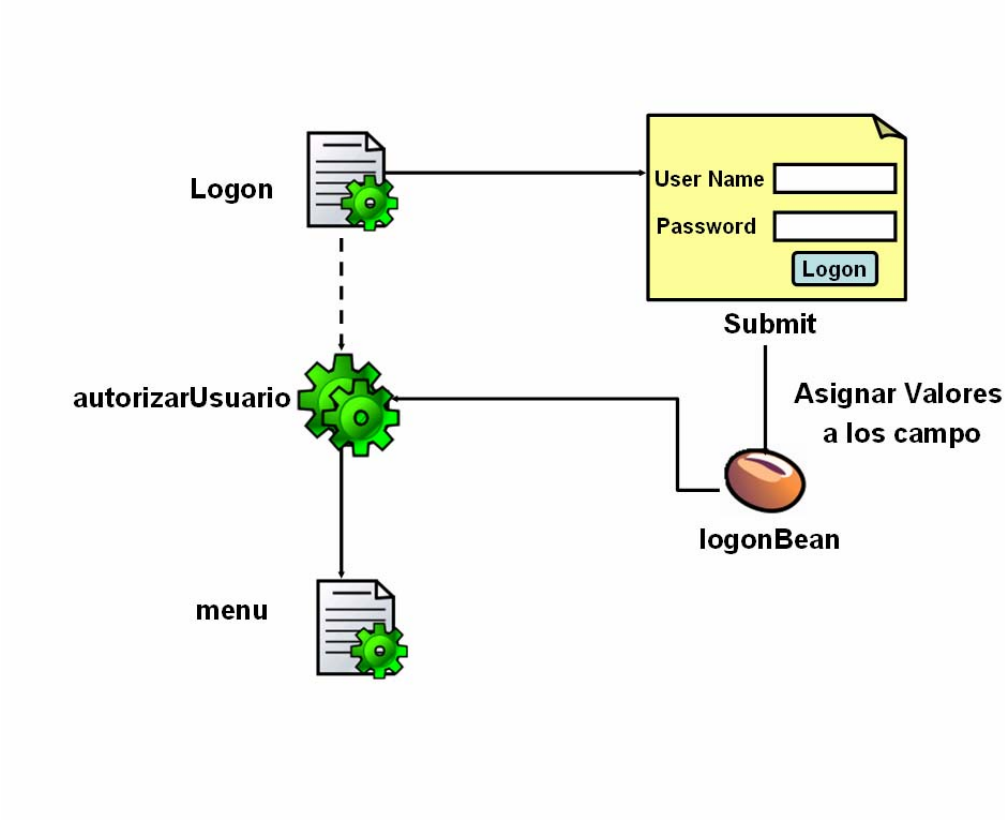
```
                HttpServletRequest request,
                HttpServletResponse response)
        throws IOException, ServletException
{
  if (request.getParameter(“usuario”).equals(“Scott”))
          return mapping.findForward("exito");
  else
          return mapping.findForward("fracaso");
}
```

En el ejemplo anterior, se obtiene el parámetro usuario del objeto *request* en el método execute(). Si el valor del parámetro usuario es igual a “Scott”, entonces la validación es satisfactoria y el método findForward() se utiliza para determinar el siguiente objeto al que debe enviarse el flujo de la aplicación, basándose en el nombre del forward “exito”. Si la validación falla, se genera el nombre del forward “fracaso”. La representación gráfica de ésta implementación se muestra en la figura 19.

4.3.7. *Form Beans*

Un *Form Bean* es una clase que esta asociada a una o más acciones. Cuando una página solicita la ejecución de una acción, el controlado sincroniza cada parámetro del objeto *request* con los campos de la clase *form bean*.

Figura 20. Asignación de valores en *form beans*



Fuente: Oracle JDeveloper 10g: Build Applicacions with ADF. Oracle University. 2005

En la figura 20 se aprecia de forma gráfica el flujo que sigue una aplicación cuando se envía parámetro desde una página. En la página de Logon el usuario ingresará sus datos, cuando presione el botón enviará la información al Action autorizarUsuario. Antes de ejecutar la acción, los datos enviados desde la página son asignados a atributos de la clase logonBean, de forma que dichos atributos pueden ser utilizados dentro de la acción.

El framework envía una referencia del form bean como parámetro al método execute() de la clase action y sus valores son accesibles a través de métodos get(). Esto incrementa la productividad del desarrollo ya que reduce la complejidad de consultar el objeto request para obtener los parámetros enviados por el usuario. Una característica importante es que se

pueden asignar valores iniciales al bean antes que la página sea mostrada, de manera que la página muestre los campos con valores iniciales, que se obtendrán del bean. Comparando los nombres de los campos de la forma HTML con los del bean. Así se logra que el proceso de asignación de valores funcione en ambas vías.

Los form beans tienen la función de transportar información entre una página y una acción.

Se pueden crear *form beans* estáticos, que son conocidos como JavaBeans, heredan la clase ActionForm y los cuales poseen métodos get y set para cada uno de los atributos que se desean soportar dentro del bean. Los *form beans* también pueden ser dinámicos. Estos son declarados en el archivo Struts XML utilizando un bean dinámico denominado DynaActionForm, el cuál no necesita programación explícita ya que su implementación es generada en tiempo de ejecución.

4.3.7.1. *Form beans* estáticos

Los form beans son implementados a través de una clase Java que hereda de la clase ActionForm. En esta clase se declaran atributos y un método get y set para cada atributo que será manejado por el form bean.

En esta clase se puede implementar un método reset() que inicializa los campos de la forma y un método validate() que retorna un objeto ActionErrors, en el cuál se pueden validar los datos ingresados en la página.

A continuación se presenta un ejemplo de la implementación de un form bean estático:

```
public class AutorizarUsuarioActionForm extends ActionForm
{
String usuario;
public String getUsuario()     { return usuario; }
public void setUsuario(String newUsuario)     { usuario=newUsuario; }
public void reset( ...
public ActionErrors validate( ...
}
```

En este ejemplo, el bean estático contiene un atributo llamado “usuario” y contiene métodos get y ser para dicho atributo. Se debe notar que el método validate() retorna un error, este error puede ser mostrado automáticamente en la página JSP que invocó al Action, agregando la etiqueta Struts <html:errors/> en dicha página.

Es necesario agregar una referencia del form bean en el archivo struts-config.xml. Para el ejemplo anterior, la referencia sería la siguiente:

```
<form-beans>
   <form-bean name="logonBean" type="view.AutorizarUsuarioActionForm"/>
 </form-beans>
```

Donde name es el nombre lógico con el cual se hará referencia al bean y type es la clase donde está implementado el form bean.

La principal ventaja de utilizar un form bean estático es porque permite definir validaciones personalizadas para los datos ingresados por el usuario y además de métodos para reiniciar los valores del bean. Si no se necesitan estas opciones, es recomendable utilizar un bean dinámico.

4.3.7.1.1. Ejemplo de uso de un *form bean* estático dentro de un *Action*

```
public ActionForward execute(
                ActionMapping mapping,
                ActionForm form,
                HttpServletRequest request,
                HttpServletResponse response)
        throws IOException, ServletException
 {
   AutorizarUsuarioActionForm bean = (AutorizarUsuarioActionForm) form;
   String usuario = bean.getUsuario();
   if (usuario.equals("Scott"))
     return mapping.findForward("exito");
   else
     return mapping.findForward("fracaso");
 }
```

Cuando se utilizan beans estáticos es necesario definir un objeto del tipo de la clase del bean, castear el parámetro form al tipo del form bean que corresponde y obtener el valor del atributo utilizando el método get correspondiente. En este ejemplo getUsuario().

4.3.7.2. *Form beans* dinámicos

Los form beans dinámicos son una alternativa para el manejo de formas cuando no se necesita hacer validaciones personalizadas.

Una ventaja de esta opción es que no se necesita tener un clase form bean y por lo tanto no hay necesidad de métodos get y set para cada

atributo. Los nombres de los campos son especificados en el archivo struts-config.xml y esto hace que nuevos campos puedan agregarse de forma declarativas sin necesidad de escribir código extra.

Esta versión de los *form beans* es definida al utilizar el tipo de forma correcto (org.apache.struts.action.DynaActionForm) en el archivo struts-config.xml y definiendo una etiqueta <form-property/> por cada campo en la clase bean dinámica. Se puede inicializar el valor de cada uno de los campos. No existe un método de validación para el bean dinámico. Para validad la información de un bean dinámico es necesario implementarla en el método execute() de la acción.

Para crear un form beans dinámico con nombre lógico logonBean y con un parámetro para almacenar el valor de “usuario” tipo String se debe agregar el siguiente código en el archivo struts-config.xml:

```
<form-bean name="logonBean"
    type="org.apache.struts.action.DynaActionForm">
<form-property name="username" type="java.lang.String"/>
```

4.3.7.2.1. Ejemplo de uso de un *form bean* dinámico dentro de un *Action*

```
public ActionForward execute(
                ActionMapping mapping,
                ActionForm form,
                HttpServletRequest request,
                HttpServletResponse response)
        throws IOException, ServletException
```

```
{
  DynaActionForm LAF = (DynaActionForm) form;
  String un = (String)LAF.get("usuario");
  if (un.equals("Scott")) {
    return mapping.findForward("exito");
  } else
    return mapping.findForward("fracaso");
}
```

En este caso se declara un objeto de tipo DynaActionForm y se castea el parámetro form a este tipo. Luego se utiliza el método get() del bean dinámico. Este método recibe como parámetro el nombre del atributo cuyo valor se desea consultar. En este ejemplo se obtiene el valor del atributo "usuario". Este método devuelve un objeto de tipo genérico, por lo que se debe transformar al tipo de objeto adecuado que, en este caso es un String.

4.3.8. *Data Actions*

Los *Data Actions* son extensiones (subclases) de la clase de Struts Action. Por defecto, proveen acceso a los servicios del negocio, así como a los view objects que son parte de los componentes del negocio. Los *Data Actions* son provistos por el framework Oracle ADF y no puede ser utilizando en programas que no utilicen esta tecnología.

Los Data Actions, tienen la función de preparar los datos que provienen de la capa del modelo y hacerlos disponibles para que las páginas los puedan consumir. De forma que un Data Action es frecuentemente utilizado para consultar datos a través de los view objects. El comportamiento por defecto del *framework* permite que los datos obtenidos por el Data Action sean trasladados al path (página JSP u otro Data Action)

defino en la etiqueta del forward para dicha acción (que por defecto es "success"). La figura 21 muestra de forma gráfica como interactúan los Data Actions con las páginas JSP.

Figura 21. Interacción entre un *Data Action* y una página JSP

En el caso de la figura 21, el Data Action crearDepartamento tiene asociado un método del modelo que permite crear un nuevo departamento. Si esta operación se realiza satisfactoriamente, el flujo de la aplicación continua y se dirige a la página ingresarDatosDept, en la cual se habilita una forma de ingreso que permitirá asignar valores a cada uno de los atributos del objeto construido por el Data Action.

La definición del Data Action descrito en la figura 21 en el archivo struts-config.xml se muestra a continuación:

```
<action path="/crearDepartamento"
 className="oracle.adf.controller.struts.actions.DataActionMapping"
type="oracle.adf.controller.struts.actions.DataAction" name="DataForm">
    <set-property property="modelReference" value="crearDepartamentoUIModel"/>
    <set-property property="methodName" value="crearDepartamentoUIModel.Create"/>
    <set-property property="resultLocation" value="${requestScope.methodResult}"/>
    <set-property property="numParams" value="0"/>
    <forward name="success" path="/ingresarDatosDept.do"/>
  </action>
```

En esta definición se puede identificar que el path con el cual se puede invocar el Data Action es "/crearDepartamento" y que el tipo de esta

clase es “oracle.adf.controller.struts.actions.DataAction”. También se aprecian las diferentes propiedades que indican el comportamiento del Data Action. Por ejemplo, la propiedad “methodName” indica el nombre del método asociado con el Data Action, que para el ejemplo, es crearDepartamentoUIModel.Create.

4.3.9. *Data Pages*

Una Data Page es una extensión (subclases) de una clase DataAction. Es la combinación de una data action y una página JSP. Este componente fue diseñado por Oracle para facilitar la interacción con los ADF BC en las páginas JSP.

Por ejemplo, en la figura 21 se puede apreciar un data action que ejecuta una operación para crear un registro y a continuación el resultado de esa operación es mostrado en la siguiente página. Una Data Page permitirá incluir el método de crear el registro y mostrar el resultado en un solo componente, permitiendo una simple identificación del flujo de la aplicación y la función que debe realizar cada componente.

Debido a que las Data Pages son extensiones de los data actions, es posible utilizar todos los componentes del modelo dentro de estas páginas y por lo tanto reutilizar componentes definidos en la capa de servicios del negocio.

Una Data Page se define en el archivo struts-config.xml y se debe asociar a un componente físico que permita mostrar los resultados. Este componente físico normalmente es una página JSP. Para el caso de la figura 21, la Data Page ingresarDatosDept se debe definir de la siguiente forma:

```
<action path="/ingresarDatosDept"
className="oracle.adf.controller.struts.actions.DataActionMapping"
type="oracle.adf.controller.struts.actions.DataForwardAction"
name="DataForm" parameter="/ingresarDatosDept.jsp">
    <set-property property="modelReference" value="ingresarDatosDeptUIModel"/>
  </action>
```

En donde la propiedad path define el nombre lógico con el cual se puede invocar a la página, el tipo del componente es “oracle.adf.controller.struts.action.DataForwardAction”. La página JSP asociada con la Data Page se encuentra identificada en el parámetro “parameter” que indica que la implementación física será “/ingresarDatosDept.jsp”.

En el caso de centralizar las operaciones de crear el departamento y mostrar el resultado en el mismo componente, es decir en una Data Page, la definición de la misma en el archivo struts-config.xml aparecerá de la siguiente forma:

```
<action path="/crearDept"
className="oracle.adf.controller.struts.actions.DataActionMapping"
type="oracle.adf.controller.struts.actions.DataForwardAction" name="DataForm"
parameter="/crearDept.jsp">
    <set-property property="modelReference" value="crearDeptUIModel"/>
    <set-property property="methodName" value="crearDeptUIModel.Create"/>
    <set-property property="resultLocation" value="${requestScope.methodResult}"/>
    <set-property property="numParams" value="0"/>
  </action>
```

5. INTEGRACIÓN DE LAS CAPAS DE ORACLE ADF

5.1. Definición de las capas

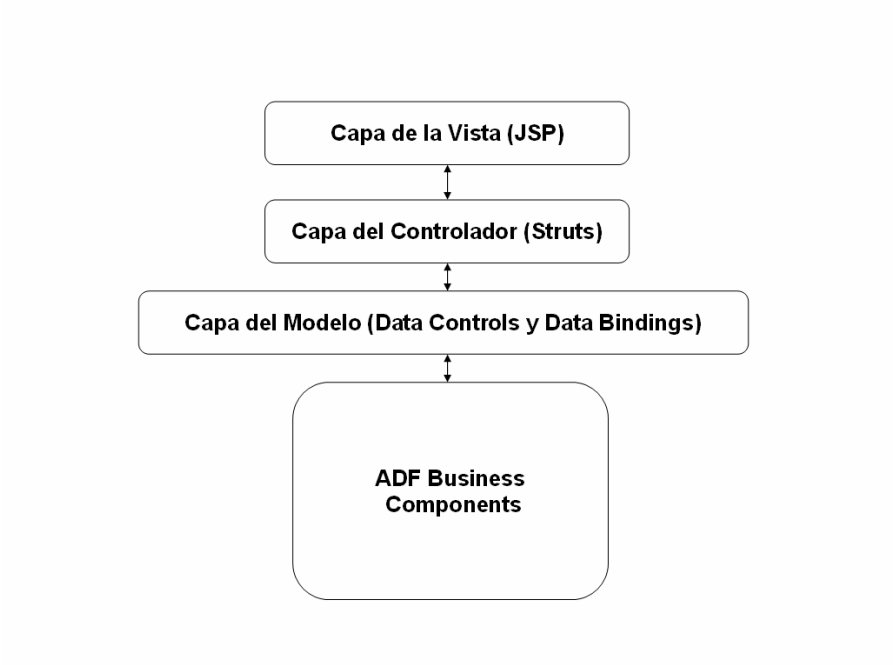
Cuando se está desarrollando una aplicación utilizando el patrón de diseño MVC, lo primero que se debe hacer es identificar que tecnología se utilizará para implementar cada una de las capas. En el caso de Oracle ADF, tomar esta decisión se simplifica, debido que el framework provee de componentes que facilitan la implementación.

Si se utiliza la tecnología por defecto que provee Oracle ADF, los componentes utilizados para implementar cada una de las capas sería: ADF BC, data bindings y data controls para implementar la capa del modelo, Struts para construir la capa del controlador y la capa de la vista se elabora con páginas JSP (figura 22).

Luego de definir la tecnología a utilizar para implementar cada capa, se debe definir que clases se construirán para implementar el modelo, cual será el flujo de la aplicación que se construirá y por último, cómo interactuarán los clientes con la información provista por el modelo a través del controlador.

Para comprender como se deben integrar las capas, se definirán los componentes necesarios para construir una aplicación que permite insertar un nuevo departamento en una base de datos que representa los diferentes departamentos que constituyen una empresa.

Figura 22. Integración de la capas en Oracle ADF utilizando tecnología por defecto



5.2. Capa del Modelo

Cuando se construye una aplicación utilizando Oracle ADF, la capa del modelo esta divida en la capa de los servicios del negocio o reglas del negocio y la capa del modelo.

La capa de los servicios del negocio para el ejemplo esta implementada con ADF BC, un entity object, un view object y un application module.

5.2.1. *Entity Object*

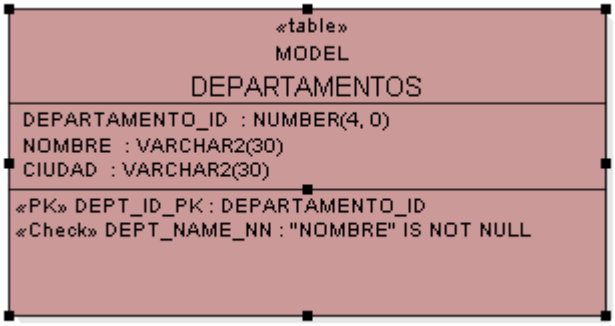
El entity object que permite manipular los datos se genera utilizando como fuente la tabla que reside en la base de datos del ejemplo. Esta tabla fue definida con las siguientes instrucciones:

```
CREATE TABLE departamentos
    ( departamento_id    NUMBER(4)
    , nombre  VARCHAR2(30)
          CONSTRAINT  dept_name_nn  NOT NULL
    , ciudad      VARCHAR2(30)
    ) ;
CREATE UNIQUE INDEX dept_id_pk
ON departamentos (departamento_id) ;

ALTER TABLE departamentos
ADD ( CONSTRAINT dept_id_pk
                    PRIMARY KEY (departamento_id)
     ) ;
```

La figura 23 muestra la representación gráfica de la tabla:

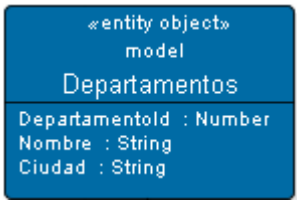
Figura 23. Tabla departamentos



Utilizando Oracle JDeveloper, se genera el Entity Object que permitirá manipular la información almacenada en esta tabla. Con esta clase se pueden realizar operaciones como crear, eliminar y actualizar registros, además de validar la información que se asigna a cada uno de los atributos de la tabla.

Gráficamente el Entity Object departamentos se representa como se muestra en la figura 24.

Figura 24. *Entity Object* Departamentos



El entity object esta compuesto por dos archivos, estos son el archivo XML Departamentos.xml y la clase de java DepartamentosImpl.java.

La clase de Java DepartamentosImpl contiene una declaración de constantes para hacer referencia a cada uno de los atributos de la clase:

```
public static final int DEPARTAMENTOID = 0;
public static final int NOMBRE = 1;
public static final int CIUDAD = 2;
```

El índice 0 representa a la columna departamento_id, el índice 1 a la columna nombre y el índice 2 a la columna ciudad.

Además, contiene métodos get y set para cada uno de los atributos, por ejemplo para el atributo departamentoid implementa los siguientes métodos:

```
public Number getDepartamentoId()
 {
   return (Number)getAttributeInternal(DEPARTAMENTOID);
 }

 public void setDepartamentoId(Number value)
 {
   setAttributeInternal(DEPARTAMENTOID, value);
 }
```

Estos métodos a su vez invocan a los métodos getAttibuteInternal y setAttributeInternal, los cuales son implementados por el contenedor J2EE en donde se ejecuta la aplicación. Estos métodos reciben como parámetro el índice de la columna que se desea manipular. En el caso de setAttributeInternal se envía también el valor que se asignará al atributo departamentoId. El método getAttributeInternal retorna un dato tipo objeto y este se debe convertir al tipo adecuado para permitir la manipulación.

El archivo XML Departamentos, contiene información sobre que fuente de datos esta representando el Entity object, que atributos manipula, declaración de validaciones, etc.

La etiqueta entity contiene información sobre la fuente de datos:

```
<Entity
  Name="Departamentos"
  DBObjectType="table"
  DBObjectName="DEPARTAMENTOS"
  AliasName="Departamentos"
  BindingStyle="Oracle"
  UseGlueCode="false"
  RowClass="model.DepartamentosImpl" >
```

En este caso, la fuente de datos es una tabla (DBObjectType) y su identificador en la base de datos es DEPARTAMENTOS.

Para definir los atributos del entity object se utiliza la etiqueta Attribute:

```
<Attribute
    Name="DepartamentoId"
    IsNotNull="true"
```

```
    Precision="4"
    Scale="0"
    ColumnName="DEPARTAMENTO_ID"
    Type="oracle.jbo.domain.Number"
    ColumnType="NUMBER"
    SQLType="NUMERIC"
    TableName="DEPARTAMENTOS"
    PrimaryKey="true" >
```

Para indicarle al contenedor J2EE sobre las validaciones que se deben hacer para un determinado atributo se utilizan distintas etiquetas contenidas dentro de la etiqueta Attribute correspondiente al atributo. Como ejemplo, se describe a continuación la forma en que se debería indicar una validación para el atributo ciudad:

```
<ListValidationBean
      OnAttribute="Ciudad"
      OperandType="LITERAL" >
      <AttrArray Name="List">
        <Item Value="GUATEMALA" />
        <Item Value="NEW YORK" />
      </AttrArray>
</ListValidationBean>
```

Se puede apreciar que se compara el atributo ciudad con una lista de valores, si el valor de este atributo es diferente que “GUATEMALA” o “NEW YORK” se considera invalido.

5.2.2. *View Object*

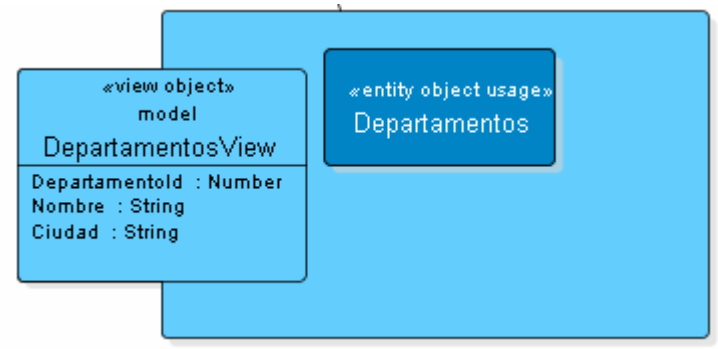
Un cliente no puede manipular directamente un Entity Object, únicamente lo puede hacer a través de un View Object. Los view object

restringen que información está expuesta para que pueda ser manipulada por los usuarios de la aplicación.

Continuando con el ejemplo y utilizando Oracle JDeveloper se debe construir un View Object que permita exponer el Entity Object Departamentos.

Este view object se nombrará DepartamentosView y su representación gráfica se puede apreciar en la figura 25.

Figura 25. *View Object* DepartamentosView



El diagrama indica que, a través de DepartamentosView se puede manipular la información del entity object Departamentos.

Un view object es una colección de objetos, en este caso objetos del tipo Departamentos. Un view object tiene asociada una consulta de SQL, la cual permite restringir y ordenar la información que se obtiene y ordenarla.

En el ejemplo, no se restringe ni se ordena la información que se obtendrá, por lo que la consulta de SQL en la cual esta basado el view object DepartamentosView será la siguiente:

```
SELECT Departamentos.DEPARTAMENTO_ID,
    Departamentos.NOMBRE,
```

```
    Departamentos.CIUDAD
FROM DEPARTAMENTOS Departamentos
```

Donde DEPARTAMENTOS es el nombre de la fuente de datos sobre la cual está construido el entity object Departamentos y los campos de la cláusula *select* son las columnas de la tabla.

El view object DepartamentosView esta formado por tres archivos, el archivo DepartamentosView.xml, la clase de java DepartamentosViewImpl y la clase de java DepartamentosViewRowImpl.

La clase de java DepartamentosViewImpl.java únicamente tiene esta estructura cuando se genera:

```
public class DepartamentosViewImpl extends ViewObjectImpl
{
  public DepartamentosViewImpl()
  {
  }
}
```

Y para este caso, únicamente sirve para crear una instancia del view object.

La clase de java DepartamentosViewRowImpl.java, tiene la misma estructura que la clase de java DepartamentosImpl.java y permite manipular de forma individual la información de cada registro obtenido por la consulta de SQL.

Por último, el archivo DepartamentosView.xml contiene información para indicarle al contenedor J2EE como debe manipular el view object y la forma en que puede interactuar con otros componentes.

En el archivo DepartamentosView.xml, la etiqueta ViewObject indica la consulta de SQL que se utiliza para obtener los datos, el nombre del view object y la clase que permite manipular cada registro obtenido entre otros. Esta definición se presenta a continuación:

```
<ViewObject
  Name="DepartamentosView"
  SelectList="Departamentos.DEPARTAMENTO_ID,
     Departamentos.NOMBRE,
     Departamentos.CIUDAD"
  FromList="DEPARTAMENTOS Departamentos"
  BindingStyle="Oracle"
  CustomQuery="false"
  RowClass="model.DepartamentosViewRowImpl"
  ComponentClass="model.DepartamentosViewImpl"
  MsgBundleClass="oracle.jbo.common.JboResourceBundle"
  UseGlueCode="false" >
```

El nombre del view object es indicado por el atributo “Name”, la consulta de SQL es lista en el atributo “SelectList” y la clase que permite manipular un registro de forma individual está referenciada en el atributo “RowClass”.

En el archivo XML también existen etiquetas para definir los atributos que estarán expuestos a las otras capas:

```
<ViewAttribute
    Name="DepartamentoId"
    IsNotNull="true"
```

```
PrecisionRule="true"
EntityAttrName="DepartamentoId"
EntityUsage="Departamentos"
AliasName="DEPARTAMENTO_ID" >
```

5.2.3. *Application Module*

El application module sirve como una sola vía de acceso a todos los componentes definidos en la capa de los servicios del negocio. Es en el application module donde se crean las instancias necesarias para hacer públicos los view objects. Además tiene asociada una conexión JDBC que permite acceder a la fuente de datos.

En el application module se pueden agregar métodos que implementen reglas específicas del negocio y publicarlos de tal forma que las otras capas los puedan invocar.

Este componente también permite administrar las transacciones de la aplicación. Debido a que todos los componentes que se utilizan en las otras capas tienen que ser accedidos a través del application module, todos ellos serán incluidos dentro de una sola transacción. Para cada sesión iniciada por un cliente.

Para el ejemplo que se describe, el application module únicamente contendrá una referencia a la instancia del view object DepartamentosView. Este application module tendrá el nombre de hrModule.

El application module está formado por dos archivos, un archivo XML que define información para que el contenedor J2EE pueda manipularlo y un

clase de java en donde se crean las instancias de los view object y se agregan métodos que implementan reglas del negocio.

La clase de java tiene el nombre de hrModuleImpl.java y contiene la siguiente estructura:

```
public class hrModuleImpl extends ApplicationModuleImpl
{
  public hrModuleImpl()
  {
  }

  public DepartamentosViewImpl getDepartamentosView1()
  {
    return (DepartamentosViewImpl)findViewObject("DepartamentosView1");
  }
 }
```

A través del método getDepartamentosView1 es que se expone el view Object DepartamentosView.

El archivo hrModule.xml contiene una etiqueta que indica cual es clase de Java en la que está implementa el application module:

```
<AppModule
   Name="hrModule"
   ComponentClass="model.hrModuleImpl" >
```

Adicionalmente, se incluyen etiquetas donde se describen los view objects a los cuales se puede tener acceso a través del application module y el nombre con el cual se pueden referenciar:

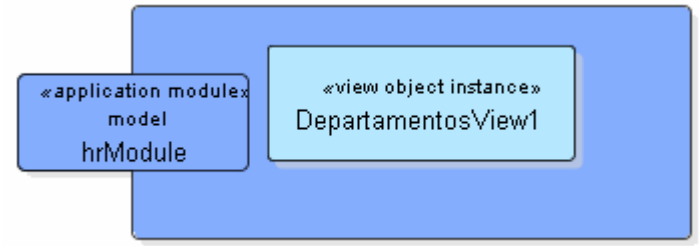
```
<ViewUsage
     Name="DepartamentosView1"
```

```
    ViewObjectName="model.DepartamentosView" >
  </ViewUsage>
```

En este caso, el atributo “ViewObjectName” indica que view object se esta publicando (model.DepartamentosView) y el atributo “Name” indica cual es el nombre con el cual se puede referenciar el view object (DepartamentosView1). Este esquema brinda la posibilidad de tener varias instancias del view object; cada una con diferente nombre y probablemente diferente funcionalidad.

Gráficamente el hrModule se visualiza de la siguiente forma (figura 26):

Figura 26. *Application Module* hrModule



5.2.4. *Data Controls* y *Data Bindings*

Los Data Controls y Data Bindings permiten que las capas de presentación y del controlador accedan a los servicios del negocio, independientemente de la tecnología utilizada para su construcción.

Los Data Controls y los Data Bindings son implementados a través de los archivos XML generados para cada uno de los componentes de la capa de servicios del negocio. Para el ejemplo, los Data Controls y los Data Bindings

están representados por los archivos Departamentos.xml, DepartamentosView.xml y hrModule.xml.

Cualquier componente que se incluya en la capa del modelo y que se necesite exponer a las otras capas deberá tener un archivo XML asociado que indique al contenedor J2EE y a las otras capas como debe ser manipulado.

5.3. Capa del controlador

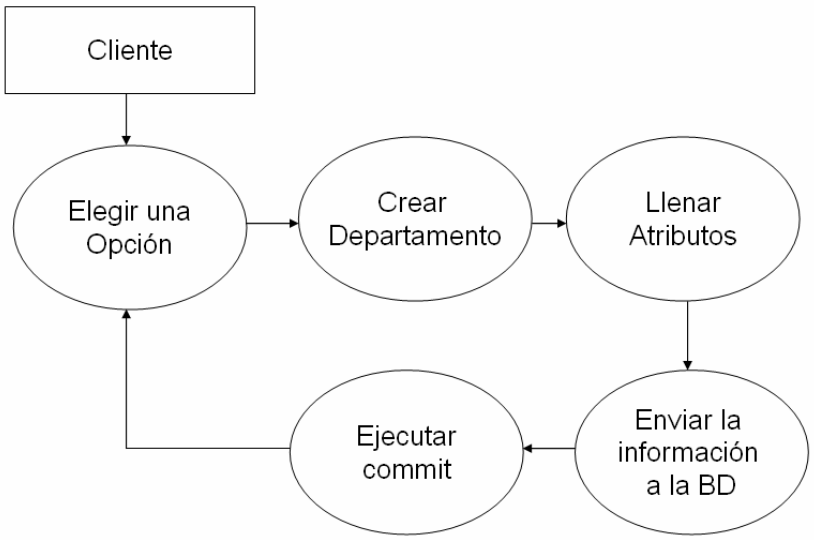
El controlador define el flujo de la aplicación y los procesos que estarán involucrados en la ejecución de una determinada tarea.

Prosiguiendo con el ejemplo, que busca ingresar un nuevo registro en la tabla de departamentos, el flujo de la aplicación sería el siguiente:

- Ingresar a la página de bienvenida y menú principal
- Elegir la opción de ingresar un nuevo departamento
- Crear el nuevo departamento
- Ingresar los datos del nuevo departamento
- Enviar la información a la base de datos
- Registrar los cambios de forma permanente (commit)
- Regresar al menú principal

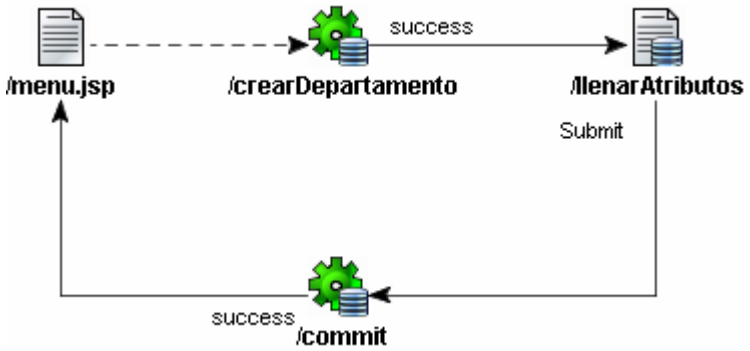
De forma gráfica el flujo de la aplicación se vería como la figura 27.

Figura 27. Flujo de una aplicación para creación de un departamento



Para implementar el flujo de la aplicación se utilizará Struts. El flujo de la aplicación utilizando esta tecnología se representa gráficamente en la figura 28.

Figura 28. Diagrama del flujo de una aplicación para creación de un departamento utilizando componentes de Struts



En la figura 28 se aprecian 4 componentes que implementan la funcionalidad de la aplicación y definen su flujo.

Se aprecia una página JSP en donde se construirá un menú, en el cual el usuario seleccionará la opción de crear un nuevo departamento. El siguiente componente es un Data Action, que ejecuta el método que crea el nuevo departamento. Luego existe tiene una Data Page que permitirá ingresar los atributos del nuevo departamento y contendrá un botón que enviará los nuevos valores a la base de datos. Este botón tiene asociado el evento “Submit”, el cuál dirigirá el flujo de la aplicación hacia el siguiente componente.

El último componente es un Data Action, que ejecuta la operación de commit definida dentro del application module. Al finalizar esta acción la aplicación presenta al usuario la página que contiene el menú principal.

El archivo struts-config.xml contiene las siguientes etiquetas que permiten identificar que realiza cada componente y como se redirecciona el flujo de la aplicación:

```
<action path="/crearDepartamento"
  className="oracle.adf.controller.struts.actions.DataActionMapping"
  type="oracle.adf.controller.struts.actions.DataAction" name="DataForm">
    <set-property property="modelReference" value="crearDepartamentoUIModel"/>
    <set-property property="methodName" value="crearDepartamentoUIModel.Create"/>
    <set-property property="resultLocation" value="${requestScope.methodResult}"/>
    <set-property property="numParams" value="0"/>
    <forward name="success" path="/llenarAtributos.do"/>
  </action>
```

En la etiqueta anterior se indica que el componente crearDepartamento es un Data Action y que ejecuta el método crearDepartamentoUIModel.Create (methodName) definido dentro del View Object DepartamentosView. Esta etiqueta también indica que si el método se ejecuta satisfactoriamente, se debe seguir el forward “success”, el cual

redirecciona el flujo de la aplicación hacia el siguiente componente, en este ejemplo llenarAtributos.do.

```
<action path="/llenarAtributos"
className="oracle.adf.controller.struts.actions.DataActionMapping"
type="oracle.adf.controller.struts.actions.DataForwardAction"
name="DataForm" parameter="/llenarAtributos.jsp">
    <set-property property="modelReference" value="llenarAtributosUIModel"/>
    <forward name="Submit" path="/commit.do"/>
  </action>
```

En esta etiqueta se indica que el componente que corresponde a la presentación de la data page llenarAtributos, será implementada en una página JSP que lleva el mismo nombre (parameter). También define que cuando se genere el evento “Submit”, el flujo de la aplicación se dirigirá al siguiente componente, que es commit.do.

```
<action path="/commit"
className="oracle.adf.controller.struts.actions.DataActionMapping"
type="oracle.adf.controller.struts.actions.DataAction" name="DataForm">
    <set-property property="modelReference" value="commitUIModel"/>
    <set-property property="methodName" value="commitUIModel.Commit"/>
    <set-property property="resultLocation" value="${requestScope.methodResult}"/>
    <set-property property="numParams" value="0"/>
    <forward name="success" path="/menu.jsp"/>
  </action>
```

Por último, el Data Action commit ejecuta el método commitUIModel.Commit definido en el application module hrModule y redirige el flujo de la aplicación a la página menu.jsp.

Oracle JDeveloper permite definir todos los componentes y asociarles los diferentes métodos de forma gráfica por lo que se facilita la construcción de esta capa.

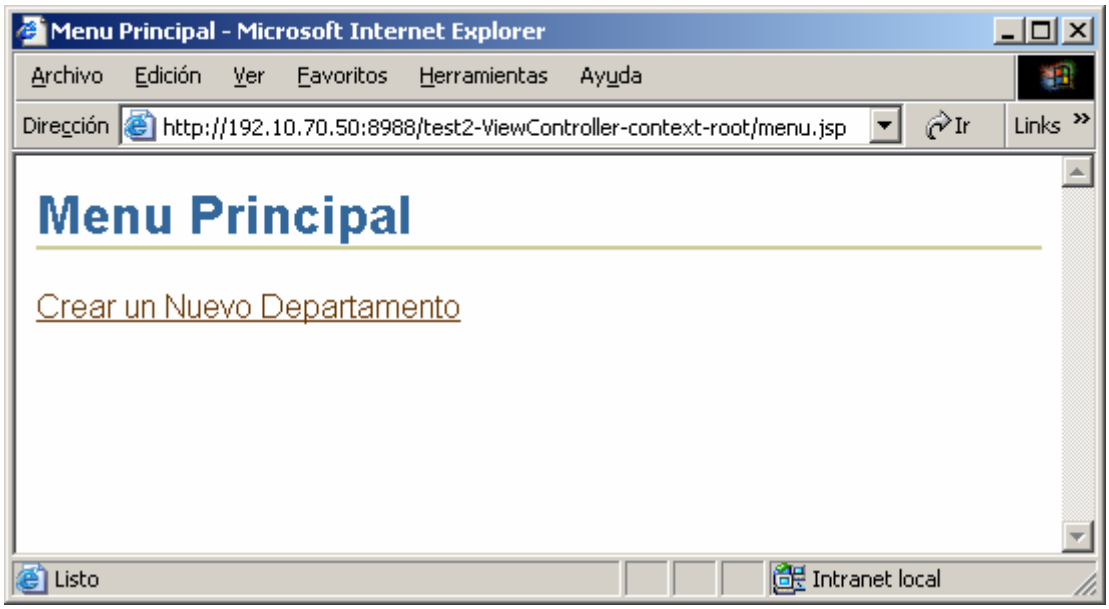
5.4. Capa de la Vista

Para concluir la construcción de la aplicación de ejemplo, se deben elaborar los componentes de la vista para que el cliente pueda interactuar con la misma.

Para este caso, se deben construir únicamente dos páginas JSP, una para el menú y otra para el componente físico de la data page llenarAtributos, donde el usuario ingresará la información del nuevo departamento.

La página menu.jsp contiene un vínculo único que el cliente puede visitar y que invocará el Data Action crearDepartamento. La figura 29 muestra la página menu.jsp.

Figura 29. Página menu.jsp



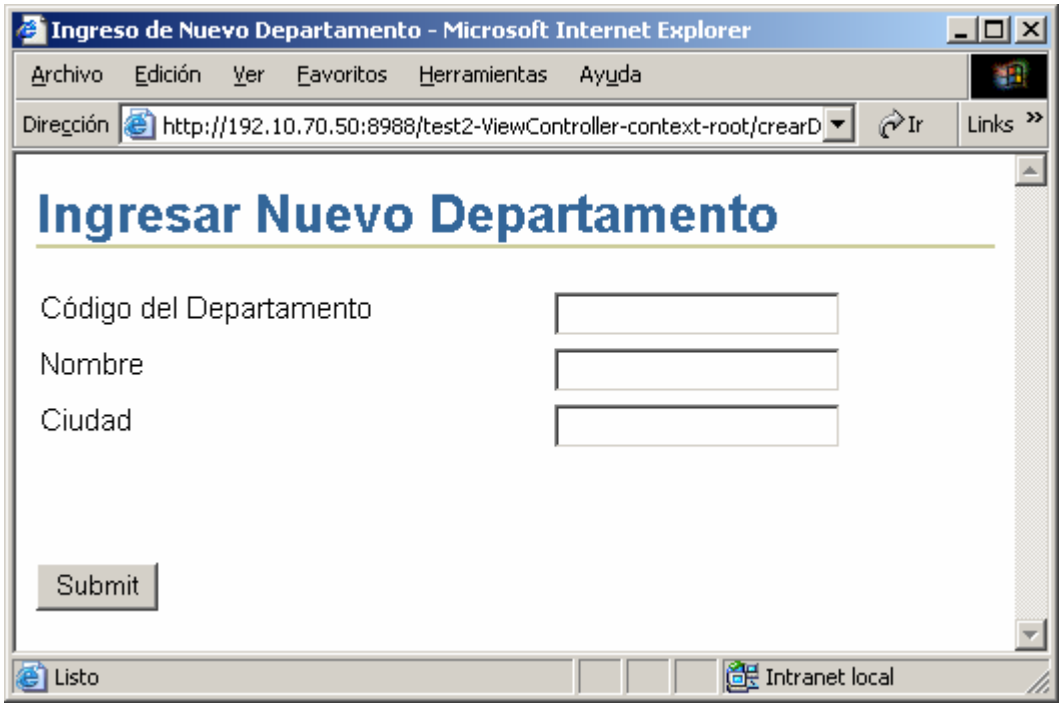
Esta página tiene el siguiente código en el área de la etiqueta body:

```
<body>
  <H1>Menu Principal</H1>
  <html:link page="/crearDepartamento.do">
    <bean:message key="link.crearDepartamento"/>
  </html:link>
</body>
```

Se utiliza una etiqueta de Struts html para crear el vínculo que invoca al Data Action crearDepartamento.do.

El siguiente componente es la página llenarAtributos.jsp que corresponde al componente físico del data page del mismo nombre. La figura 30 muestra la interfaz llenarAtributos.jsp.

Figura 30. Página llenarAtributos.jsp



Esta página contiene una forma de HTML y text fields para ingresar los valores del nuevo departamento. La forma de HTML contiene un botón (Submit) que genera el evento submit y que enviará los datos provistos por el usuario y el flujo de la aplicación al siguiente componente indicado en la capa del controlador.

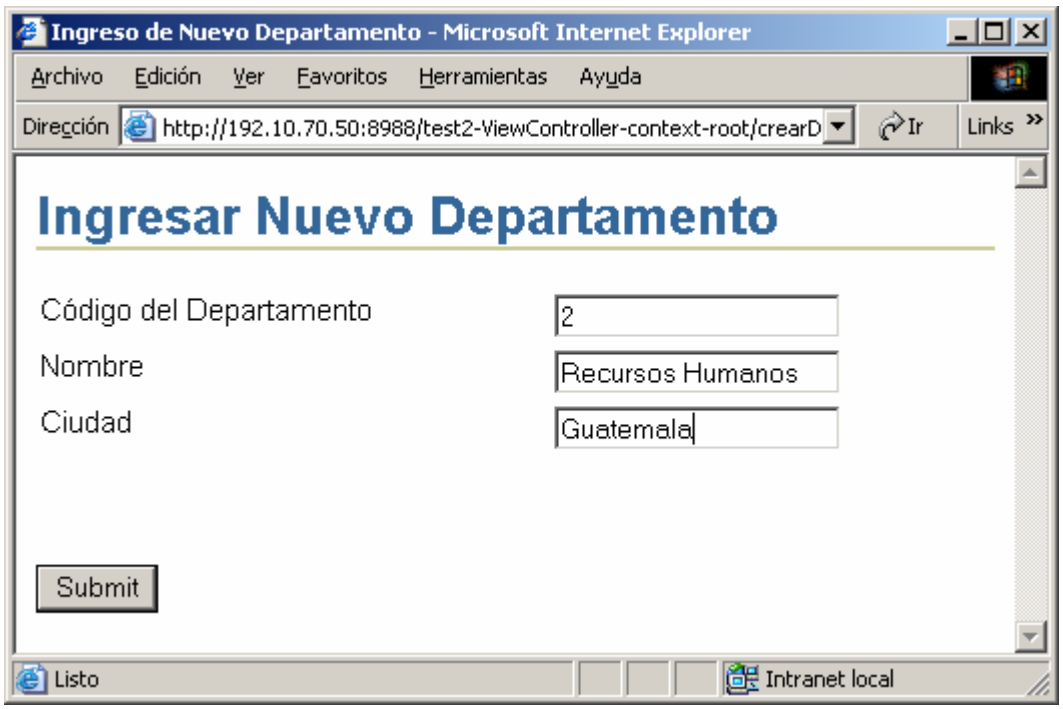
El código HTML de esta página se lista a continuación:

```
<body>
  <html:errors/>
  <H1>Ingresar Nuevo Departamento</H1>
  <html:form action="/llenarAtributos.do">
    <input type="hidden"
           name="<c:out value='${bindings.statetokenid}'/>"
           value="<c:out value='${bindings.statetoken}'/>"/>
    <table border="0" width="100%" cellpadding="2" cellspacing="0">
      <tr>
        <td>C&oacute;digo del Departamento</td>
        <td><html:text property="DepartamentoId"/></td>
      </tr>
      <tr>
        <td>Nombre</td>
        <td><html:text property="Nombre"/></td>
      </tr>
      <tr>
        <td>Ciudad</td>
        <td><html:text property="Ciudad"/></td>
      </tr>
    </table>
    <br/>
    <c:out value="${bindings.editingMode}"/>
    <br/>
    <br/>
    <input name="event_Submit" type="submit" value="Submit"/>
  </html:form>
```

</body>

Una vez ingresados los datos, como se muestra en la figura 31, el usuario debe presionar el botón etiquetado Submit y la información proporcionada será enviada hacia la base de datos.

Figura 31. Página llenarAtributos.jsp



Oracle ADF, a través de Oracle Jdeveloper, ofrece la posibilidad de generar la forma de ingresos incluida en la página llenarAtributos.jsp de manera automática, basándose en los Data Controls y Data Bindings de la capa del modelo definidos para el view object DepartamentosView.

6. CASO PRÁCTICO. DISEÑO E IMPLEMENTACIÓN DE UNA APLICACIÓN PARA INTERNET PARA LA ESCUELA DE CIENCIA Y SISTEMAS DE LA FACULTAD DE INGENIERÍA UTILIZANDO ORACLE ADF

6.1. Definición de problema

6.1.1. Antecedentes

Actualmente la Escuela de Ciencias y Sistemas de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, no cuenta con un proceso automatizado por medio del cual pueda llevar el control de su presupuesto, esto es, cuantas plazas tiene a su cargo, cuando devenga por cada una de ellas, que cantidad de horas tiene asignada cada plaza, quién ocupa cada plaza, etc.

Dicha administración del presupuesto, además de ser sencilla de utilizar, debe poder realizarse desde cualquier ubicación y no estar instalada en una sola máquina. Es por ello que se decidió implementarla a través de una aplicación para el Web, de tal forma que las personas encargadas de la administración del presupuesto podrán hacerlo desde cualquier máquina y desde cualquier ubicación.

El desarrollo de la aplicación se debe realizar utilizando una especificación que permita la construcción de aplicaciones con una arquitectura de componentes distribuidos de tal forma que sea escalable y que además puede ser publicada en cualquier plataforma y sistema operativo. Por esto se decide utilizar la especificación J2EE para implementar dicho desarrollo. Sin embargo, el desarrollo de aplicaciones

utilizando la especificación J2EE puede ser un proceso largo y tedioso, especialmente debido a que cada uno de los procedimientos básicos debe ser programado una y otra vez para satisfacer los requerimientos de cada proceso en específico.

Oracle ADF, es una framework de desarrollo que permitirá construir la aplicación utilizando la especificación J2EE, pero mejorando por mucho el tiempo de desarrollo y permitiendo que el enfoque se centre en las reglas del negocio y no en los detalles técnicos como el acceso a la base de datos y el manejo de transacciones, candados entre otros.

6.1.2. Identificación de requerimientos

Entre los requerimientos que se identificaron al hacer el análisis correspondiente se encuentra los que se listan a continuación:

- Mantenimiento de todas las entidades involucradas en los procesos, esto incluye creación, edición, eliminación y consultas de registros.
- Módulo para administración de contratos
- Módulo para administración de plazas
- Módulo para reprogramación de plazas por semestre de acuerdo a cierto presupuesto preestablecido.

De lo anterior se pudieron identificar los siguientes procesos:

- Contratación: Agregar personas nombradas, número de plaza y vigencia.
- Vencimiento de Contrato: Indicar que plazas dejarían de estar vigentes en la fecha de finalización del semestre.

- Permiso: Establecer la plaza como disponible tomando en cuenta la vigencia del permiso.
- Despido/renuncia: Establecer como disponible una plaza en el caso de despido o renuncia.
- Reprogramación: Tomar todas las plazas disponibles y sumar los salarios semestrales y poner el presupuesto restante para la creación de nuevas plazas.
- Mantenimiento de entidades (altas, bajas y cambios).

6.2. Definición de las entidades

A continuación se definen las entidades utilizadas para administrar la información y almacenar de forma eficiente.

6.2.1. Categoría

Con esta entidad se pretende representar la información de las categorías con las cuales una persona puede hacerse cargo de una plaza. Por ejemplo, Auxiliar de Cátedra I, Titular I, etc.

Dentro de esta entidad se manejará información del código de la categoría, una descripción o nombre de la categoría y el salario que devenga por hora al mes una persona que cumpla con un cargo de esta categoría.

6.2.2. Curso

Con esta entidad se representa la información de los cursos que se pueden cubrir con una plaza, por ejemplo Software Avanzado. En esta entidad se indicará el código del curso y su nombre.

6.2.3. Semestre

Esta entidad permite manipular la información acerca de los semestres en los cuales se asignan las plazas. Maneja información acerca del año, el número de semestre, el presupuesto asignada para dicho semestre y la fecha de inicio y finalización del mismo.

6.2.4. Horario

Horario es una entidad que representa la información de los horarios asignados a una determina plaza para un semestre en particular. Expone información referente al número de plaza, el semestre y año en el cuál el horario es válido, el día y la hora de inicio y finalización del período.

6.2.5. Plaza

La entidad plaza permite obtener información acerca de las plazas de las cuales dispone la Escuela de Ciencias y Sistemas. Entre la información representada a través de esta entidad se encuentra un número de plaza y una cantidad de horas asigna a dicha plaza.

6.2.6. País

Información sobre países, se debe tomar en cuenta que existe la posibilidad de que alguno de los miembros del personal de la Escuela de Ciencias y Sistemas puede ser de un país extranjero. La información representada por esta entidad incluye el nombre del país y un código asignado por una secuencia.

6.2.7. Departamento

Información acerca de los departamentos de la República de Guatemala que incluye el nombre del departamento y un código asignado por medio de una secuencia. La información que se presenta a través de esta entidad se utiliza para asignar a la entidad personal (descrita posteriormente), una ubicación de nacimiento y de obtención de cédula para aquello empleados que tengan mayoría de edad.

6.2.8. Municipio

Información acerca de los diferentes municipios dentro de los departamentos de la República de Guatemala. Es entidad incluye información acerca del nombre del municipio, el departamento al que pertenece y un código para identificarlo de forma única el cuál se genera por medio de una secuencia. Esta entidad es utilizada con el mismo propósito que la entidad Departamento.

6.2.9. Personal

Entidad que representa el recurso humano de la Escuela de Ciencia y Sistemas. Se incluye información sobre el número de registro del empleado, el apellido y nombres, así como el apellido de casada en caso de tenerlo. Además se maneja información como el NIT (Número de Identificación Tributaria), número de cédula o pasaporte, departamento y municipio en donde se extendió la cédula, país de origen, departamento y municipio de origen (lugar de nacimiento), fecha de nacimiento, sexo, número de afiliación del IGSS, estado civil, teléfono, dirección de domicilio, nivel de estudio, grado académico más alto obtenido, número de colegiado, fecha de incorporación a la escuela y observaciones.

6.2.10. Atribuciones de una Plaza

Esta entidad es la más importante de todas, debido a que es la entidad que permite manipular información acerca de las plazas asignadas en un semestre en particular y la persona que tiene asignada dicha plaza y en que categoría. Manipula información sobre el semestre y el año en el que la plaza esta asignada, la categoría que se asignará y la persona que cubrirá la plaza. Se obtiene también, a través de esta entidad, información acerca de la fecha de inicio de la asignación de la plaza, y la de finalización del contrato. Una descripción de si la plaza se encuentra ocupada o disponible y por que razones. El curso que a la que se le asignará la plaza.

6.2.11. Permiso

Entidad que representa la aceptación de un permiso sobre una plaza, durante el tiempo que dura el permiso la plaza puede ser ocupada por otra persona, sin embargo una vez finalizado el permiso, la persona asignada previamente retoma su puesto. Representa información acerca de la plaza y la persona que solicita el permiso, la fecha de inicio y finalización del permiso.

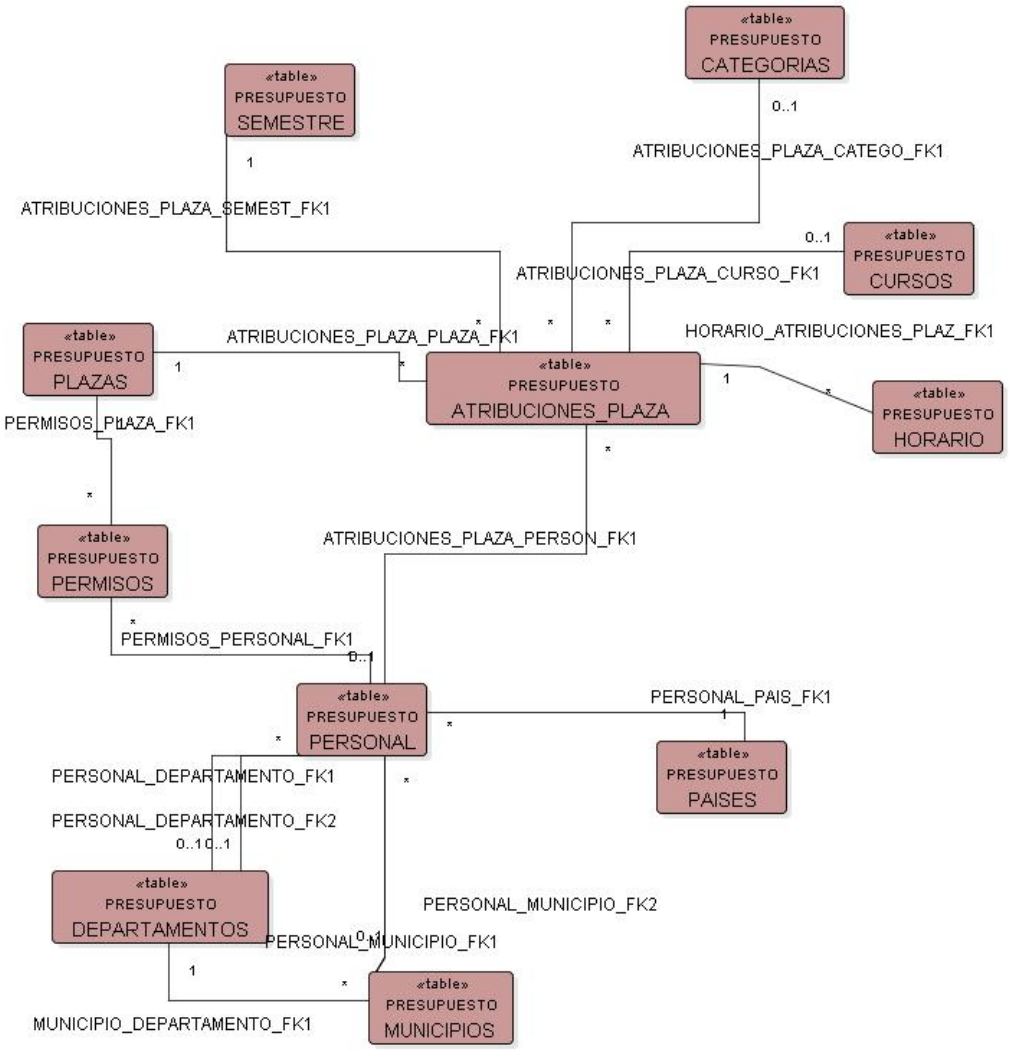
6.3. Modelo Entidad / Relación

El modelo que se presenta en la figura 32 permite apreciar las relaciones que existen entre las diferentes entidades. Estas relaciones permiten que, de forma conjunta, el sistema pueda administrar la información de tal forma que sea consistente y resumida.

La figura 32 muestra que, las la entidades cursos, países, departamentos, plazas y categorías servirán únicamente como catálogos,

mientras que las entidad personal, semestre y atribuciones plazas son entidades que interactúan para la programación de las plaza en determinado semestre.

Figura 32. Modelo Entidad / Relación



6.4. Diseño de la solución utilizando el patrón modelo-vista-controlador

Como se ha descrito, el patrón de diseño MVC, constituye una arquitectura eficiente para la construcción de aplicaciones en la Web. Provee una estructura de la aplicación basada en componentes y de esa manera permite la reutilización de los mismos. Soporta también la escalabilidad de la aplicación.

El patrón de diseño MVC se compone de tres capas, la capa del Modelo, la capa de la Vista y la capa del Controlador. Para el caso de la aplicación que se describe en esta sección y corresponde a la administración del presupuesto de la Escuela de Ciencias y Sistemas, se tiene previsto que el modelo sea implementado utilizando ADF Business Components, la vista sea implementada utilizando páginas JSP y el Controlador sea implementado utilizando Apache Struts.

Para implementar cada una de las capas se crearon clases dentro del paquete gt.usac.edu.ing.sistemas.presupuesto que corresponde a cada uno de los componentes.

6.4.1. Diseño del Modelo

Para implementar el diseño se utilizaron ADF Business Components. Cuando se implementa la capa del modelo utilizando ADF, esta se divide en tres tipos de componentes distintos, entre ellos los Entity Objects, los View Objects y el Application Module. Todas las clases correspondientes a la capa del modelo fueron creadas en el paquete

gt.usac.edu.ing.sistemas.presupuesto.model. A continuación se describen los componentes utilizados para implementar la presente capa.

6.4.1.1. *Entity Objects*

La figura 33 muestra el modelo utilizado para definir los Entity Objects que tomarían parte en la ejecución de los diferentes procesos de la aplicación. Como se puede apreciar en la figura 34, por cada entidad mencionada en una sección anterior de este mismo capítulo, se crea un entity object que corresponde a la misma estructura y que permitirá la manipulación de la persistencia.

En la figura 33 también se aprecian las relaciones que existen entre los diferentes entity objects y que son representaciones programadas de las relaciones que existen entre las diferentes entidades del modelo de datos.

6.4.1.2. *View Objects*

La figura 34 y figura 35 presenta el diagrama de los view objects utilizados en la aplicación de la administración del presupuesto.

Figura 33. Diagrama *Entity Objects*

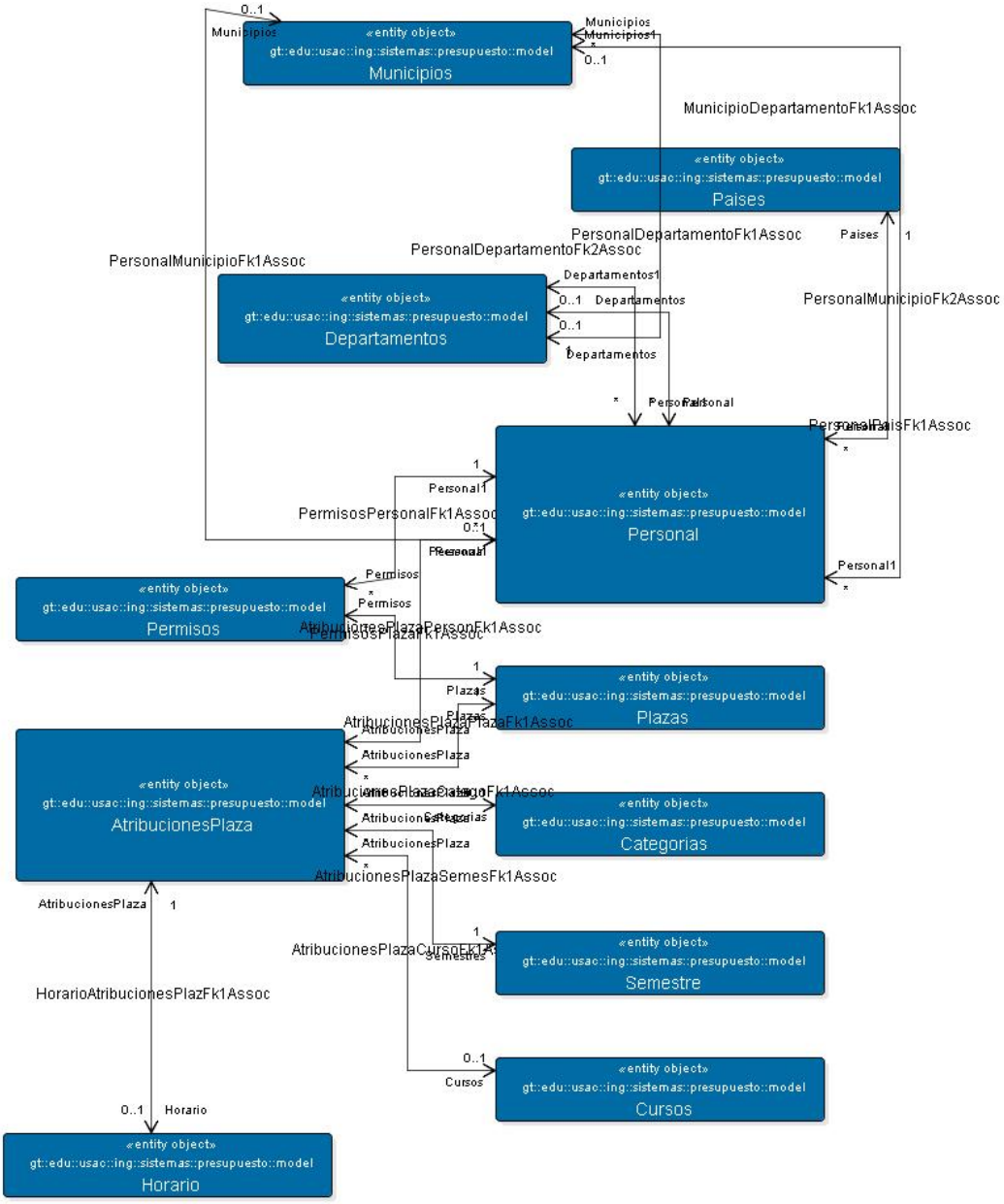


Figura 34. Diagrama *View Objects* por defecto

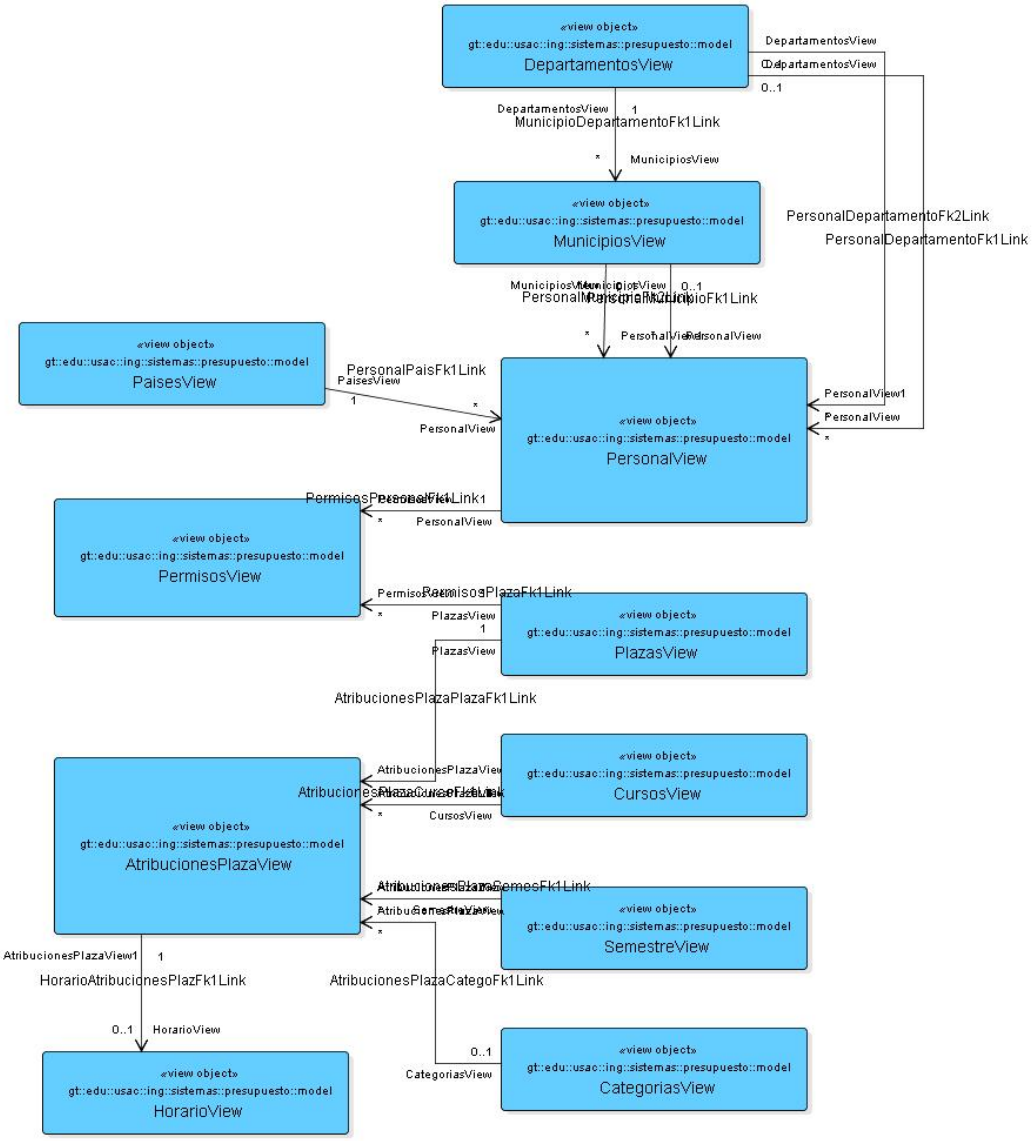
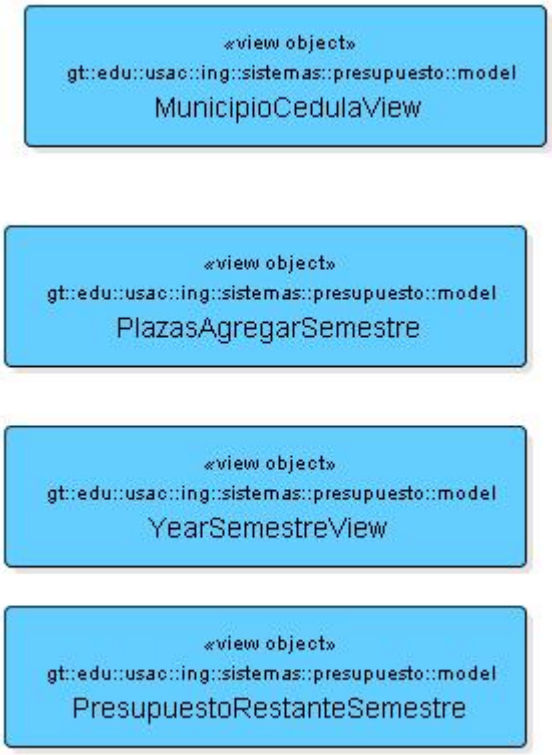


Figura 35. Diagrama *Entity Objects* personalizados



La consulta SQL que corresponde a cada uno de los view objects y una descripción de la forma en la que son utilizados es descrita a continuación.

6.4.1.2.1 AtribucionesPlazaView

Este view object es utilizado para manipular la información de la tabla de AtribucionesPlaza y mostrar las descripciones de cada una de las llaves foráneas que se crearon a nivel de la base de datos. La consulta SQL que permite cargar los datos desde la base de datos hacia el view object es el siguiente:

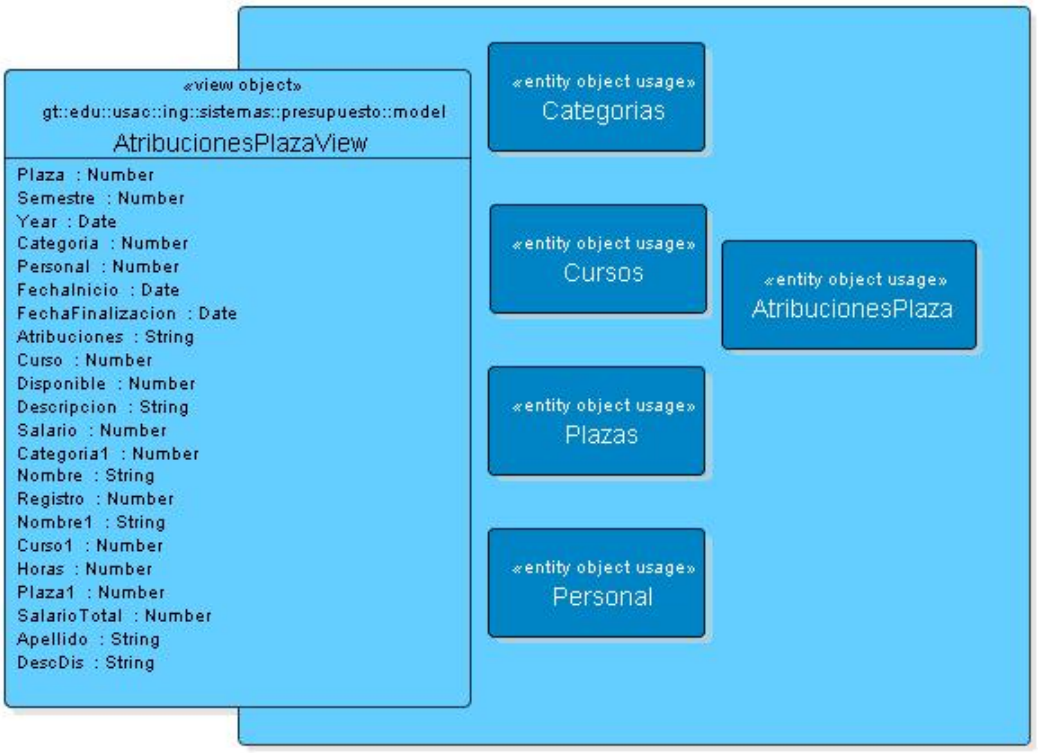
```
SELECT AtribucionesPlaza.PLAZA,
```

```
      AtribucionesPlaza.SEMESTRE,
      AtribucionesPlaza.YEAR,
      AtribucionesPlaza.CATEGORIA,
      AtribucionesPlaza.PERSONAL,
      AtribucionesPlaza.FECHA_INICIO,
      AtribucionesPlaza.FECHA_FINALIZACION,
      AtribucionesPlaza.ATRIBUCIONES,
      AtribucionesPlaza.CURSO,
      AtribucionesPlaza.DISPONIBLE,
      Categorias.DESCRIPCION,
      Categorias.SALARIO,
      Categorias.CATEGORIA AS CATEGORIA1,
      Personal.NOMBRE,
      Personal.REGISTRO,
      Cursos.NOMBRE AS NOMBRE1,
      Cursos.CURSO AS CURSO1,
      Plazas.HORAS,
      Plazas.PLAZA AS PLAZA1,
      Categorias.salario*Plazas.horas AS SALARIO_TOTAL,
      Personal.APELLIDO,
decode(AtribucionesPlaza.disponible,1,'Disponible',0,'Ocupada','Pre-Asignada') AS DESC_DIS
FROM (((
(ATRIBUCIONES_PLAZA AtribucionesPlaza
 LEFT OUTER JOIN CATEGORIAS Categorias
ON AtribucionesPlaza.CATEGORIA = Categorias.CATEGORIA)
 LEFT OUTER JOIN PERSONAL Personal
ON AtribucionesPlaza.PERSONAL = Personal.REGISTRO)
 LEFT OUTER JOIN CURSOS Cursos
ON AtribucionesPlaza.CURSO = Cursos.CURSO)
 LEFT OUTER JOIN PLAZAS Plazas
ON AtribucionesPlaza.PLAZA = Plazas.PLAZA)
```

Como se menciona, este view object es el núcleo de la administración del presupuesto, debido que es a través de este view object que se crean y modifican las plazas que se asignan en cada uno de los diferentes semestres.

La representación gráfica de este view object es la que se muestra en la figura 36.

Figura 36. AtribucionesPlazaView



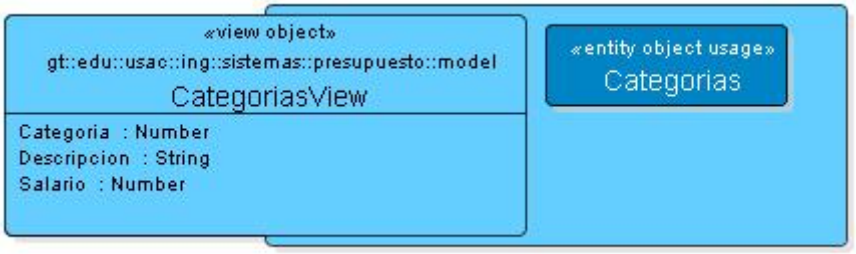
6.4.1.2.2. CategoriasView

Este view object sirve como catalogo para alimentar el view object de AtribucionesPlazaView. La consulta SQL utilizada para crear este view object es la siguiente:

```
SELECT Categorias.CATEGORIA,
       Categorias.DESCRIPCION,
       Categorias.SALARIO
FROM CATEGORIAS Categorías
```

Como se observa, permite manipular datos de la tabla de categorías creada en la base de datos. La representación gráfica de este view object se muestra en la figura 37.

Figura 37. CategoriasView



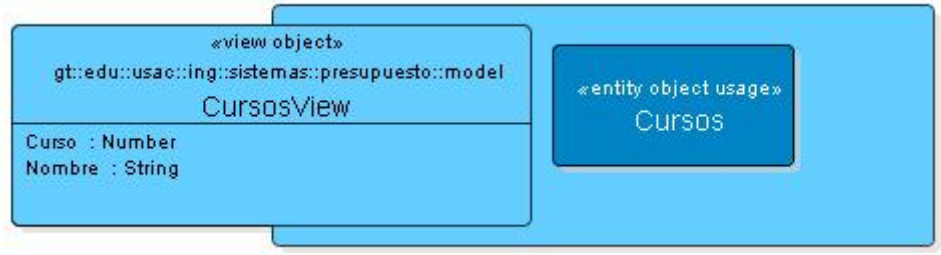
6.4.1.2.3. CursosView

Tiene el propósito de servir como catálogo para alimentar la tabla AtribucionesPlaza. La consulta SQL utilizada para crear este view object es la siguiente:

```
SELECT Cursos.CURSO,
       Cursos.NOMBRE
FROM CURSOS Cursos
```

La representación gráfica de este view object es la que aparece en la figura 38.

Figura 38. CursosView



6.4.1.2.4. DepartamentosView

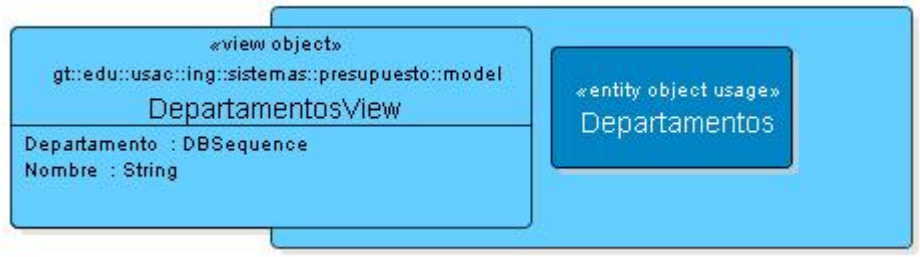
View object que permite obtener información acerca de los departamentos de la República de Guatemala. Sirve como catalogo para alimentar el lugar de nacimiento y el departamento en donde se ha extendido una cédula para la tabla de personal.

La consulta SQL utilizada para generar este view object es la siguiente:

```
SELECT Departamentos.DEPARTAMENTO,
    Departamentos.NOMBRE
FROM DEPARTAMENTOS Departamentos
```

La representación gráfica de este view object se muestra en la figura 39.

Figura 39. DepartamentosView



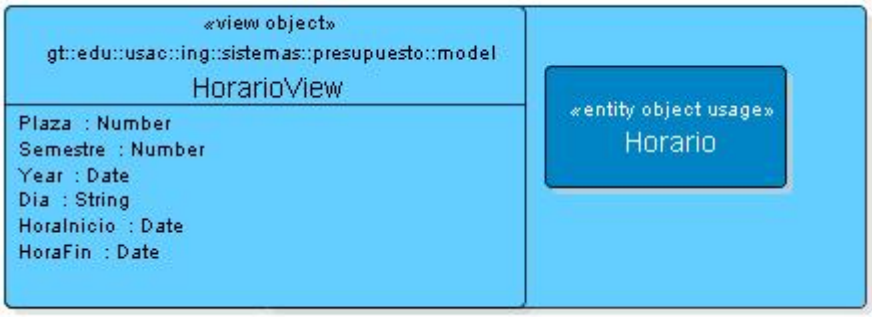
6.4.1.2.5. HorarioView

Representa los horario asignados a una determina plaza dentro de un semestre particular. La consulta SQL que se utilizó para generar el view object es la siguiente:

```
SELECT Horario.PLAZA,
    Horario.SEMESTRE,
    Horario.YEAR,
    Horario.DIA,
    Horario.HORA_INICIO,
    Horario.HORA_FIN
FROM HORARIO Horario
```

La representación gráfica de este view object se muestra en la figura 40.

Figura 40. HorarioView



6.4.1.2.6. MunicipioCedulaView

Este view object sirve como catálogo para alimentar el municipio de nacimiento y extensión de la cédula para la tabla de personal, fue creada de forma especifica para ser fuente de una lista de valores sincronizada con el view object de departamentos, no se utiliza para administrar la información de la tabla en la que esta basada. La consulta utilizada para crear este view object fue la siguiente:

```
SELECT Municipios.MUNICIPIO,
    Municipios.DEPARTAMENTO,
    Municipios.NOMBRE
FROM MUNICIPIOS Municipios
```

La representación gráfica de este componente se puede apreciar en la figura 41.

Figura 41. MunicipioCedulaView

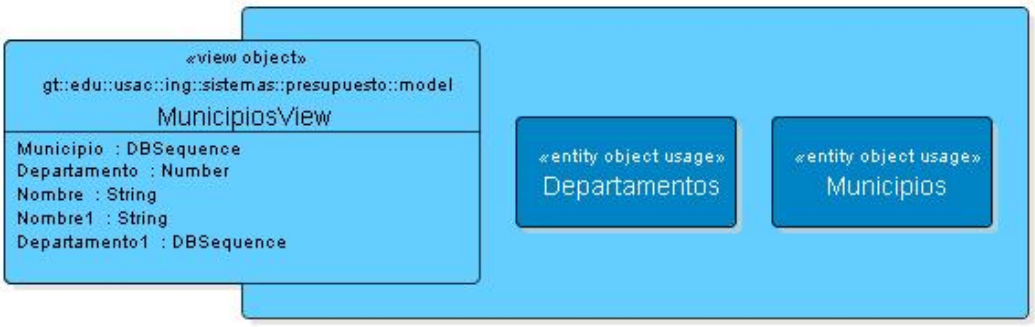
6.4.1.2.7. MunicipiosView

Al igual que el componente anterior, la información manejada por este view object sirve como fuente de datos de la tabla personal, sin embargo, a diferencia del componente anterior, este view object si permite la administración de la información de la tabla de municipios de tal forma que se puede insertar nuevos registros, modificar y eliminarlos de dicha tabla. La consulta SQL utilizada como referencia para crear este view object se muestra a continuación:

```
SELECT Municipios.MUNICIPIO,
    Municipios.DEPARTAMENTO,
    Municipios.NOMBRE,
    Departamentos.NOMBRE AS NOMBRE1,
    Departamentos.DEPARTAMENTO AS DEPARTAMENTO1
FROM MUNICIPIOS Municipios, DEPARTAMENTOS Departamentos
WHERE Municipios.DEPARTAMENTO = Departamentos.DEPARTAMENTO
ORDER BY Departamentos.NOMBRE,Municipios.NOMBRE
```

La representación gráfica de este componente se muestra en la figura 42.

Figura 42. MunicipiosView



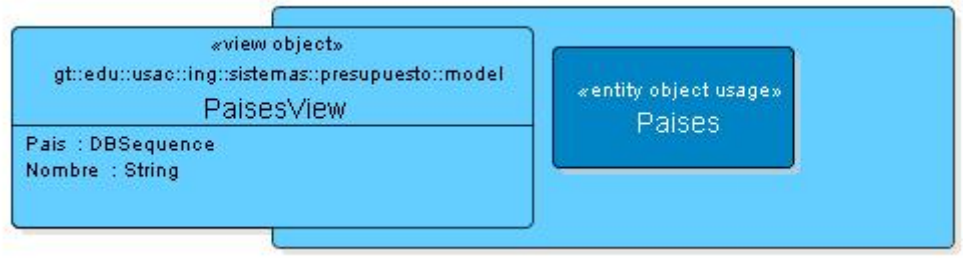
6.4.1.2.8. PaisesView

View object cuyo propósito es servir de catalogo para definir el país de origen de un determinado miembro del personal de la Escuela de Ciencia y Sistemas. La consulta SQL utilizada para generar el view object es la siguiente:

```
SELECT Paises.PAIS,
       Paises.NOMBRE
FROM PAISES Paises
```

La representación gráfica del view object se representa en la figura 43.

Figura 43. PaisesView



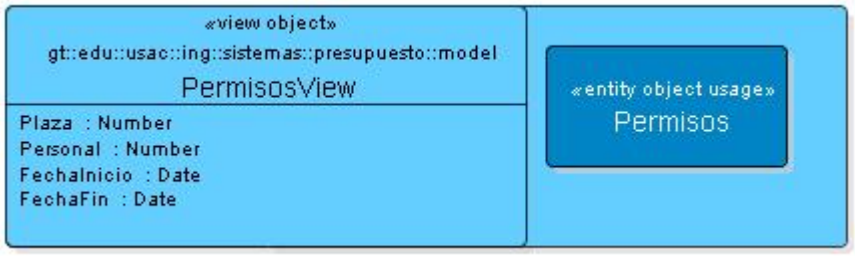
6.4.1.2.9. PermisosView

Este view object permite manipular la información correspondiente a la tabla de permisos. El view object está construido basado en la siguiente consulta SQL:

```
SELECT Permisos.PLAZA,
    Permisos.PERSONAL,
    Permisos.FECHA_INICIO,
    Permisos.FECHA_FIN
FROM PERMISOS Permisos
```

La representación gráfica de este view object se puede apreciar en la figura 44.

Figura 44. PermisosView



6.4.1.2.10. PersonalView

PersonalView es un componente que permite manipular la información de la tabla de personal. Este view object esta basado en la siguiente consulta SQL:

```
SELECT Personal.REGISTRO,  Personal.APELLIDO,
    Personal.APPELLIDO_CONYUGUE, Personal.NOMBRE,
```

```
    Personal.NIT, Personal.ORDEN_CEDULA,
    Personal.REG_CEDULA, Personal.DEPT_CEDULA,
    Personal.MUN_CEDULA, Personal.PAIS_ORIGEN,
    Personal.DEPT_NACIMIENTO,  Personal.MUN_NACIMIENTO,
    Personal.FECHA_NACIMIENTO, Personal.SEXO,
    Personal.AFILIACION_IGSS, Personal.ESTADO_CIVIL,
    Personal.TELEFONO, Personal.DOMICILIO,
    Personal.PASAPORTE, Personal.NIVEL_ESTUDIOS,
    Personal.TITULO, Personal.COLEGIADO,
    Personal.FECHA_CONTRATACION, Personal.OBSERVACIONES
FROM PERSONAL Personal
ORDER BY Personal.APELLIDO,
Personal.APPELLIDO_CONYUGUE,
Personal.NOMBRE
```

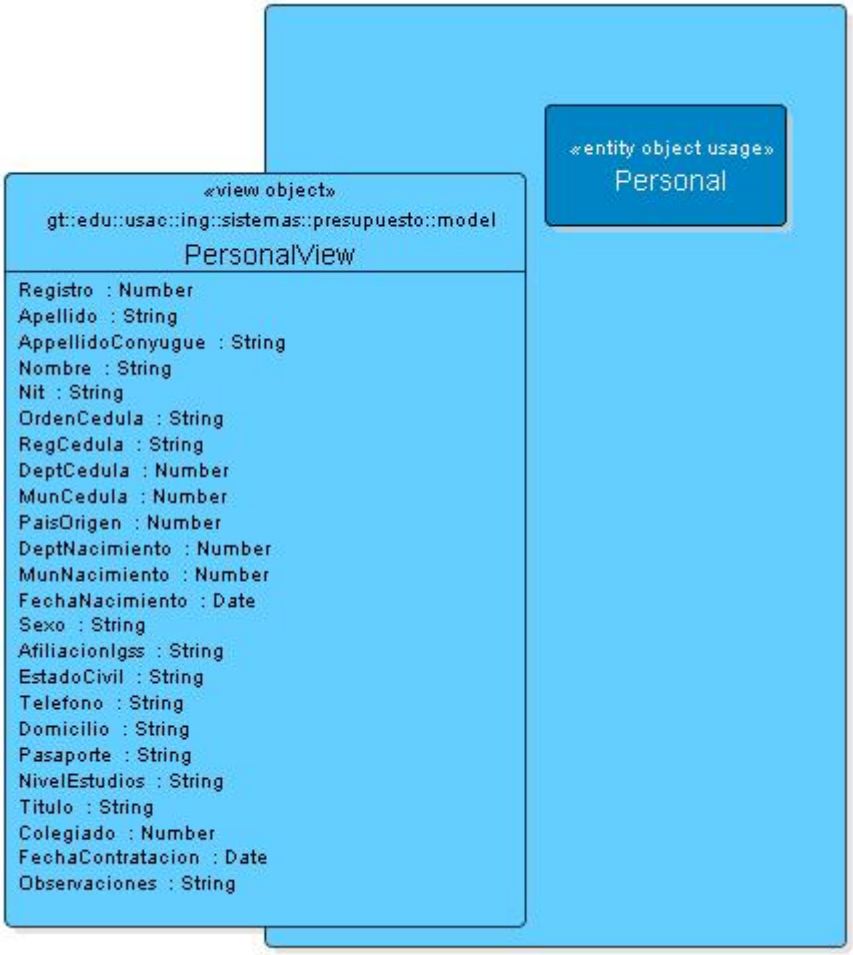
La representación gráfica de este componente se puede apreciar en la figura 45.

6.4.1.2.11. PlazaAgregarSemestre

Este view object fue creado con el propósito de generar un listado de aquellas plazas que no pertenece al semestre que se está actualizando, de tal forma que el usuario pueda escoger una de estas plazas, agregarla al semestre y luego asignarle los demás recursos. La consulta utilizada para generar este view object es la siguiente:

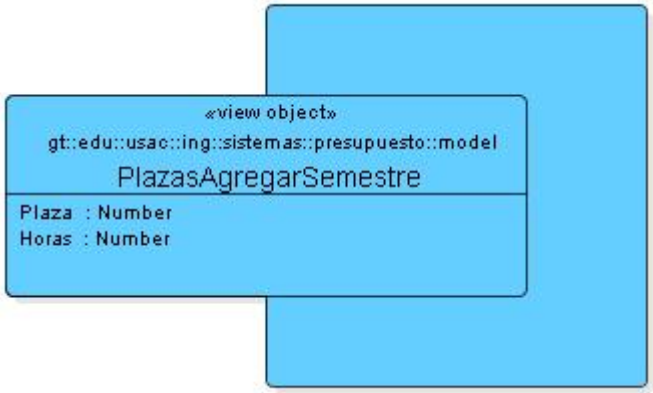
```
SELECT Plazas.plaza, Plazas.horas
FROM plazas Plazas
WHERE Plazas.plaza NOT IN
(SELECT Atri.plaza
 FROM atribuciones_plaza Atri
 WHERE atri.semestre = ?
 AND TO_CHAR(atri.year,'yyyy') = ?)
```

Figura 45. PersonalView



La representación gráfica de este view object es la que aparece en la figura 46.

Figura 46. PlazaAgregarSemestre



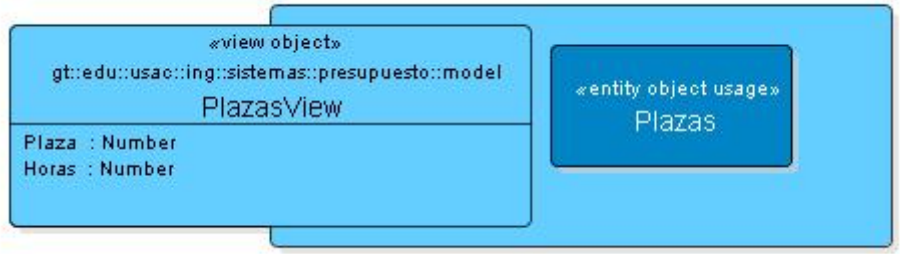
6.4.1.2.12. PlazasView

Permite manipular la información de la tabla plazas. Sirve como catálogo para alimentar la tabla AtribucionesPlaza. La consulta utilizada para generar este view object es el siguiente:

```
SELECT Plazas.PLAZA,
       Plazas.HORAS
FROM PLAZAS Plazas
ORDER BY Plazas.PLAZA
```

La representación gráfica se puede aprecia en la figura 47.

Figura 47. PlazasView



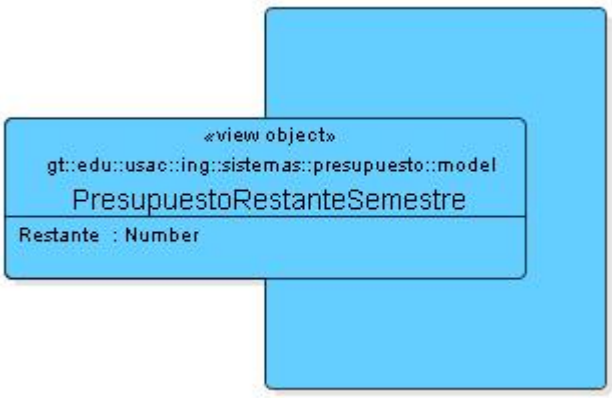
6.4.1.2.13. PresupuestoRestanteSemestre

Este view object fue creado para obtener el presupuesto restante de un determinado semestre. Este valor se obtiene en base a la sumatoria de los salarios devengados por cada una de las plazas asignadas al semestre, dicho resultado se resta al presupuesto asignado para dicho semestre. El view object fue generado en base a la siguiente consulta SQL.

```
SELECT Semestre.presupuesto - TotalSemestre AS Restante FROM
(SELECT SUM(Categorias.salario*Plazas.horas) TotalSemestre
 FROM CATEGORIAS Categorias, PLAZAS Plazas, ATRIBUCIONES_PLAZA    AtribucionesPlaza
WHERE Categorias.categoria = AtribucionesPlaza.categoria
AND Plazas.plaza = AtribucionesPlaza.plaza
AND TO_CHAR(AtribucionesPlaza.year,'YYYY') = ?
AND AtribucionesPlaza.semestre = ?
AND AtribucionesPlaza.disponible <> 1), SEMESTRE Semestre
WHERE TO_CHAR(Semestre.year,'YYYY') = ?
AND Semestre.semestre = ?
```

La representación gráfica de este componente se aprecia en la figura 48.

Figura 48. PresupuestoRestanteSemestre



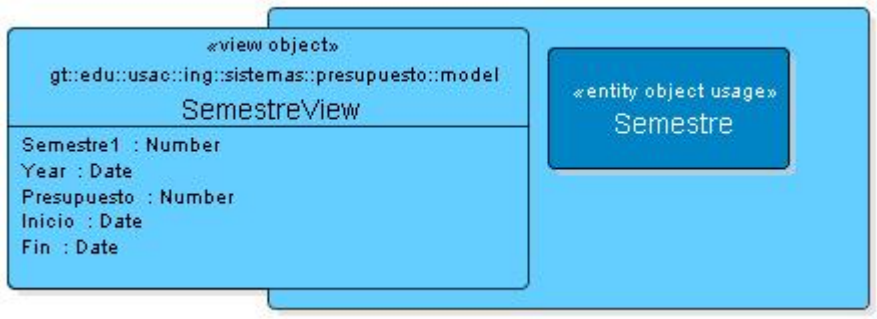
6.4.1.2.14. SemestreView

Este componente permite administrar la información de la tabla Semestre, la consulta SQL utilizada para generar el view object es la siguiente:

```
SELECT Semestre.SEMESTRE,
       Semestre.YEAR,
       Semestre.PRESUPUESTO,
       Semestre.INICIO,
       Semestre.FIN
FROM SEMESTRE Semestre
ORDER BY Semestre.YEAR,Semestre.SEMESTRE
```

La figura 49 muestra la representación gráfica de este componente.

Figura 49. SemestreView



6.4.1.2.15. YearSemestreView

Este view object fue generado de tal forma que permitiera generar una lista de valores de los años en los que se han creado semestres. Sirve para escoger el semestre que se administrará en los módulos de la aplicación.

La consulta SQL que genera la lista es la siguiente:

```
SELECT DISTINCT to_char(YEAR,'YYYY') year
FROM SEMESTRE
ORDER BY year desc
```

La figura 50 muestra la representación gráfica de este componente.

Figura 50. YearSemestreView

6.4.1.3. *Application Module*, PresupuestoAppModule

Este componente constituye la puerta de enlace entre el modelo y las demás capas del patrón de diseño MVC. Mediante este componente es que se exponen los view objects y algunos métodos personalizados para que los clientes puedan utilizarlos.

A través de este componente se instancia el datasource que sirve como enlace de la infraestructura lógica de la aplicación hacia el sistema de manejo de información, el cual, para el caso que se describe, es una base de datos relacional. El datasource que utiliza el application module es jdbc/usacDS.

Como se mencionó, es a través de este componente que se da acceso a ciertos métodos del negocio. Los métodos del negocio que fueron programados y expuestos a través del Application Module, son los siguientes:

- **agregarPlazaSemestre():** Este método recibe como parámetro un número de plaza que se desea que se agregue al semestre actual para que puede ser reprogramada.

- **buscarSemestre():** El método buscaSemetre() recibe dos parámetros, uno de ellos representa el número de semestre y el otro el número de año. A partir de estos parámetros se hace una consulta utilizando el view object SemestreView para recuperar los datos y plazas asignadas al semestre de interés. Si el semestre que se busca no existe despliega un mensaje de error.

- **calcularPresupuestoRestante():** Este método permite obtener el número de semestre que se está procesando y con esos datos invoca al view object PresupuestoRestanteSemestre para obtener el presupuesto restante en función de las plazas ocupadas para dicho semestre.

- **cambiarCriterioAtribucionesPlazaSemestre():** Permite filtrar el listado obtenido por el view object AtribucionesPlazaView a partir de cierto criterio que es enviado como parámetro.

- **cambiarCriterioListaEdicionMunicipios():** Permite filtrar el listado obtenido por el view object MunicipiosView a partir de cierto criterio que es enviado como parámetro.

- **cargarListaMunicipiosNacimiento():** Este método permite sincronizar las listas de valores del departamento de nacimiento con el municipio de nacimiento en la forma de ingreso de un empleado.

- **crearNuevoSemestre():** Invoca a la función crear_nuevo_semestre creada en la base de datos. La función crear_nuevo_semestre toma el último semestre que se ha creado y crea el siguiente semestre, poniendo disponibles las plazas que lo están y ocupando las plazas cuyo contrato no se ha vencido.

- **disponblePlazaPreAsignadas():** Por medio de este método se recorren todas las plazas del semestre que se está trabajando y se actualiza las plazas con estado preasignado a disponible.

- **obtenerListaPlazaAgregarSemestre():** Este método permite obtener la lista de las plazas que se puede agregar al semestre que se está manipulando. Si no existe más plazas que se puedan asignar se devuelve un mensaje al usuario para que cree en primera instancia una nueva plaza.

- **ocuparPlazaPreAsignadas():** Por medio de este método se recorren todas las plazas del semestre que se está trabajando y se actualiza las plazas con estado preasignado a ocupado.

- **validacionesPersonal():** Este método valida la información que fue provista en el formulario de ingreso o edición de los empleados.

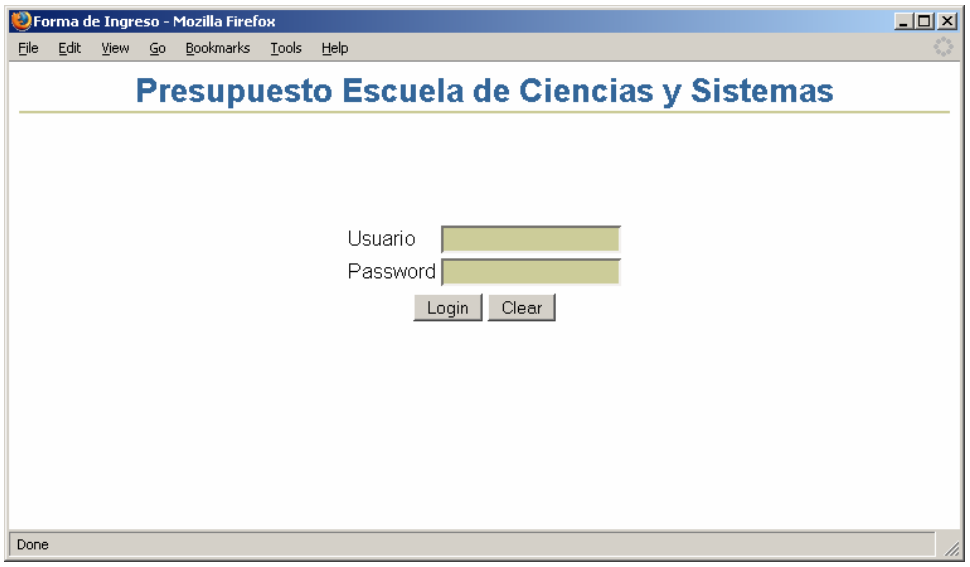
6.4.2. Diseño de la vista

Para implementar la vista se utilizaron páginas JSP, en esta sección se describe que información es desplegada o solicitada en cada página que se construyó para la aplicación.

6.4.2.1. login.html

Esta página corresponde a la página de inicio de sesión. La seguridad de esta aplicación es implementa utilizando el servicio de autenticación y autorización de Java (JAAS por sus siglas en inglés). Este módulo de seguridad requiere que se cree un página de inicio de sesión y una página de error para indicar que las credenciales provistas en la forma son incorrectas.

Figura 51. login.html



En la figura 51 se aprecia el aspecto visual de esta página.

6.4.2.2. errorlogin.html

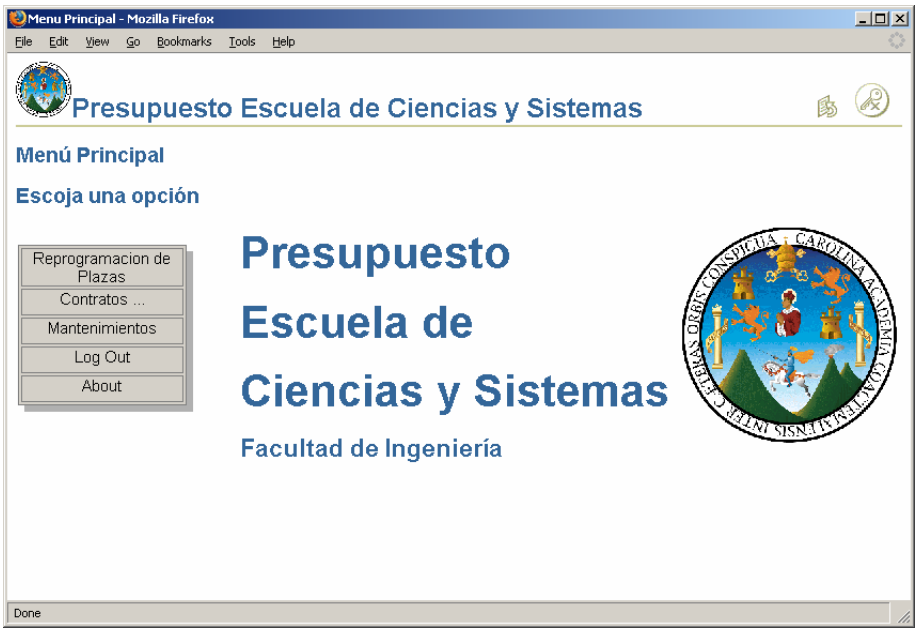
Esta página es utilizada para mostrar un mensaje de error si el usuario ingrese credenciales inválidas en el momento de intentar iniciar una sesión. La figura 52 muestra la página.

Figura 52. errorlogin.html

6.4.2.3. main.jsp

Página principal de la aplicación desde donde se accede al menú de opciones desde el cuál se accede a los diferentes módulos. La figura 53 muestra el aspecto visual de esta página.

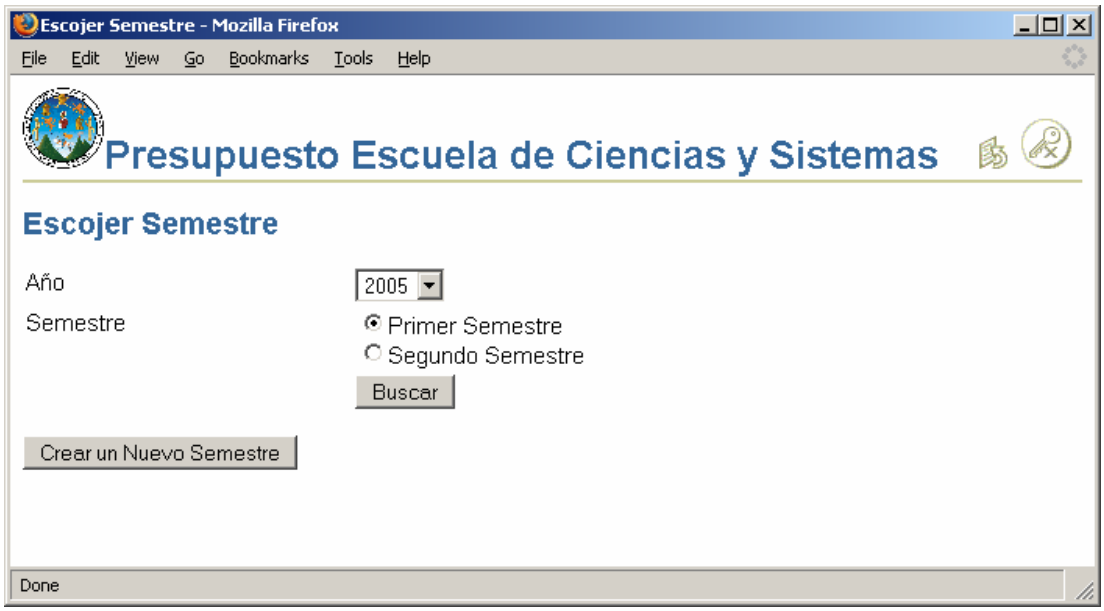
Figura 53. main.jsp



6.4.2.4. criterioSemestre.jsp

Esta página permite escoger el semestre con el que se desea trabajar. La figura 54 muestra la página.

Figura 54. criterioSemestre.jsp



6.4.2.5. datosSemestre.jsp

Muestra la información del semestre que se seleccionó en la página criterioSemestre.jsp, además del presupuesto asigna y restante, también muestra información sobre las plazas asignadas en dicho semestre. A partir de esta página se puede modificar los datos de las plazas, eliminar plazas del semestre, agregar nuevas plazas, entre otras operaciones. La figura 55 muestra el aspecto visual de este componente.

6.4.2.6. editSemestre.jsp

En esta página se puede modificar el presupuesto asignado para un determinado semestre. La figura 56 muestra la página.

Figura 55. datosSemestre.jsp

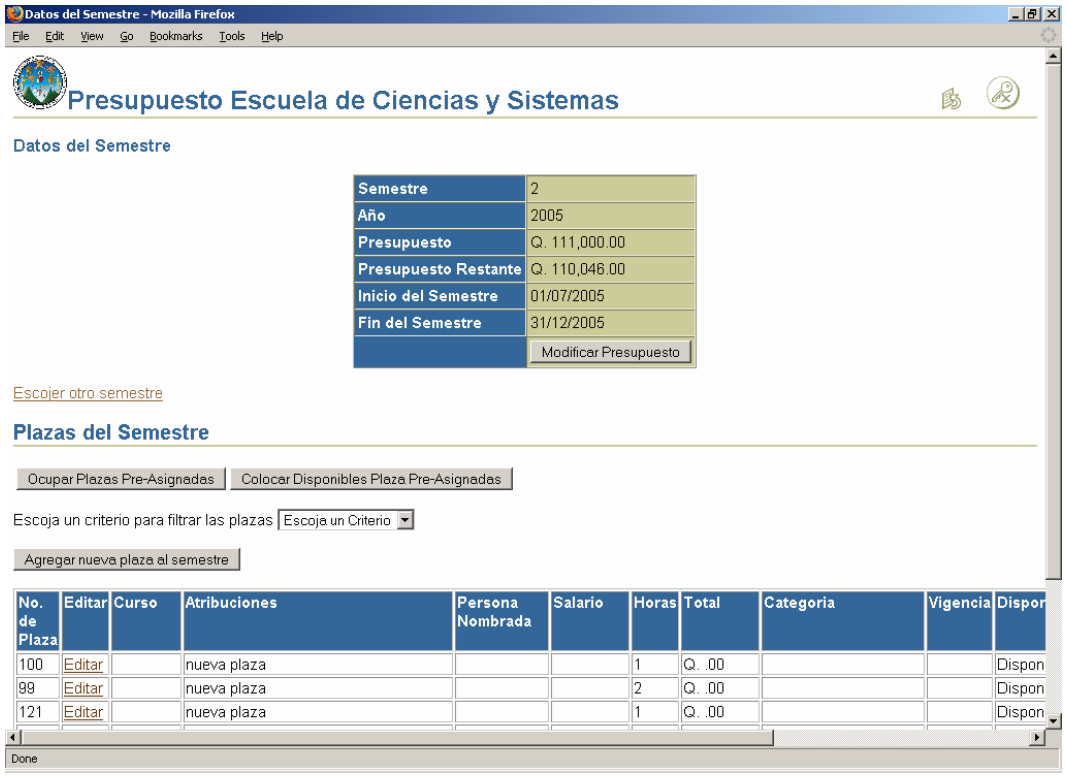
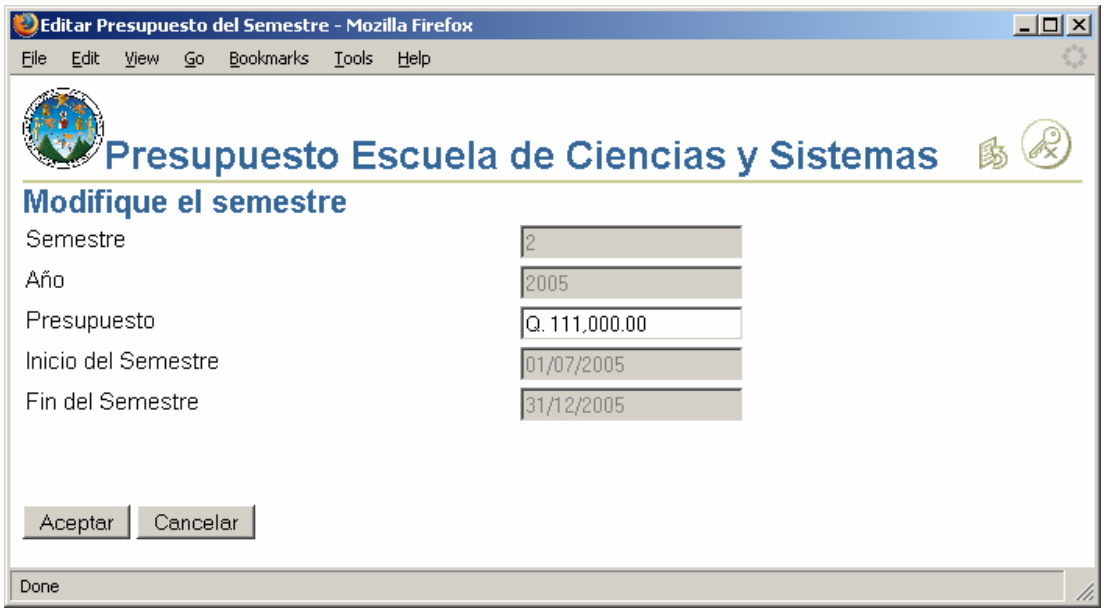


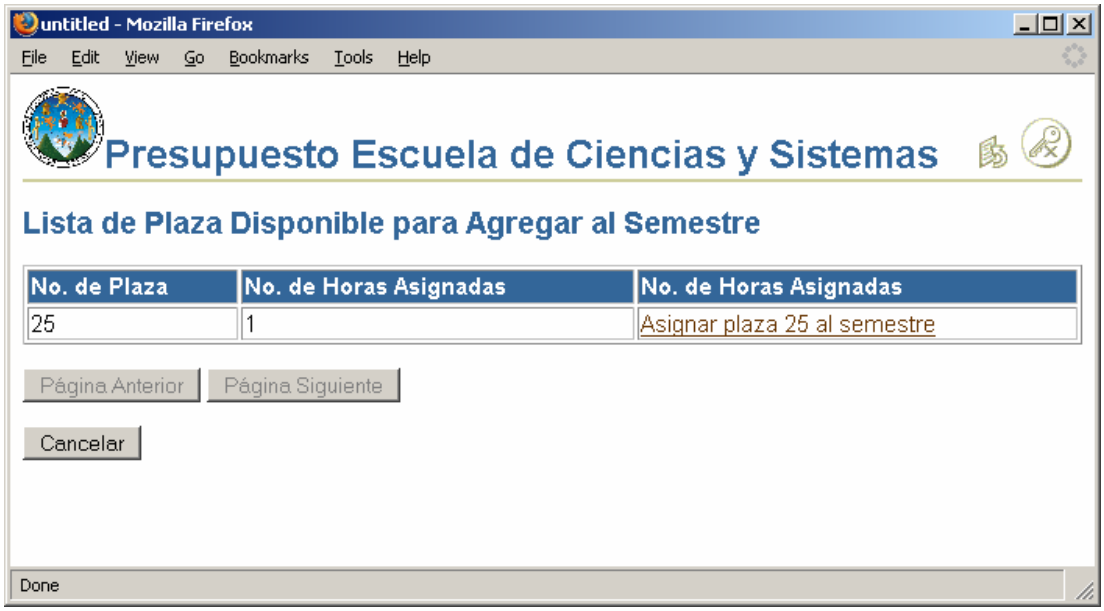
Figura 56. editSemestre.jsp



6.4.2.7. listaAsignarPlazaSem.jsp

Despliega un listado de las plazas que puede ser agregada al semestre. La figura 57 muestra el aspecto visual de la página.

Figura 57. listaAsignarPlazaSem.jsp



6.4.2.8. editarAtribucionPlaza.jsp

En esta página se modifican los datos de una plaza asignada en un determinado semestre. La figura 58 muestra interfaz visual de este componente.

6.4.2.9. crearPersonalNacional.jsp

Está página permite la edición y creación de los contratos de los empleados de la Escuela de Ciencias y Sistemas. La figura 59 muestra la página.

Figura 58. editarAtribuciones.jsp

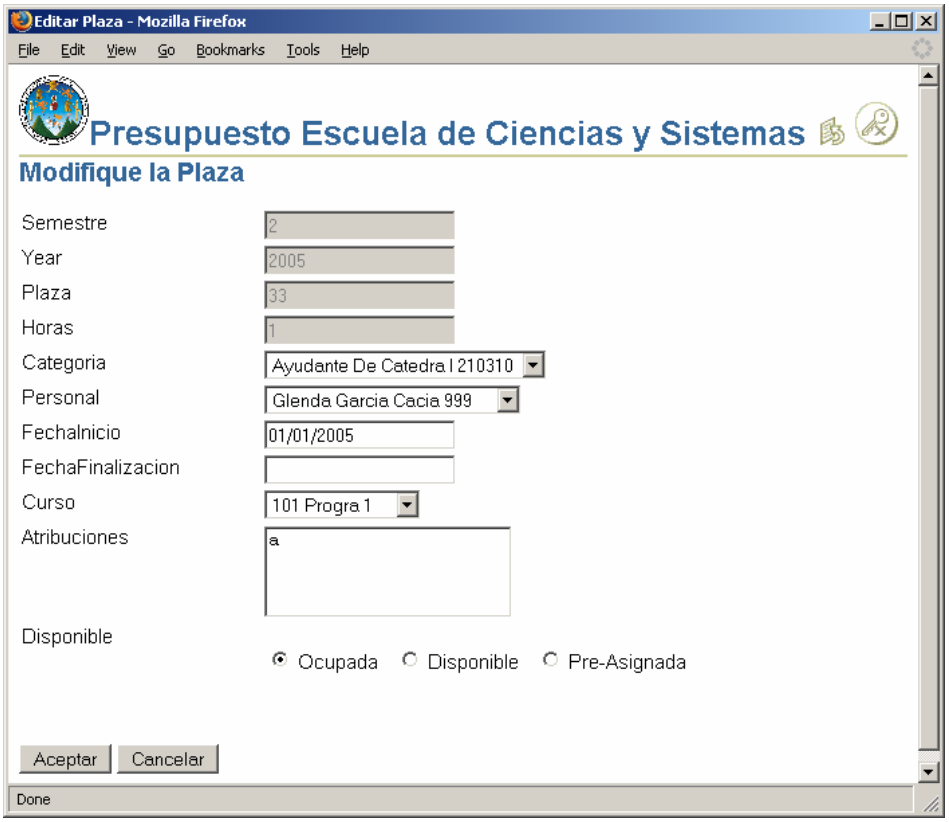
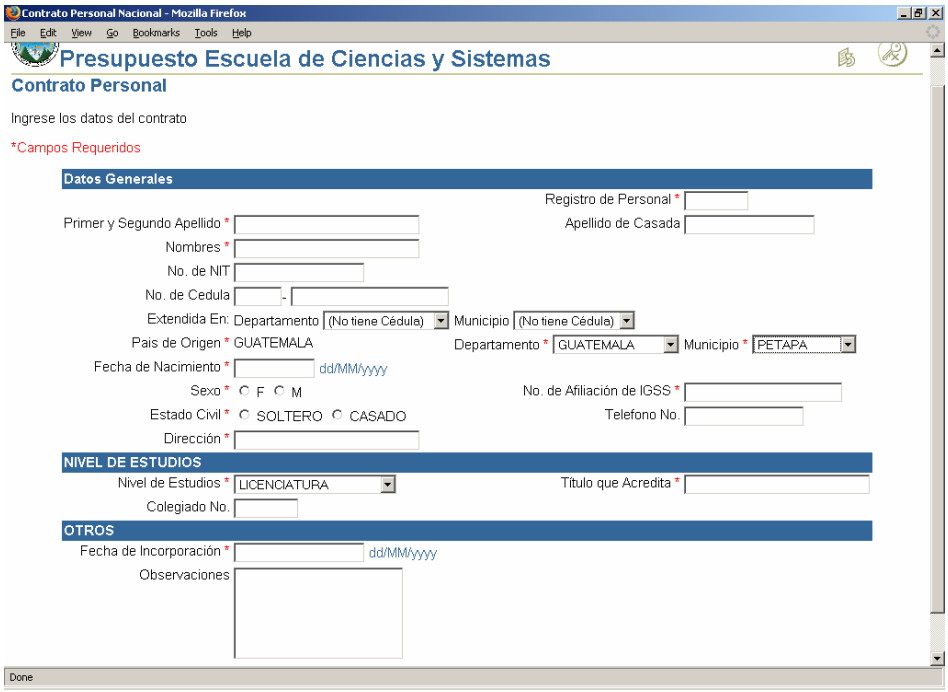


Figura 59. crearPersonalNacional.jsp



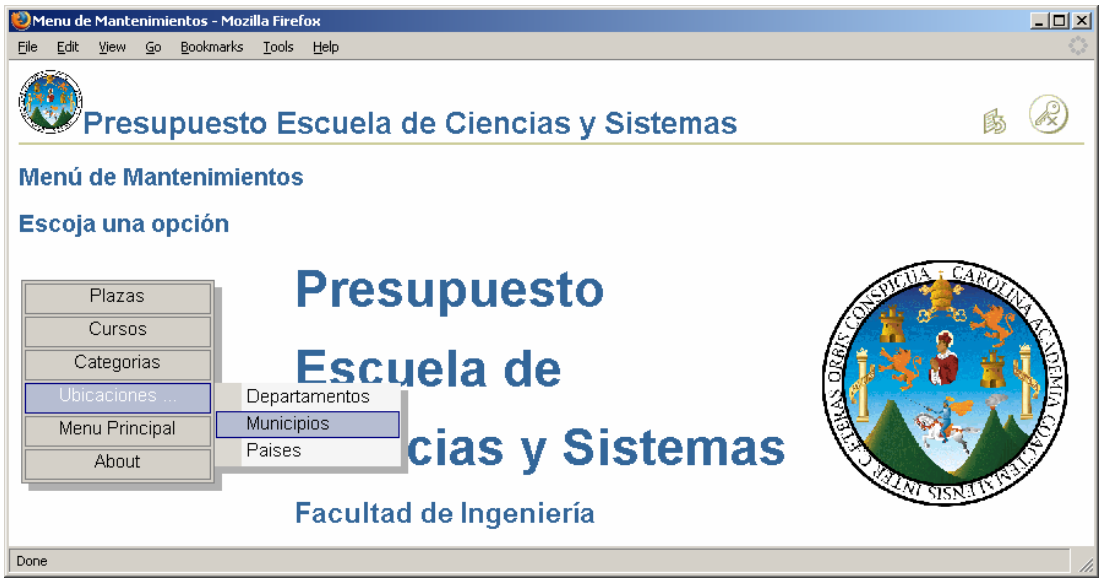
6.4.2.10. mantenimientos.jsp

Menú secundario desde el cual se puede acceder a las páginas de mantenimiento de las diferentes tablas que conforma el modelo de datos. El mantenimiento incluye obtener un listado, edición y eliminación de registros. La figura 60 muestra esta página.

6.4.2.11. Páginas de mantenimientos

El mantenimiento de las tablas esta compuesto por tres páginas. Una de ella presenta una lista (figura 61) desde la cuál existen vínculos para modificar los datos de cada registro, además desde esta página se puede invocar la página desde la cuál se puede crear un nuevo registro.

Figura 60. mantenimientos.jsp



La siguiente página corresponde a la edición o bien creación de registros, dependiendo la forma en la que se acceda a la página, un ejemplo genérico del aspecto visual de estas páginas se aprecia en la figura 62.

Figura 61. Mantenimientos, página de consulta

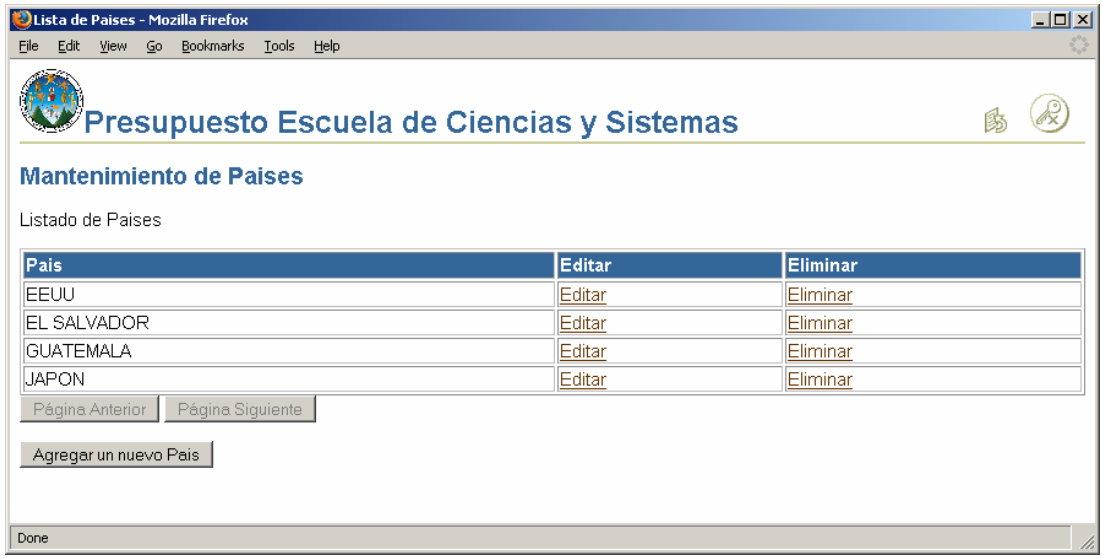
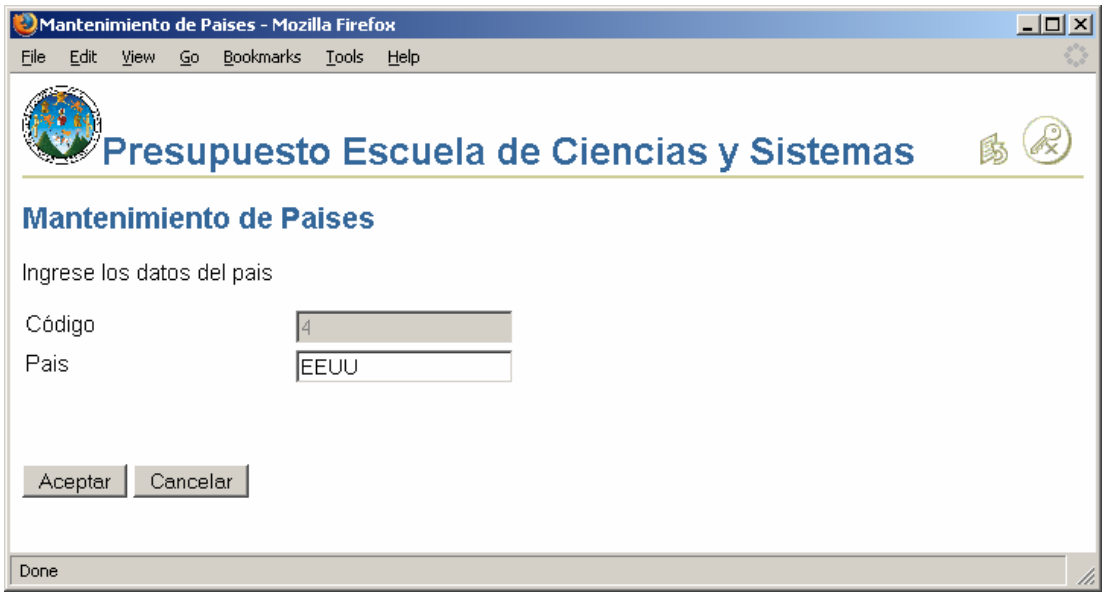
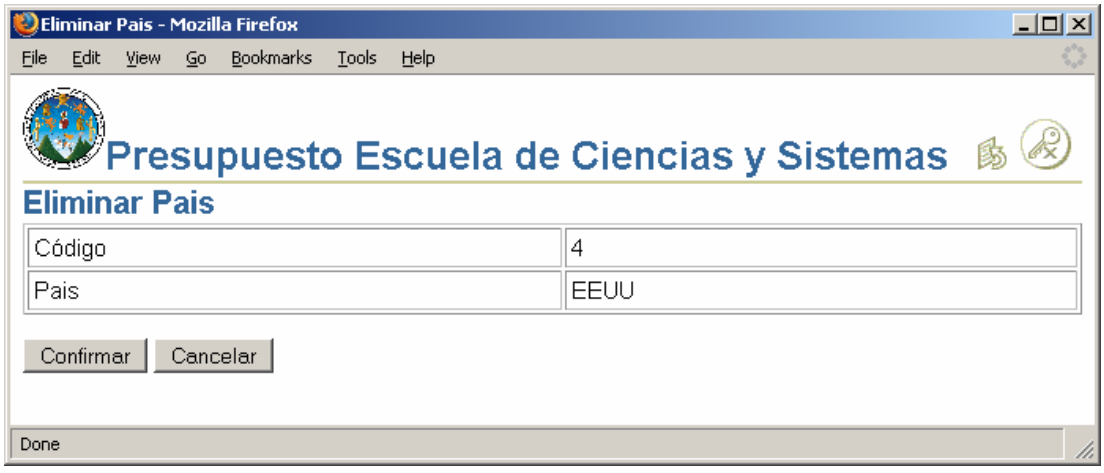


Figura 62. Mantenimientos, creación o edición de registros



La última página que compone el mantenimiento de tablas corresponde a una página de confirmación para la eliminación de un determinado registro. La página puede ser apreciada en la figura 63.

Figura 63. Mantenimientos, confirmación de eliminación



6.4.3. Diseño del Controlador

El controlador fue implementado utilizando Jakarta Struts y algunas clases extendidas de estos componentes propias de Oracle ADF. El

controlador corresponde al flujo de proceso de la aplicación y es la capa que debe integrar el modelo con la vista.

6.4.3.1. Definición del flujo de procesos

La aplicación que se construyó incluye varios módulos de tres categorías distintas. Un módulo corresponde a la reprogramación de las plazas en un semestre. Otro módulo corresponde a la creación y modificación de contratos y la última categoría de módulos corresponde a los mantenimientos. En esta sección se describe el flujo de proceso para cada uno de estos módulos.

6.4.3.1.1. Flujo de procesos de reprogramación de plazas

El flujo de proceso presenta el siguiente algoritmo:

- Se ingresan los datos del semestre que se quiere ingresar; ésta operación se realiza en la página criterioSemestre.jsp.
- Los datos ingresados son enviados dentro de un formulario HTML a el DataAction findSemestre.do desde el cuál se invoca el método buscarSemestre(). Dentro del DataAction findSemestre.do se evalúa si el semestre que se está buscando existe o no, si existe, el flujo de la aplicación se traslada a la página datosSemestre.jsp dando la posibilidad al cliente de que reprograme las plazas del semestre que seleccionó. Si el semestre no existe el flujo retorna a la página criterioSemestre.jsp.
- Si el semestre no existiera, el usuario puede crear uno nuevo invocando el DataAction crearNuevoSemestre.do desde el cuál se invoca el método crearNuevoSemestre(), luego el flujo de de la aplicación se dirige a datosSemestre.jsp para que pueda administrar el semestre que se ha creado.

- Cada vez que se desea llegar a la página datosSemestre.jsp el flujo de la aplicación se dirigirá hacia el DataAction calResPresupuesto.do el cuál invoca al método del negocio calcularPresupuestoRestante(), de tal forma que se calcula el presupuesto restante después de cada modificación que se haga a las plazas.

La página datosSemestre.jsp corresponde el punto de partida de los procesos que complementan la reprogramación de plazas. Desde este punto se puede modificar el presupuesto asignado a un semestre, editar las plazas que corresponden al semestre y agregar o eliminar plazas. Además se pueden barrer las plazas del semestre para cambiar el estado de todas las que se encuentren preasignadas a disponibles o bien cambiar el estado a ocupadas.

El flujo de procesos que corresponde a editar las plazas del semestre es el siguiente:

- Desde la página datosSemestre.jsp se genera el evento EditarAtrPlaza el cual dirige el flujo de la aplicación a la página editarAtribucionesPlaza.jsp. En esta página se pueden generar dos eventos, Commit o Rollback. Si se genera el evento Rollback se cancela la edición de la plaza y el flujo retorna a datosSemestre.jsp. Si se genera el evento Commit se confirma la información que fue modificada y se retorna a datosSemestre.jsp, pasando por calcResPresupuesto.do.

El flujo para eliminar una plaza del semestre corresponde al descrito a continuación:

- Desde la página datosSemestre.jsp se genera el evento EliminarAtrPlaza.
- Al generarse el evento el flujo de la aplicación se dirige a delPlazaSemestre.jsp, página en la cuál el usuario puede generar el evento Delete o Cancelar.
- Si genera el evento Delete el flujo de la aplicación se dirige hacia commitAtriSemestre.do en donde se ejecuta un Commit y luego se invoca calResPresupuesto.do.
- Si se genera el evento Cancelar el flujo de la aplicación se dirige de nuevo a datosSemestre.jsp.

Para agregar una nueva plaza al semestre se ejecuta el siguiente flujo de procesos:

- Desde la página datosSemestre.jsp se genere el evento AgregarPlaza el cual dirige el flujo de la aplicación ha el DataAction getListaPlazaSemestre.do en el cual se ejecuta el método obtenerListaPlazaAgregarSemestre().
- Desde el DataAction getListaPlazaSemestre.do se evalúa el resultado del método, de tal forma que si se obtuvo una lista se dirija el flujo de la aplicación hacia la página listaAsignaPlazaSem.jsp. Si no existen plazas que se puedan agregar, se retorna a la página datosSemestre.jsp.
- En la página listaAsignarPlazaSem.jsp se pueden generar los eventos Cancelar o Asignar.
- Si se genera el evento Cancelar, se vuelve a la página datosSemestre.jsp.
- Si se genera el evento Asignar, se invoca el DataAction agregarPlazaSem.do desde el cuál se invoca el método

agregarPlazaASemestre() por medio del cual se agrega una plaza al semestre que se está trabajando.

El proceso de cambiar el estado de las plazas preasignadas a ocupadas o disponibles consiste en generar desde datosSemestre.jsp el evento correspondiente:

- Si se genera el evento Ocupar se invoca al DataAction ocuparPlazaPreAsignada.do el cual ejecuta el método ocuparPlazaPreAsignadas().

- Si se genera el evento Disponible se invoca al DataAction disponiblePlazaPreAsignada.do el cual ejecuta el método disponblePlazaPreAsignadas().

- Luego el flujo de la aplicación se redirige hacia calcResPresupuesto.do.

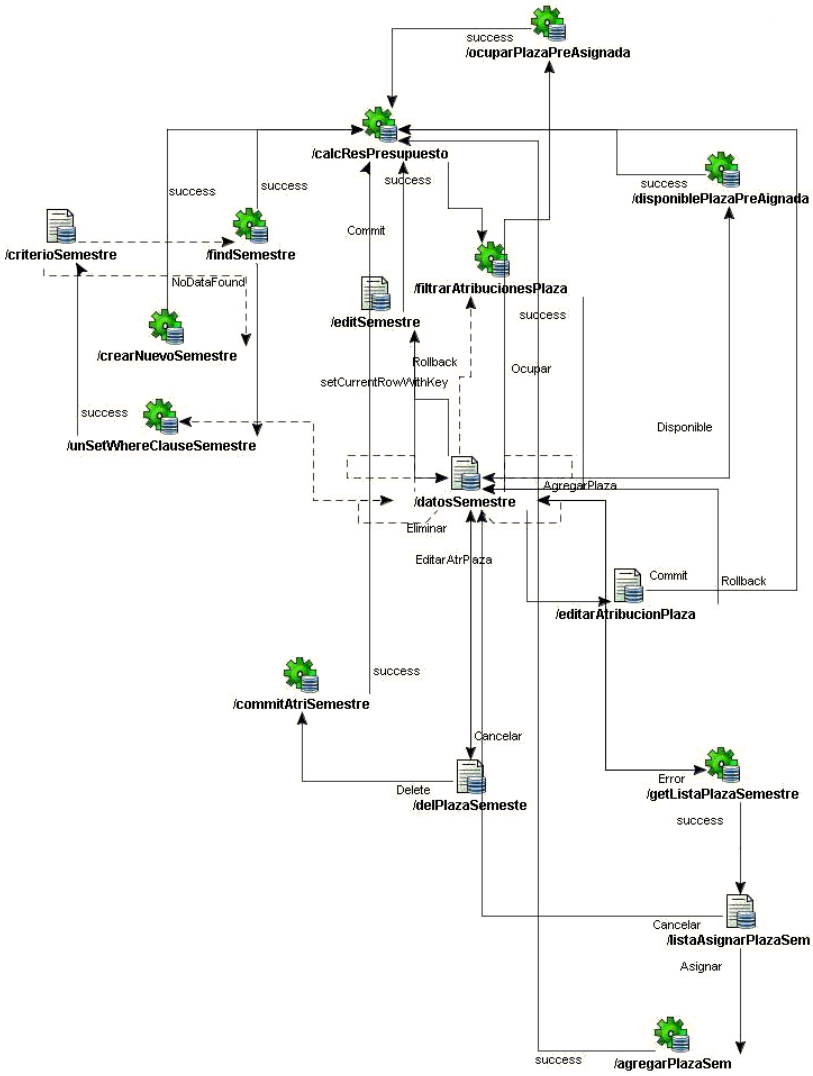
En la figura 64 se puede apreciar de forma gráfica la implementación del controlador para el módulo descrito.

6.4.3.1.2. Flujo de proceso de contratación de personal

Este módulo corresponde al ingreso y edición de contratos. El flujo de proceso es el siguiente:

- La página listaPersonalNacional.jsp presenta una lista de las personas que trabajan para la Escuela de Ciencias y Sistemas. Desde esta página se pueden generar dos eventos, CrearPersonal o VerContrato.

Figura 64. Flujo de procesos para reprogramación de plazas



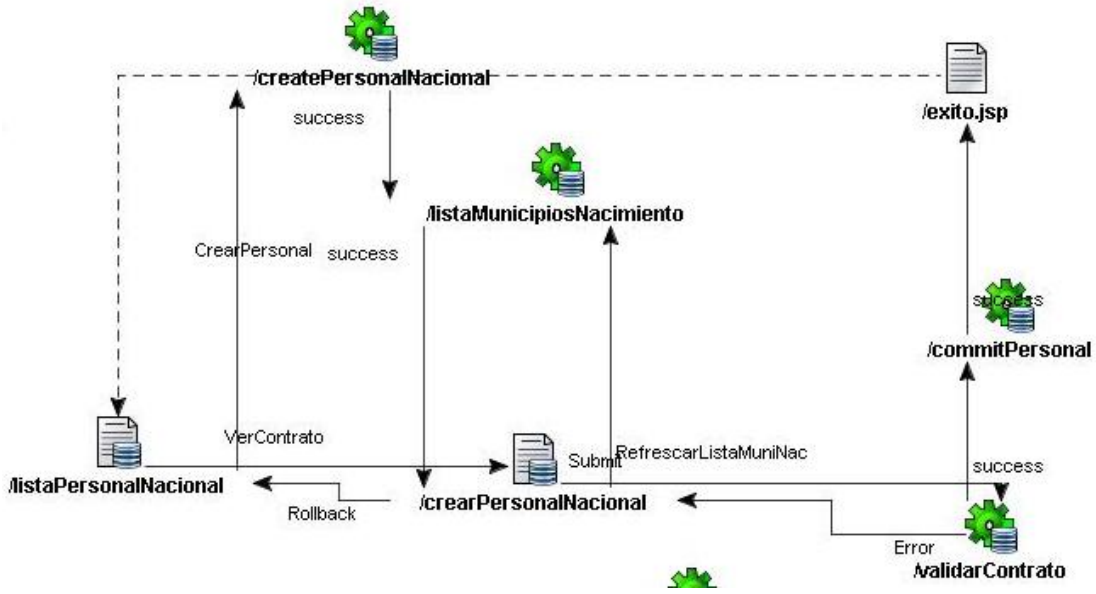
- Si se genera el evento CrearPersonal se invoca el DataAction createPersonalNacional.do el cuál invoca el método Create() del view object PersonalView.
- Luego el flujo de la aplicación se dirige hacia la página crearPersonalNacional.jsp en donde se ingresan los datos del contrato. En esta página se puede generar el evento Summit o Rollback. Si se genera el evento Summit se envía la información del formulario HTML hacia el DataAction validarContrato.do el cual invoca al método validacionesPersonal() en donde se validan los datos. Si uno de los

datos enviado por el formulario es incorrecto, se devuelve el flujo de la aplicación hacia la página crearPersonalNacional.jsp. Si los datos son correctos se invoca el DataAction commitPersonal.do en donde se ejecuta un Commit.

- Si se genera el evento VerContrato desde la página listaPersonalNacional.jsp se invoca a la página crearPersonalNacional.jsp mostrando la información de un empleado en particular.

La figura 65 muestra de forma gráfica el flujo descrito.

Figura 65. Flujo de proceso de contratación de personal



6.4.3.1.3. Flujo de procesos de un mantenimiento

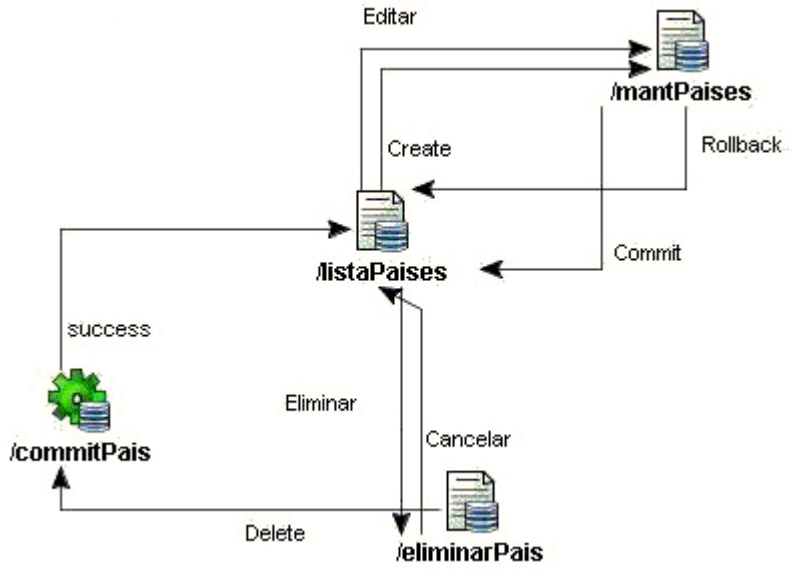
Para los mantenimientos se define el mismo flujo de procesos:

- Se presenta un listado de los registros, desde el cual se puede generar el evento Editar, Eliminar o Create.

- Si se genera Create o Editar se accede a una página con una forma HTML en la cual se pueden ingresar los valores del registro.
- Si se genera Eliminar el flujo de la aplicación se dirige hacia una página de confirmación de eliminación.
- Desde la página de eliminación o edición se generan los eventos Rollback o Commit.
- Si el usuario genera el evento Commit los datos son grabados en la base de datos.
- Si el usuario genera el evento Rollback se cancela la operación.

La figura 66 muestra un ejemplo genérico de este flujo.

Figura 66. Flujo de procesos de un mantenimiento



CONCLUSIONES

1. La especificación J2EE es una plataforma que permite la construcción de aplicaciones robustas basadas en componentes, portables y escalables. Sin embargo, la construcción de aplicaciones utilizando esta especificación es compleja y consume mucho tiempo en el proceso de desarrollo.

2. Existen un gran número de patrones de diseño que intentar facilitar la construcción de aplicaciones utilizando J2EE. MVC, modelo-vista-controlador, es un patrón de diseño popular y fácil de implementar en una solución empresarial de este tipo.

3. Implementar una solución utilizando el patrón de diseño MVC permite construir aplicaciones, en las cuales se obtiene el máximo provecho de la plataforma J2EE facilitando la implementación de la solución así como su administración y, sobre todo, su mantenimiento.

4. El patrón de diseño MVC permite que los desarrolladores se especialicen en las diferentes capas, por ejemplo: los programadores puede concentrarse en la implementación de la lógica del negocio, los diseñadores en la implementación de la interfaz del usuario y el arquitecto o integrador de la aplicación concentra sus esfuerzos en definir el flujo de los procesos.

5. Los frameworks para desarrollo de aplicación facilitan, enormemente, la construcción de aplicaciones al proveer los cimientos y la infraestructura lógica de la aplicación, evitando que el programador tenga que

preocuparse por como implementar procesos repetitivos y complejos como el manejo de persistencia.

6. Oracle ADF, es un framework que facilita la construcción de aplicaciones para el Web, implementando el patrón de diseño MVC utilizando las especificaciones J2EE, de tal forma que las aplicaciones resultan simples y funcionales, además de cumplir con los estándares de la especificación.

7. Oracle ADF extiende la funcionalidad de los componentes comúnmente utilizados para la construcción de aplicaciones para el Web, de manera que una aplicación construida con la ayuda de este framework puede ser implantada en cualquier contenedor J2EE que cumpla con la especificación.

8. Oracle ADF no es un lenguaje de programación nuevo, no es una especificación, ni una plataforma, es un facilitador para la construcción de aplicaciones J2EE.

RECOMENDACIONES

1. Aunque se implemente un patrón de diseño, la construcción de aplicaciones utilizando la especificación J2EE sigue siendo compleja y prolongada. Por tal motivo, debe hacerse uso de un framework para el desarrollo de aplicaciones que facilite este proceso y que permita al desarrollador concentrarse en la implementación de la lógica del negocio y no en construir código repetitivo que defina procesos de mantenimiento o de manejo de persistencia por ejemplo. Oracle ADF facilita este desarrollo de forma eficiente y cumpliendo con los estándares de la especificación, por lo que puede ser utilizado con este propósito.

2. Cuando se construye una aplicación para Internet, se debe tener en cuenta que el rendimiento y funcionalidad de la misma dependen enteramente de su arquitectura. Es necesario que siempre que se construya una aplicación se utiliza un patrón de diseño, tal es el caso de MVC, el cual, como se pudo comprobar en este trabajo final de graduación, presenta ventajas significantes sobre los otros patrones de diseño que existen.

3. La automatización de procesos, como el caso de la administración del presupuesto de la Escuela de Ciencias y Sistemas de la Facultad de Ingeniería, debe realizarse de forma segura, confiable y rápida. La automatización de proceso permite que se agilicen las soluciones y mantener un control bien definido sobre como fue obtenida cada solución. Además, la implementación debe realizarse de forma que la aplicación sea portable, escalable y funcional, por lo cuál debe

implementarse bajo estándares y especificaciones que aporten estas características a la aplicación como es el caso de J2EE.

4. La aplicación para manejo de presupuesto de la Escuela de Sistemas puede se extendida en su funcionalidad agregando módulos especificacos para administradores, que permitan deshacer los cambios realizados por los usuarios finales. Además, podría definirse una interfaz que permitiese imprimir los contratos, interfaz que deberá ser implementada con una tecnología distinta a la descrita en este trabajo de graduación.

5. Los errores de la aplicación que se construyó para la Escuela de Sistemas pueden ser definidos de manera más amigable.

6. La aplicación debe de implementar un modulo para manejar una bitácora que permitira tener control de quien y cuando se han efectuado cambios al presupuesto.

REFERENCIAS ELECTRÓNICAS

1. Para más información sobre J2EE, consultar http://java.sun.com/j2ee/

2. Ver la sección 2.2.1.2.4 y 2.2.1.2.5

3. Para más información sobre la interfaz HttpServletRequest consultar http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/http/HttpServletRequest.html

4. Para más información sobre la interfaz HttpServletResponse y sus métodos consultar http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/http/HttpServletResponse.html

5. Para más información sobre la directiva page consultar http://java.sun.com/products/jsp/tags/11/syntaxref11.fm7.html

6. Para más información sobre JSR 227 consultar http://web1.jcp.org/en/jsr/detail?id=227

7. Para más información sobre Apache Struts consultar http://struts.apache.org/

8. Para más información de las tecnologías estándar utilizadas por Struts consultar http://struts.apache.org/userGuide/preface.html

9. Para más información sobre los paquetes Jakarta Commons consultar http://jakarta.apache.org/commons/index.html

BIBLIOGRAFÍA

1. Patrones de diseño, E. Gamma et al. Editorial Addison-Wesley.

2. Apache Software Foundation: Struts: http://struts.apache.org/. 28 de marzo 2005

3. Wikipedia, encyclopedia electrónica, Jakarta-Struts: http://es.wikipedia.org/wiki/Jakarta_Struts. 28 de marzo 2005

4. Java en Castellano, El API Struts: http://www.programacion.com/java/tutorial/struts/4/. 28 de marzo 2005

5. Designing Enterprise Applications with the J2EETM Platform, Second Edition: http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/web-tier/web-tier5.html. 28 de marzo 2005

6. Sun Java Center - J2EE Patterns: http://java.sun.com/developer/technicalArticles/J2EE/patterns/. 28 de marzo 2005

7. Oracle Technology Network, http://www.oracle.com/technology/. 6 de septiembre 2005

8. Oracle JDeveloper 10g: Build Applicacions with ADF. Oracle University. 2005.

9. Oracle10g: Build J2EE Applications. Oracle University. 2005.

APÉNDICES

APÉNDICE 1. Archivo departamentos.xml

```
<?xml version='1.0' encoding='windows-1252' ?>
<!DOCTYPE Entity SYSTEM "jbo_03_01.dtd">
<Entity
  Name="Departamentos"
  DBObjectType="table"
  DBObjectName="DEPARTAMENTOS"
  AliasName="Departamentos"
  BindingStyle="Oracle"
  UseGlueCode="false"
  RowClass="model.DepartamentosImpl" >
  <Data>
    <Property Name ="IS_ABSTRACT" Value ="FALSE" />
    <Property Name ="COMPLETE_LIBRARY" Value ="FALSE" />
    <Property Name ="IS_LEAF" Value ="FALSE" />
    <Property Name ="IS_ROOT" Value ="FALSE" />
    <Property Name ="ID" Value ="9ad5c2a2-0104-1000-8026-
c00a4632a698::model.Departamentos::EntityObject" />
    <Property Name ="VISIBILITY" Value ="PUBLIC" />
    <Property Name ="IS_ACTIVE" Value ="FALSE" />
  </Data>
  <DesignTime>
    <Attr Name="_isCodegen" Value="true" />
    <Attr Name="_version" Value="9.0.5.16.0" />
    <Attr Name="_codeGenFlag2" Value="Access" />
    <AttrArray Name="_publishEvents">
    </AttrArray>
  </DesignTime>
  <Attribute
    Name="DepartamentoId"
    IsNotNull="true"
    Precision="4"
    Scale="0"
```

```
      ColumnName="DEPARTAMENTO_ID"
      Type="oracle.jbo.domain.Number"
      ColumnType="NUMBER"
      SQLType="NUMERIC"
      TableName="DEPARTAMENTOS"
      PrimaryKey="true" >
      <Data>
        <Property Name ="ID" Value ="9ad5c2a2-0104-1000-8027-
c00a4632a698::model.Departamentos::EntityObjectAttribute" />
        <Property Name ="CHANGEABILITY" Value ="CHANGEABLE" />
        <Property Name ="OWNER_SCOPE" Value ="INSTANCE" />
        <Property Name ="MULTIPLICITY" Value ="1" />
        <Property Name ="VISIBILITY" Value ="PACKAGE" />
      </Data>
      <DesignTime>
        <Attr Name="_DisplaySize" Value="22" />
      </DesignTime>
    </Attribute>
    <Attribute
      Name="Nombre"
      IsNotNull="true"
      Precision="30"
      ColumnName="NOMBRE"
      Type="java.lang.String"
      ColumnType="VARCHAR2"
      SQLType="VARCHAR"
      TableName="DEPARTAMENTOS" >
      <Data>
        <Property Name ="ID" Value ="9ad5c2a2-0104-1000-8028-
c00a4632a698::model.Departamentos::EntityObjectAttribute" />
        <Property Name ="CHANGEABILITY" Value ="CHANGEABLE" />
        <Property Name ="OWNER_SCOPE" Value ="INSTANCE" />
        <Property Name ="MULTIPLICITY" Value ="1" />
        <Property Name ="VISIBILITY" Value ="PACKAGE" />
      </Data>
      <DesignTime>
```

```
        <Attr Name="_DisplaySize" Value="30" />
      </DesignTime>
    </Attribute>
    <Attribute
      Name="Ciudad"
      Precision="30"
      ColumnName="CIUDAD"
      Type="java.lang.String"
      ColumnType="VARCHAR2"
      SQLType="VARCHAR"
      TableName="DEPARTAMENTOS" >
      <Data>
        <Property Name ="ID" Value ="9ad5c2a2-0104-1000-8029-
c00a4632a698::model.Departamentos::EntityObjectAttribute" />
        <Property Name ="CHANGEABILITY" Value ="CHANGEABLE" />
        <Property Name ="OWNER_SCOPE" Value ="INSTANCE" />
        <Property Name ="MULTIPLICITY" Value ="1" />
        <Property Name ="VISIBILITY" Value ="PACKAGE" />
      </Data>
      <DesignTime>
        <Attr Name="_DisplaySize" Value="30" />
      </DesignTime>
    </Attribute>
    <AccessorAttribute
      Name="Empleados"
      Association="model.EmpDeptFkAssoc"
      AssociationEnd="model.EmpDeptFkAssoc.Empleados"
      AssociationOtherEnd="model.EmpDeptFkAssoc.Departamentos"
      Type="oracle.jbo.RowIterator"
      IsUpdateable="false" >
    </AccessorAttribute>
    <Key
      Name="DeptIdPk" >
      <AttrArray Name="Attributes">
        <Item Value="model.Departamentos.DepartamentoId" />
      </AttrArray>
```

```
      <DesignTime>
        <Attr Name="_DBObjectName" Value="DEPT_ID_PK" />
        <Attr Name="_isPrimary" Value="true" />
      </DesignTime>
    </Key>
    <Key
      Name="DeptNameNn" >
      <AttrArray Name="Attributes">
        <Item Value="model.Departamentos.Nombre" />
      </AttrArray>
      <DesignTime>
        <Attr Name="_DBObjectName" Value="DEPT_NAME_NN" />
        <Attr Name="_checkCondition" Value="&amp;#34;NOMBRE&amp;#34; IS NOT NULL" />
        <Attr Name="_isCheck" Value="true" />
      </DesignTime>
    </Key>
  </Entity>
```

APÉNDICE 2. DepartamentosImpl.java

```
package model;
import oracle.jbo.server.EntityImpl;
import oracle.jbo.server.EntityDefImpl;
import oracle.jbo.server.AttributeDefImpl;
import oracle.jbo.domain.Number;
import oracle.jbo.Key;
import oracle.jbo.RowIterator;
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---------------------------------------------------------------------

public class DepartamentosImpl extends EntityImpl
{
  public static final int DEPARTAMENTOID = 0;
  public static final int NOMBRE = 1;
  public static final int CIUDAD = 2;
  public static final int EMPLEADOS = 3;

  private static EntityDefImpl mDefinitionObject;
  /**
   *  This is the default constructor (do not remove)
   */
  public DepartamentosImpl()
  {
  }

  /**
   *  Retrieves the definition object for this instance class.
   */
  public static synchronized EntityDefImpl getDefinitionObject()
  {
    if (mDefinitionObject == null)
    {
      mDefinitionObject = (EntityDefImpl)EntityDefImpl.findDefObject("model.Departamentos");
    }
    return mDefinitionObject;
  }

  /**
   *  Gets the attribute value for DepartamentoId, using the alias name DepartamentoId
   */
  public Number getDepartamentoId()
  {
    return (Number)getAttributeInternal(DEPARTAMENTOID);
  }

  /**
   *  Sets <code>value</code> as the attribute value for DepartamentoId
   */
  public void setDepartamentoId(Number value)
  {
    setAttributeInternal(DEPARTAMENTOID, value);
```

```
}

/**
 *  Gets the attribute value for Nombre, using the alias name Nombre
 */
public String getNombre()
{
  return (String)getAttributeInternal(NOMBRE);
}

/**
 *  Sets <code>value</code> as the attribute value for Nombre
 */
public void setNombre(String value)
{
  setAttributeInternal(NOMBRE, value);
}

/**
 *  Gets the attribute value for Ciudad, using the alias name Ciudad
 */
public String getCiudad()
{
  return (String)getAttributeInternal(CIUDAD);
}

/**
 *  Sets <code>value</code> as the attribute value for Ciudad
 */
public void setCiudad(String value)
{
  setAttributeInternal(CIUDAD, value);
}
//  Generated method. Do not modify.

protected Object getAttrInvokeAccessor(int index, AttributeDefImpl attrDef) throws Exception
{
  switch (index)
    {
    case DEPARTAMENTOID:
      return getDepartamentoId();
    case NOMBRE:
      return getNombre();
    case CIUDAD:
      return getCiudad();
    case EMPLEADOS:
      return getEmpleados();
    default:
      return super.getAttrInvokeAccessor(index, attrDef);
    }
}
//  Generated method. Do not modify.

protected void setAttrInvokeAccessor(int index, Object value, AttributeDefImpl attrDef) throws
Exception
{
  switch (index)
```

```
    {
    case DEPARTAMENTOID:
      setDepartamentoId((Number)value);
      return;
    case NOMBRE:
      setNombre((String)value);
      return;
    case CIUDAD:
      setCiudad((String)value);
      return;
    default:
      super.setAttrInvokeAccessor(index, value, attrDef);
      return;
    }
  }


  /**
   *  Gets the associated entity oracle.jbo.RowIterator
   */
  public RowIterator getEmpleados()
  {
    return (RowIterator)getAttributeInternal(EMPLEADOS);
  }

  /**
   *  Creates a Key object based on given key constituents
   */
  public static Key createPrimaryKey(Number departamentoId)
  {
    return new Key(new Object[] {departamentoId});
  }

}
```

APÉNDICE 3. EmpleadosView.xml

```
<?xml version='1.0' encoding='windows-1252' ?>
<!DOCTYPE ViewObject SYSTEM "jbo_03_01.dtd">
<ViewObject
  Name="EmpleadosView"
  SelectList="Empleados.EMPLEADO_ID,
      Empleados.NOMBRE,
      Empleados.APELLIDO,
      Empleados.EMAIL,
      Empleados.TELEFONO,
      Empleados.FECHA_CONTRATACION,
      Empleados.PUESTO,
      Empleados.SALARIO,
      Empleados.COMISION_PCT,
      Empleados.DEPARTAMENTO_ID"
  FromList="EMPLEADOS Empleados"
  BindingStyle="Oracle"
  CustomQuery="false"
  ComponentClass="model.EmpleadosViewImpl"
  MsgBundleClass="oracle.jbo.common.JboResourceBundle"
  UseGlueCode="false" >
  <DesignTime>
    <Attr Name="_version" Value="9.0.5.16.0" />
    <Attr Name="_codeGenFlag2" Value="Access|Coll" />
    <Attr Name="_isExpertMode" Value="false" />
  </DesignTime>
  <EntityUsage
    Name="Empleados"
    Entity="model.Empleados" >
    <DesignTime>
      <Attr Name="_EntireObjectTable" Value="false" />
      <Attr Name="_queryClause" Value="false" />
    </DesignTime>
  </EntityUsage>
  <ViewAttribute
    Name="EmpleadoId"
    IsNotNull="true"
    PrecisionRule="true"
    EntityAttrName="EmpleadoId"
    EntityUsage="Empleados"
    AliasName="EMPLEADO_ID" >
  </ViewAttribute>
  <ViewAttribute
    Name="Nombre"
    PrecisionRule="true"
    EntityAttrName="Nombre"
    EntityUsage="Empleados"
    AliasName="NOMBRE" >
  </ViewAttribute>
  <ViewAttribute
    Name="Apellido"
    IsNotNull="true"
```

```
    PrecisionRule="true"
    EntityAttrName="Apellido"
    EntityUsage="Empleados"
    AliasName="APELLIDO" >
  </ViewAttribute>
  <ViewAttribute
    Name="Email"
    IsUnique="true"
    IsNotNull="true"
    PrecisionRule="true"
    EntityAttrName="Email"
    EntityUsage="Empleados"
    AliasName="EMAIL" >
  </ViewAttribute>
  <ViewAttribute
    Name="Telefono"
    PrecisionRule="true"
    EntityAttrName="Telefono"
    EntityUsage="Empleados"
    AliasName="TELEFONO" >
  </ViewAttribute>
  <ViewAttribute
    Name="FechaContratacion"
    IsNotNull="true"
    PrecisionRule="true"
    EntityAttrName="FechaContratacion"
    EntityUsage="Empleados"
    AliasName="FECHA_CONTRATACION" >
  </ViewAttribute>
  <ViewAttribute
    Name="Puesto"
    IsNotNull="true"
    PrecisionRule="true"
    EntityAttrName="Puesto"
    EntityUsage="Empleados"
    AliasName="PUESTO" >
  </ViewAttribute>
  <ViewAttribute
    Name="Salario"
    PrecisionRule="true"
    EntityAttrName="Salario"
    EntityUsage="Empleados"
    AliasName="SALARIO" >
  </ViewAttribute>
  <ViewAttribute
    Name="ComisionPct"
    PrecisionRule="true"
    EntityAttrName="ComisionPct"
    EntityUsage="Empleados"
    AliasName="COMISION_PCT" >
  </ViewAttribute>
  <ViewAttribute
    Name="DepartamentoId"
    PrecisionRule="true"
```

```
      EntityAttrName="DepartamentoId"
      EntityUsage="Empleados"
      AliasName="DEPARTAMENTO_ID" >
    </ViewAttribute>
</ViewObject>
```

APÉNDICE 4. EmpleadosViewImpl.java

```
package model;
import oracle.jbo.server.ViewObjectImpl;
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---------------------------------------------------------------------

public class EmpleadosViewImpl extends ViewObjectImpl
{
  /**
   *
   *  This is the default constructor (do not remove)
   */
  public EmpleadosViewImpl()
  {
  }
}
```

APÉNDICE 5. EmpleadosViewImpl.java

```
package model;
import oracle.jbo.server.ViewRowImpl;
import oracle.jbo.server.AttributeDefImpl;
import oracle.jbo.domain.Number;
import oracle.jbo.domain.Date;
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---------------------------------------------------------------------

public class EmpleadosViewRowImpl extends ViewRowImpl
{
  public static final int EMPLEADOID = 0;
  public static final int NOMBRE = 1;
  public static final int APELLIDO = 2;
  public static final int EMAIL = 3;
  public static final int TELEFONO = 4;
  public static final int FECHACONTRATACION = 5;
  public static final int PUESTO = 6;
  public static final int SALARIO = 7;
  public static final int COMISIONPCT = 8;
  public static final int DEPARTAMENTOID = 9;
  /**
   *  This is the default constructor (do not remove)
   */
  public EmpleadosViewRowImpl()
  {
  }

  /**
   *  Gets Empleados entity object.
   */
  public EmpleadosImpl getEmpleados()
  {
    return (EmpleadosImpl)getEntity(0);
  }

  /**
   *  Gets the attribute value for EMPLEADO_ID using the alias name EmpleadoId
   */
  public Number getEmpleadoId()
  {
    return (Number)getAttributeInternal(EMPLEADOID);
  }

  /**
   *  Sets <code>value</code> as attribute value for EMPLEADO_ID using the alias name
EmpleadoId
   */
  public void setEmpleadoId(Number value)
```

```
  {
    setAttributeInternal(EMPLEADOID, value);
  }

  /**
   *  Gets the attribute value for NOMBRE using the alias name Nombre
   */
  public String getNombre()
  {
    return (String)getAttributeInternal(NOMBRE);
  }

  /**
   *  Sets <code>value</code> as attribute value for NOMBRE using the alias name Nombre
   */
  public void setNombre(String value)
  {
    setAttributeInternal(NOMBRE, value);
  }

  /**
   *  Gets the attribute value for APELLIDO using the alias name Apellido
   */
  public String getApellido()
  {
    return (String)getAttributeInternal(APELLIDO);
  }

  /**
   *  Sets <code>value</code> as attribute value for APELLIDO using the alias name
Apellido
   */
  public void setApellido(String value)
  {
    setAttributeInternal(APELLIDO, value);
  }

  /**
   *  Gets the attribute value for EMAIL using the alias name Email
   */
  public String getEmail()
  {
    return (String)getAttributeInternal(EMAIL);
  }

  /**
   *  Sets <code>value</code> as attribute value for EMAIL using the alias name Email
   */
  public void setEmail(String value)
  {
    setAttributeInternal(EMAIL, value);
  }

  /**
```

```
   *  Gets the attribute value for TELEFONO using the alias name Telefono
   */
  public String getTelefono()
  {
    return (String)getAttributeInternal(TELEFONO);
  }

  /**
   *  Sets <code>value</code> as attribute value for TELEFONO using the alias name
Telefono
   */
  public void setTelefono(String value)
  {
    setAttributeInternal(TELEFONO, value);
  }

  /**
   *    Gets the attribute value for FECHA_CONTRATACION using the alias name
FechaContratacion
   */
  public Date getFechaContratacion()
  {
    return (Date)getAttributeInternal(FECHACONTRATACION);
  }

  /**
   *  Sets <code>value</code> as attribute value for FECHA_CONTRATACION using the
alias name FechaContratacion
   */
  public void setFechaContratacion(Date value)
  {
    setAttributeInternal(FECHACONTRATACION, value);
  }

  /**
   *  Gets the attribute value for PUESTO using the alias name Puesto
   */
  public String getPuesto()
  {
    return (String)getAttributeInternal(PUESTO);
  }

  /**
   *  Sets <code>value</code> as attribute value for PUESTO using the alias name Puesto
   */
  public void setPuesto(String value)
  {
    setAttributeInternal(PUESTO, value);
  }

  /**
   *  Gets the attribute value for SALARIO using the alias name Salario
   */
  public Number getSalario()
```

```
  {
    return (Number)getAttributeInternal(SALARIO);
  }

  /**
   *  Sets <code>value</code> as attribute value for SALARIO using the alias name Salario
   */
  public void setSalario(Number value)
  {
    setAttributeInternal(SALARIO, value);
  }

  /**
   *  Gets the attribute value for COMISION_PCT using the alias name ComisionPct
   */
  public Number getComisionPct()
  {
    return (Number)getAttributeInternal(COMISIONPCT);
  }

  /**
   *  Sets <code>value</code> as attribute value for COMISION_PCT using the alias name
ComisionPct
   */
  public void setComisionPct(Number value)
  {
    setAttributeInternal(COMISIONPCT, value);
  }

  /**
   *  Gets the attribute value for DEPARTAMENTO_ID using the alias name DepartamentoId
   */
  public Number getDepartamentoId()
  {
    return (Number)getAttributeInternal(DEPARTAMENTOID);
  }

  /**
   *  Sets <code>value</code> as attribute value for DEPARTAMENTO_ID using the alias
name DepartamentoId
   */
  public void setDepartamentoId(Number value)
  {
    setAttributeInternal(DEPARTAMENTOID, value);
  }
  //  Generated method. Do not modify.

  protected Object getAttrInvokeAccessor(int index, AttributeDefImpl attrDef) throws
Exception
  {
    switch (index)
      {
      case EMPLEADOID:
        return getEmpleadoId();
```

```
      case NOMBRE:
        return getNombre();
      case APELLIDO:
        return getApellido();
      case EMAIL:
        return getEmail();
      case TELEFONO:
        return getTelefono();
      case FECHACONTRATACION:
        return getFechaContratacion();
      case PUESTO:
        return getPuesto();
      case SALARIO:
        return getSalario();
      case COMISIONPCT:
        return getComisionPct();
      case DEPARTAMENTOID:
        return getDepartamentoId();
      default:
        return super.getAttrInvokeAccessor(index, attrDef);
      }
  }
  //  Generated method. Do not modify.

  protected void setAttrInvokeAccessor(int index, Object value, AttributeDefImpl attrDef)
throws Exception
  {
    switch (index)
      {
      case EMPLEADOID:
        setEmpleadoId((Number)value);
        return;
      case NOMBRE:
        setNombre((String)value);
        return;
      case APELLIDO:
        setApellido((String)value);
        return;
      case EMAIL:
        setEmail((String)value);
        return;
      case TELEFONO:
        setTelefono((String)value);
        return;
      case FECHACONTRATACION:
        setFechaContratacion((Date)value);
        return;
      case PUESTO:
        setPuesto((String)value);
        return;
      case SALARIO:
        setSalario((Number)value);
        return;
      case COMISIONPCT:
```

```
        setComisionPct((Number)value);
        return;
      case DEPARTAMENTOID:
        setDepartamentoId((Number)value);
        return;
      default:
        super.setAttrInvokeAccessor(index, value, attrDef);
        return;
      }
  }
}
```

APÉNDICE 6. AppModule.xml

```
<?xml version='1.0' encoding='windows-1252' ?>
<!DOCTYPE AppModule SYSTEM "jbo_03_01.dtd">

<AppModule
  Name="AppModule"
  ComponentClass="model.AppModuleImpl" >
  <DesignTime>
    <Attr Name="_isCodegen" Value="true" />
    <Attr Name="_version" Value="9.0.5.16.0" />
    <Attr Name="_deployType" Value="0" />
    <Attr Name="_ejbAppModule" Value="false" />
  </DesignTime>
  <ViewUsage
    Name="DepartamentosView1"
    ViewObjectName="model.DepartamentosView" >
  </ViewUsage>
  <ViewUsage
    Name="EmpleadosView1"
    ViewObjectName="model.EmpleadosView" >
  </ViewUsage>
  <ViewUsage
    Name="EmpleadosView2"
    ViewObjectName="model.EmpleadosView" >
  </ViewUsage>
  <ViewLinkUsage
    Name="EmpDeptFkLink1"
    ViewLinkObjectName="model.EmpDeptFkLink"
    SrcViewUsageName="model.AppModule.DepartamentosView1"
    DstViewUsageName="model.AppModule.EmpleadosView2" >
    <DesignTime>
      <Attr Name="_isCodegen" Value="true" />
      <Attr Name="_version" Value="9.0.5.16.0" />
    </DesignTime>
  </ViewLinkUsage>
</AppModule>
```

APÉNDICE 7. AppModuleImpl.java

```
package model;
import oracle.jbo.server.ApplicationModuleImpl;
import oracle.jbo.server.ViewLinkImpl;
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---------------------------------------------------------------------

public class AppModuleImpl extends ApplicationModuleImpl
{
  /**
   *  This is the default constructor (do not remove)
   */
  public AppModuleImpl()
  {
  }

  /**
   *  Container's getter for DepartamentosView1
   */
  public DepartamentosViewImpl getDepartamentosView1()
  {
    return (DepartamentosViewImpl)findViewObject("DepartamentosView1");
  }

  /**
   *  Container's getter for EmpleadosView1
   */
  public EmpleadosViewImpl getEmpleadosView1()
  {
    return (EmpleadosViewImpl)findViewObject("EmpleadosView1");
  }

  /**
   *  Container's getter for EmpleadosView2
   */
  public EmpleadosViewImpl getEmpleadosView2()
  {
    return (EmpleadosViewImpl)findViewObject("EmpleadosView2");
  }

  /**
   *  Container's getter for EmpDeptFkLink1
   */
  public ViewLinkImpl getEmpDeptFkLink1()
  {
    return (ViewLinkImpl)findViewLink("EmpDeptFkLink1");
  }

  /**
```

```
   *  Sample main for debugging Business Components code using the tester.
   */
  public static void main(String[] args)
  {
    launchTester("model", "AppModuleLocal");
  }
}
```