UNIVERSIDAD AUSTRAL DE CHILE

FACULTAD DE CIENCIAS DE LA INGENIERÍA

ESCUELA DE INGENIERÍA CIVIL EN INFORMÁTICA

DEFINICIÓN Y APLICACIÓN DE UN PROCESO PARA EL DESARROLLO DE APLICACIONES WEB BASADAS EN LA PLATAFORMA J2EE

Tesis de grado para optar al Título de
Ingeniero Civil en Informática

PROFESOR PATROCINANTE:
ING. MAURICIO RUIZ-TAGLE MOLINA

COPATROCINANTE:
ING. LUIS ALEJANDRO DELANNOY

CRISTIAN RODRIGO PORFLITT BARRIENTOS

VALDIVIA – CHILE
2004

Agradecimientos

A mi mamá Cecilia, infinitas gracias por el amor maternal y protector que siempre he recibido de ti, por la educación y valores entregados, que sin duda han significado constante sacrificio, dedicación y preocupación por brindar a mi y a mis hermanos un mejor bienestar.

A mi papá David, por el apoyo inquebrantable, persistencia y confianza que siempre ha depositado en mi como padre y amigo.

Doy gracias también a Alejandra, una de las personas más importantes en mi vida, por acompañarme, apoyarme y aguantarme desde mis años de estudiante en la UACh hasta días de hoy.

A Alejandro Delannoy, por el apoyo y conocimientos compartidos durante mi primera experiencia laboral y proyecto de tesis, lo cual me permitió conocer las tecnologías sobre la cual desarrollé éste tema de tesis.

Universidad Austral de Chile
Instituto de Informática

Valdivia. 30 de marzo de 2004

Sra.
Miguelina Vega R.
Directora Escuela Ingeniería Civil en Informática

De mi consideración:

Mediante la presente, hago llegar a Ud. mi evaluación como profesor patrocinante, del trabajo de Tesis de Grado de Ingeniero Civil en Informática del Sr. Cristian Rodrigo Porflitt Barrientos, titulado "DEFINICIÓN Y APLICACIÓN DE UN PROCESO PARA EL DESARROLLO DE APLICACIONES WEB BASADAS EN LA PLATAFORMA J2EE".

Estimo que el trabajo de titulación del Sr. Cristian Porflitt cumple los objetivos propuestos al plantear de manera sistemática y rigurosa una metodología específica de desarrollo de aplicaciones sobre WEB bajo una plataforma de creciente importancia como J2EE.

La relevancia actual del desarrollo de software en plataformas de estas características, hace que este trabajo se transforme en un aporte no sólo pertinente, sino además de gran utilidad para quienes enfrentan este tipo de desafíos.

No obstante, se debe corregir algunos aspectos formales puntuales en la presentación del trabajo, para no mermar la calidad del análisis y metodología propuestos.

Por todo lo anteriormente expuesto, califico el trabajo de titulación del Sr. Cristian Porflitt Barrientos, con nota 6.6 (seis coma seis).

Sin otro particular, se despide atte.

Mauricio Ruiz-Tagle Molina
Instituto de Informática

General Lagos 2086 · Campus Miraflores · Valdivia · Chile
Casilla 567 · Fono: 56 63 221427 · Fax: 56 63 293115 · instituto@inf.uach.cl · www.inf. uach.cl

Valdivia, marzo 29 del 2004

DE: LUIS ALEJANDRO DELANNOY IVERSEN

A: DIRECTORA DE ESCUELA INGENIERIA CIVIL EN INFORMATICA

MOTIVO:

INFORME TRABAJO DE TITULACION

Nombre Trabajo de Titulación: DEFINICION Y APLICACION DE UN PROCESO PARA EL DESARROLLO DE APLICACIONES WEB BASADAS EN LA PLATAFORMA J2EE

Nombre del Alumno: CRISTIAN RODRIGO PORFLITT BARRIENTOS

Nota: 6,5 Seis coma cinco

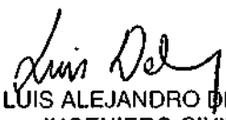
(en números) (en letras)

Fundamentos de la nota:

Se alcanza la totalidad de los objetivos propuestos, proporcionando una metodología aplicable a proyectos de software en ambientes web de cualquier tamaño. La rigurosidad de la investigación sobre la infraestructura J2EE, permite formase una idea clara de la plataforma y sus aplicaciones.

El trabajo muestra una metodología formal para desarrollar aplicaciones web, acercando el proceso al cliente. Con esto se acortan los tiempos de desarrollo y se llega a soluciones que cumplen los requerimientos.

Falta una propuesta para organizar y almacenar los documentos de análisis y diseño de tal forma de controlar los cambios en los proyectos.



LUIS ALEJANDRO DELANNOY IVERSEN
INGENIERO CIVIL EN INFORMÁTICA

Universidad Austral de Chile
Instituto de Informática

Valdivia, 30 de marzo de 2004.

De : Luis Hernán Vidal Vidal.
A : Sra. Miguelina Vega R.
Directora de Escuela de Ingeniería Civil en Informática.
Ref. : Informa Calificación Trabajo de Titulación.

MOTIVO: Informar revisión y calificación del Proyecto de Título "Definición y Aplicación de un Proceso para el Desarrollo de Aplicaciones Web basadas en la Plataforma J2EE", presentado por el alumno Cristian Rodrigo Porflitt Barrientos, que refleja lo siguiente:

Se logró el objetivo general planteado de definir un proceso formal para el desarrollo de aplicaciones Web construidas sobre plataforma tecnológica J2EE, aún cuando el objetivo específico de aplicar el proceso propuesto en un ejemplo práctico se incluyó, el trabajo de tesis carece de un análisis que se refleje los pro y contra del uso del proceso formal propuesto en la tesis.

La revisión hecha sobre los estándares y tecnologías empleadas entregan una base referencial clara y de gran valor al momento de abordar proyectos Web sobre la plataforma J2EE.

El trabajo de tesis presentó una muy buena solución ante la gran diversidad de herramientas de desarrollo disponibles, tanto de modelamiento como de implementación.

Por todo lo anterior expuesto califico el trabajo de titulación del Sr. Cristian Rodrigo Porflitt Barrientos con nota 6.4 (seis como cuatro).

Sin otro particular, se despide atentamente.

UNIVERSIDAD AUSTRAL DE CHILE
INSTITUTO DE INFORMATICA
FACULTAD DE CIENCIAS DE LA INGENIERIA

Ing. Luis Hernán Vidal Vidal
Profesor Instituto de Informática.
Facultad de Ciencias de la Ingeniería.
Universidad Austral de Chile.

Índice de Contenidos

CAPÍTULO III: TECNOLOGÍAS JAVA PARA J2EE

Índice de figuras

Síntesis

El siguiente proyecto de tesis propone un proceso formal para el desarrollo de aplicaciones Web empresariales, implementadas sobre la plataforma tecnológica Java 2 Enterprise Edition. El proceso, de carácter iterativo e incremental, permite construir sistemas complejos, dado que cada iteración se enfoca a resolver un requerimiento específico de la aplicación. Cada iteración se traduce finalmente en un incremento en el producto final. Basado en técnicas de Análisis y Diseño Orientado a Objetos (ADOO), el proceso permite modelar el conjunto de clases que encapsularán la lógica de negocio del dominio de la aplicación, haciendo partícipe de éstas actividades al propio cliente y usuario. Posteriormente la lógica de negocio es utilizada junto con el patrón arquitectónico Modelo-Vista-Controlador (MVC) para construir una solución en base a componentes tecnológicos de la plataforma J2EE.

Abstract

Following thesis project proposes a formal process for the development of Enterprise Web Applications, implemented on Java 2 Platform Enterprise Edition. This process has iterative and incremental character, allows constructing complex systems, each iteration focuses to solve a specific requirement inside the application. Each iteration is translated finally in an increase on Web Application. Based on techniques of Object Oriented Analysis and Design (OOAD), the process allows modelling the set of classes that will encapsulate the business logic of application domain, doing it participates in these activities to the own client and user. Later the businesses logic is used along with the Model-View-Controller (MVC) architectonic pattern to construct a solution on the basis of technological components of J2EE platform.

Capítulo I

Introducción

1. Presentación

La continua y acelerada evolución que ha experimentado Internet durante los últimos años, ha dado lugar a la proliferación de nuevas tecnologías para el desarrollo de aplicaciones cada vez más complejas que utilizan esta red como medio de comunicación. Lo anterior ha generado la necesidad de contar con procesos formales que permitan analizar, diseñar e implementar en forma seria este tipo de aplicaciones.

Para ilustrar lo anterior, podemos mencionar cómo las instituciones bancarias de nuestro país han ido adaptando sus portales Web y poniendo al alcance de sus clientes funcionalidades que permiten realizar delicadas y complejas transacciones en línea. Estas aplicaciones conforman características de liderazgo y diferenciación para este tipo de organizaciones, actuando como herramientas de retención y satisfacción de sus clientes, al mejorar en muchos aspectos la calidad del servicio, "…No haga más la cola, haga clic".

Cualquier sistema de complejidad no trivial, necesita ser analizado y modelado. Las aplicaciones Web, al igual que otras aplicaciones, necesitan de métodos formales de análisis y diseño para su adecuada construcción.

Un proceso de desarrollo de software define quién esta haciendo qué, cuándo y cómo alcanzar un determinado objetivo. En Ingeniería de software el objetivo principal es construir un producto software de calidad o mejorar uno existente [Jac+, 00]. Un proceso efectivo debe proporcionar normas para el desarrollo eficiente y debe hacer participe del desarrollo a todos los involucrados en el proyecto (usuarios, diseñadores, programadores, etc.) mediante un lenguaje común.

El presente proyecto de tesis esta enfocado a adquirir conocimiento teórico y práctico de la tecnología Java 2 Enterprise Edition, con el fin de proponer una metodología para el desarrollo de aplicaciones Web.

2. Antecedentes

El Proceso Unificado de Desarrollo de Software (RUP, Rational Unified Process) es quizás uno de los procesos más aceptados y adoptados para el desarrollo de aplicaciones que se construye sobre la plataforma J2EE. Catalogado por la comunidad informática como un proceso pesado, RUP tiene el inconveniente de ser un proceso muy complejo de implantar en organizaciones pequeñas y medianas de desarrollo de software, como lo son las mayorías de las empresas de nuestro país. Uno de los mayores inconvenientes de RUP es que todas las actividades relacionadas a éste son apoyadas mediante herramientas de alto costo, inaccesibles a la mayoría de las organizaciones. Las herramientas y los conocimientos necesarios para su correcto uso, son comercializados por Rational e IBM, empresas que han formado una alianza estratégica para colaborar en iniciativas puramente comerciales.

3. Motivación

Los factores principales que motivaron el desarrollo de este tema de tesis son:

- Adquirir conocimiento teórico y práctico sobre el funcionamiento de la plataforma J2EE y otras tecnologías Java que la complementan.
- Necesidad de contar con un proceso formal y claro para el desarrollo de aplicaciones Web basadas en J2EE.

4. Objetivos

4.1 Objetivo General

El siguiente proyecto de tesis tiene como objetivo definir un proceso formal para el desarrollo de aplicaciones Web construidas sobre la plataforma tecnológica Java 2 Enterprise Edition.

4.2 Objetivos Específicos

1. Investigar la plataforma J2EE. El resultado verificable de este objetivo será un documento con nociones básicas sobre el funcionamiento y principales características de la plataforma. Como base teórica se utilizará la versión 3 de la especificación de la plataforma J2EE proporcionada por Sun Microsystem.
2. Investigación de las principales tecnologías Java que se ejecutan sobre la plataforma J2EE. Como resultado se generará un resumen que describa el funcionamiento y características más significativas de tecnologías Java tales como: Java Server Pages, Enterprise JavaBeans, Java Servlet y APIs disponibles para procesamiento de XML.
3. Definir un proceso de desarrollo para aplicaciones Web basadas en tecnología J2EE. El resultado verificable es un documento que contendrá las actividades necesarias que se deben realizar para analizar, diseñar e implementar aplicaciones Web basadas en tecnología J2EE.
4. Aplicar en un ejemplo práctico, el proceso propuesto en el objetivo específico 3 de esta tesis. Con el fin de ejemplificar el uso de la metodología de desarrollo propuesta, ésta será aplicada en un proyecto real de desarrollo de una aplicación Web. Como resultado verificable se entregará la documentación de todas las actividades presentes en el proceso de desarrollo.

Capítulo II

La Plataforma J2EE

1. Introducción

En este capítulo se describe brevemente los inicios del lenguaje de programación Java y las principales características por la cual fue adoptado en la creación de una de las más importantes especificaciones de estandarización de servidores de aplicaciones empresariales que existe hoy en día. Posteriormente se detallará su arquitectura, los servicios que presta como plataforma, ambiente para la construcción y ejecución de aplicaciones empresariales, tomando como base teórica la versión 1.3 de la especificación de la plataforma J2EE, proporcionada por Sun Microsystems y documentada por Hill Shannon en Julio del 2001.

2. El lenguaje de programación Java

2.1 Inicios de Java

Java, inicialmente llamado Oak, fue concebido como un lenguaje para construir pequeños programas introducidos en dispositivos eléctricos, PDA (Personal Digital Assistants) y un poco más adelante se utilizó para ejecutar aplicaciones para televisores. Aunque ninguno de estos productos tuvo éxito comercial, posteriormente Java fue utilizado para programar Applets, pequeñas aplicaciones que se ejecutan en los navegadores Web que enriquecen y complementan los limitados componentes de la interfaz Web programada en HTML.

2.2 Principales características de Java

Las principales características de Java son:

- Simple y poderoso. Es fácil de aprender y tiene la posibilidad de expresar cualquier sistema que sea concebido bajo el paradigma orientado a objetos.

- Seguro. Uno de los principios claves de Java es la seguridad, no posee características inseguras que hayan tenido que corregirse. La diferencia más importante respecto a C/C++ es que Java no maneja punteros. C y C++ permiten la declaración y uso de punteros, que pueden ser utilizados en cualquier lugar. Esta tremenda flexibilidad resulta muy útil, pero también es la causa de que podamos colgar todo el sistema. La intención principal en el uso de los punteros es comunicarnos más directamente con el hardware, haciendo que el código se acelere. Desafortunadamente, este modelo de tan bajo nivel hace que perdamos robustez y seguridad en la programación y hace muy difíciles tareas como la liberación automática de memoria, la defragmentación de memoria, o realizar programación distribuida de forma clara y eficiente [URL1]

- Orientación a Objetos. Desde sus inicios Java fue construido con el enfoque de lenguaje de programación orientado a objetos. Aunque el paradigma de orientación objetos parece difícil de comprender y es visto como un obstáculo para quienes se inician en la programación de este lenguaje, este presenta importantes beneficios para la construcción grandes sistemas.

- Robusto. Java disminuye el peso de la responsabilidad que tienen los programadores en preocuparse de algunos aspectos tales como administración de memoria y manejo de excepciones, las cuales son manejadas internamente por la Maquina Virtual Java.

- Múltiples Hilos. Java permite ejecutar múltiples procesos ligeros en forma concurrente, dando mayor dinamismo e interactividad a las aplicaciones.

- Arquitectura neutral. Java cuenta con la conocida característica “Escriba una vez y utilícelo donde quiera”, lo que permite asegurar que un programa escrito en Java funcionará en cualquier plataforma de hardware y sistema operativo que cuente con la Maquina Virtual Java. Lo anterior nos asegura la portabilidad de nuestras aplicaciones.

- Interpretado y de alto rendimiento. Gracias a la representación intermedia llamada “Código de byte de Java”, este se puede ejecutar en cualquier sistema que cuente con un interprete Java. Ha quedado demostrado que otros lenguajes de programación que trataron de ser independientes de la plataforma, como por ejemplo Basic, Tcl y Perl, sufrieron problemas de rendimiento debido a que eran interpretados. Sin embargo, Java fue diseñado para tener un buen rendimiento en procesadores de poca potencia. Gracias al cuidado que se tuvo en su diseño, se logró conseguir rendimientos altos debido a la sencillez con que se traduce el código de byte a código de máquina nativo.

Estas y otras características han hecho que Java haya aumentado progresivamente su popularidad y aceptación en la comunidad informática. Java ha comenzado a ser utilizado para construir importantes sistemas empresariales, sin embargo, el lenguaje por si sólo no basta para satisfacer los requerimientos de grandes sistemas de información. Se requiere una plataforma que provea servicios y aspectos básicos que apoyen el buen funcionamiento de una aplicación empresarial. Entre estos servicios se pueden citar:

- Capacidad de almacenar información en bases de datos.

- Capacidad de distribuir una aplicación en varios servidores.
- Manejo de transacciones, para asegurar la integridad de la información.
- Procesos multi-hilos, para permitir accesos concurrentes a los servicios.
- Pool de conexiones, para asegurar el buen desempeño de nuestra base de datos.
- Escalabilidad, con el fin de permitir el crecimiento de la aplicación en la medida que aparezcan nuevos requerimientos. Esta característica esta siempre presente en las aplicaciones empresariales debido a que ellas cambian continuamente sus procesos de negocio.

2.3 Variantes de Java

Existen tres variantes principales de Java, estas deben ser utilizadas según las características y ambientes en que se ejecutará la aplicación desarrollada.

- J2SE (Java 2 Standard Edition), Edición Estándar de Java 2, es la base de la tecnología Java.
- J2EE (Java 2 Enterprise Edition), extiende las funcionalidades de J2SE mediante bibliotecas y servicios especiales para la construcción de aplicaciones de carácter empresarial.
- J2ME (Java 2 Micro Edition), es una plataforma para desarrollar y ejecutar aplicaciones Java en pequeños dispositivos tales como PDA o teléfonos móviles.

3. Arquitectura de Aplicaciones

3.1 Introducción

La noción de arquitectura tratada en esta sección hace referencia a la forma de estructurar los elementos que componen el software. La estrategia elegida para estructurar estos elementos impactará directamente en factores cómo:

- Escalabilidad de la aplicación.
- Facilidad de compresión del sistema y mantención.
- Reutilización de componentes.
- Tráfico generado en redes de comunicación.
- Grado de dependencia entre capas.
- Esfuerzo invertido para actualizar el sistema o proveer de nuevas versiones a los usuarios.

A continuación se exponen los dos modelos más utilizados en la arquitectura de aplicaciones.

3.2 Arquitectura de dos capas

La arquitectura clásica de dos capas o *Cliente-Servidor* fue muy utilizada en los años 80' y 90'. Tal como muestra la figura 1, en esta arquitectura la lógica de presentación (código que produce el despliegue y captura de información en forma gráfica) es presentada en el lado del cliente de la aplicación junto con la lógica de negocio.

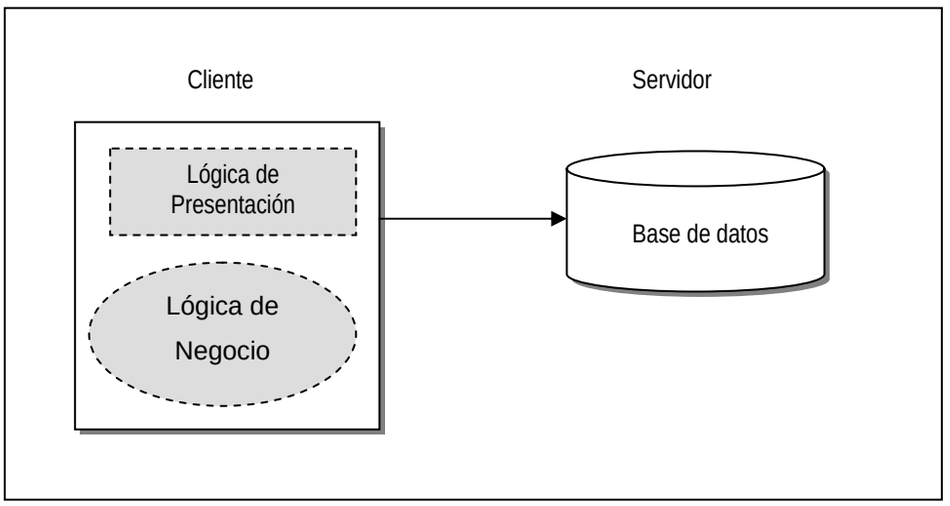


Fig. 1 Modelo de dos capas

Los datos son almacenados, por lo general, en una base de datos en donde la aplicación cliente se comunica utilizando secuencias SQL. En algunas ocasiones, parte de la lógica de negocios suele implementarse en el servidor de base de datos, en forma de procedimientos almacenados o procesos demonios que se ejecutan realizando tareas repetitivas.

La desventaja de esta arquitectura es que la lógica de presentación y de negocios están juntas en la misma capa, siendo difícil de entender y mantener.

3.3 Arquitectura de tres capas

La característica principal del modelo de 3 capas, representado en la figura 2, es que separa la lógica de presentación de la de negocio. Las dos capas son desarrolladas tan independientemente como sea posible, de esta forma la capa de presentación sólo envía peticiones sin saber cómo la capa de negocio procesa los datos y entrega una respuesta, la capa de negocio actúa como una especie de caja negra. De la misma forma, a la capa de negocios no le interesa saber en que forma serán mostrados los datos.

La independencia entre capa de presentación y la de negocio nos permite administrar y mantener ambas capas por separado. También se puede tener distintas tecnologías en la capa de presentación para una misma lógica de negocios, logrando así el despliegue de la aplicación en distintos dispositivos, por ejemplo: computadores personales, PDAs o asistentes personales, teléfonos móviles, etc. Una técnica para implementar esta arquitectura es la aplicación del patrón arquitectónico Modelo-Vista-Controlador, el cual será abordado en los capitulo posteriores.

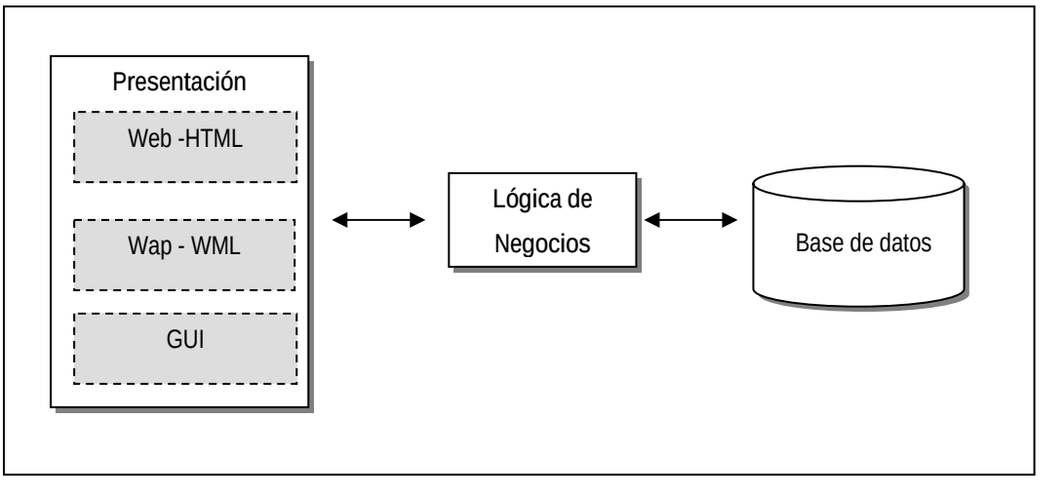


Fig. 2 Modelo de tres Capas

4. Servidores de Aplicaciones

Los servidores de aplicaciones proporcionan las condiciones ambientales necesarias para desarrollar aplicaciones multi-capas, y proporcionando servicios tales como distribución, multi-hilos, seguridad y persistencia.

En el pasado, las empresas que construían estos servidores de aplicaciones, no incorporaban todos los servicios necesarios para desarrollar una aplicación integra, por lo cual los desarrolladores debían usar varios productos para implementar finalmente todos los requerimientos de la aplicación. Es más, estos servicios eran implementados de manera diferente sin ningún estándar que los rija.

En 1997, un grupo de organizaciones, entre ellas Oracle, IBM, Bea Systems y Sun, comenzaron a trabajar juntos para definir un estándar para servidores de aplicaciones basado en el lenguaje Java. La visión fue crear un conjunto de servicios estandarizados y APIs (Interfaz de Programación de Aplicaciones) estándar para acceder a estos servicios.

4.1 El estándar J2EE

El estándar J2EE define rigurosamente un conjunto de servicios que el servidor de aplicaciones debe poseer, además de una colección de APIs para acceder a estos servicios. El estándar esta ligado al lenguaje de programación Java.

Es importante notar que J2EE no es un producto, sino más bien una especificación para la cual existen muchas implementaciones, comerciales y no comerciales. Las implementaciones deben ser construidas conforme a la especificación para ser consideradas compatibles con J2EE.

Sun Microsystem proporciona un producto de implementación de referencia conforme al estándar J2EE, este se puede utilizar libremente y es ideal para aprender el funcionamiento de la plataforma. Esta implementación no hace nada más que lo que la especificación requiere, en comparación con implementaciones comerciales que poseen mayores prestaciones, son más robustas para aplicaciones reales, poseen interfaces gráficas de administración y asistentes para el desarrollo de aplicaciones.

5. Introducción a l**a plataforma J2EE**

5.1 Arquitectura

La figura 3 muestra la arquitectura de J2EE mediante relaciones lógicas entre los elementos.

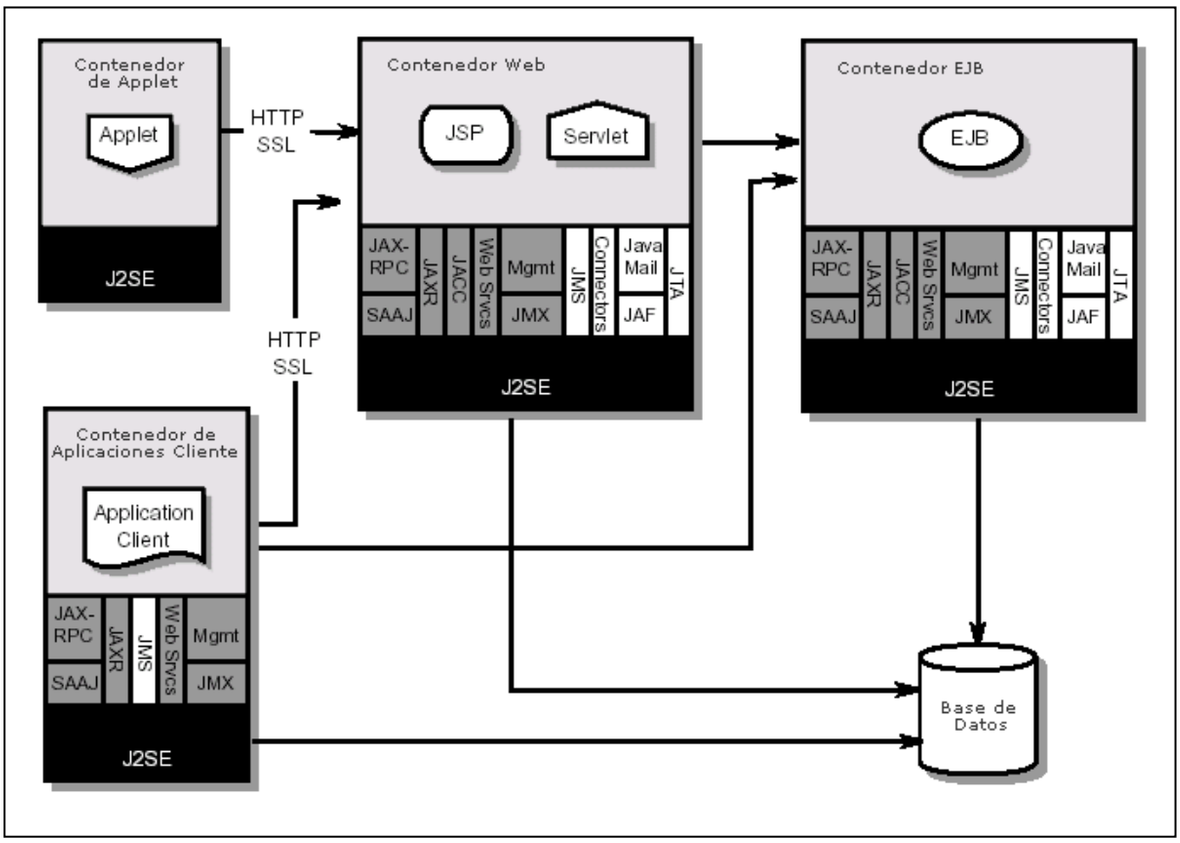


Fig.3 Diagrama de Arquitectura J2EE

Los contenedores, denotados por rectángulos, son ambientes de ejecución J2EE que proporcionan los servicios necesarios a los componentes de aplicaciones, representados en la parte superior de los rectángulos. Por ejemplo, el contenedor Web, proporciona el servicio Java Messaging Service (JMS) para las páginas Java Server Pages (JSP) y Servlet.

Como se aprecia en la figura anterior, cada tipo de componentes de aplicaciones es ejecutado en un tipo específico de Contenedor.

5.2 Componentes de Aplicación

Un componente de aplicación es una unidad de software autosuficiente, que se ensambla en una aplicación J2EE con sus clases y ficheros relacionados y que posee la capacidad de comunicarse con otros componentes. Una aplicación J2EE puede estar compuesta por uno o más componentes.

Los componentes son unidades independientes que interactúan entre si, logrando formar un sistema. Además poseen interfaces para permitir a los usuarios, y otros componentes, acceder a los servicios que proporcionan.

Existen fundamentalmente 4 tipos de componentes que un producto J2EE debe incluir:

1. Aplicaciones Clientes. Son programas escritos en lenguaje Java que se ejecutan sobre el cliente como interfaz de usuario. Tienen acceso a todas las facilidades de la capa intermedia de J2EE y también a la capa de Sistemas de Información empresarial, como por ejemplo bases de datos.

2. Applets. Son pequeños programas que pueden ser desplegados en un navegador Web a través de una página Web. Pueden también ser ejecutados en otros dispositivos que soporten el modelo de programación Applet. Los Applet enriquecen las limitadas prestaciones de la interfaz gráfica del HTML, por lo cual son principalmente usados como parte de la interfaz de usuario. Los applets cuentan con su propia especificación.

3. Servlets, páginas JSP, filtros y detectores de eventos Web. Se ejecutan en servidores Web y pueden responder a solicitudes HTTP desde clientes Web. Estos componentes son usados para generar páginas HTML en forma dinámica que posteriormente son interpretadas por un navegador Web. Estas páginas Web constituyen la interfaz de usuario de una aplicación. También se pueden usar estos componentes para generar otros tipos de salidas, como por ejemplo XML u otros formatos que son consumidos por otros componentes de aplicaciones. Los Servlets, páginas JSP, filtros y detectores de eventos Web son llamados también *"Componentes Web*".

4. Enterprise JavaBeans. Son componentes que encapsulan la lógica de negocios de una aplicación y se ejecutan en ambientes administrados con soporte de transacciones. Los Enterprise JavaBean cuentan con su propia especificación.

En los próximos capítulos se tratará en más detalles cada uno de estos componentes.

5.3 Contenedores

Los contenedores proporcionan un ambiente de ejecución a los componentes de aplicaciones y forman parte de la implementación comercial de un servidor J2EE. Los componentes de aplicaciones utilizan los protocolos y métodos de comunicación de los contenedores para comunicarse entre si o para interactuar con otros servicios de la plataforma.

Un producto típico de J2EE, proporciona un contenedor para cada tipo de componente, es decir, debieran existir los siguientes tipos de contenedores:

- Contenedor de Aplicaciones Clientes
- Contenedor de Applet

- Contenedor de Componentes Web
- Contenedor de Enterprise Bean

Cada contenedor debe:

1. Proporcionar un ambiente de ejecución compatible con Java.
2. Ser implementado por un proveedor de productos J2EE.
3. Incluir un conjunto de servicios estándar.
4. Disponer de APIs para que los componentes de aplicaciones hagan uso de estos servicios.
5. Proveer formas de extender los servicios J2EE hacia otros sistemas de aplicaciones no J2EE por medio de “conectores”.

La especificación no describe cómo los contenedores, servicios y funciones deben ser divididos entre diferentes procesos o instalados distribuidamente en distintos servidores físicos. Una implementación de J2EE muy simplista podría estar compuesta de una sola máquina virtual Java que permita la ejecución de todos los componentes en un sólo contenedor. Por otro lado podríamos tener una implementación que divida el servidor de componentes en múltiples servidores, cada uno distribuido en diferentes máquinas con el objetivo de realizar un balance de carga.

Una aplicación será portátil cuando pueda ser instalada, y se ejecute, en cualquiera de estas configuraciones.

5.4 Drivers Administradores de Recursos

Un driver administrador de recursos, es un componente de software que trabaja a nivel de sistema para implementar una conectividad de red hacia un administrador de recursos externo, extendiendo de esta forma las funcionalidades de la plataforma J2EE. Las nuevas funcionalidades deben ser accedidas a través de APIs, un ejemplo de driver es JDBC, ya que permite conectar J2EE con un servidor de base de datos. Si un driver es implementado usando J2EE SPI (Interfaz de Proveedor de Servicios J2EE), este podrá funcionar en cualquier producto J2EE, de otra forma estará limitado a funcionar sólo con la implementación de plataforma J2EE para la cual fue construido.

Por lo general las aplicaciones J2EE necesitan conectividad con bases de datos para almacenar y extraer información. Esta conectividad es implementada a través de la API JDBC. De esta forma la base de datos es accesible desde componentes Web, enterprise beans y componentes de aplicaciones de negocio.

5.5 Servicios Estándar J2EE

La plataforma J2EE incluye los siguientes servicios estándares:

- HTTP.
- HTTPS.
- Java Transaction API (JTA)
- RMI-IIOP
- Java IDL.
- JDBC API.
- Java Message Service (JMS).
- Java Naming and Directory Interface™ (JNDI)
- JavaMail™
- JavaBeans™ Activation Framework (JAF)
- Java™ API for XML Parsing (JAXP)
- J2EE™ Connector Architecture
- Security Services
- Web Servicies
- Management
- Deployment

5.6 Extensiones del Producto J2EE

Aunque la especificación describe un conjunto mínimo de facilidades que deben incluir todos los productos J2EE, es posible que cada proveedor pueda incluir sus propias prestaciones más allá del conjunto mínimo requerido, ampliando de esta forma las funcionalidades disponibles, por ejemplo: compatibilidad con otros protocolos de comunicación, nuevos servicios, apoyar la ejecución de programas escritos en otros lenguajes de programación, conectividad con otros sistemas de información, etc.

Al igual que en la plataforma J2SE, se debe tener en cuenta que un producto J2EE no puede agregar clases al paquete del lenguaje de programación Java incluida en esta especificación, y no puede agregar métodos o alterar las clases especificadas.

Se debe tener presente que una aplicación que haga uso de extensiones no requeridas en la especificación será menos portátil. Dependiendo de las facilidades usadas, la perdida de portabilidad puede ser menor o significante.

Con esta flexibilidad, se espera que los proveedores incorporen nuevas facilidades a sus productos J2EE, los cuales tal vez puedan ser considerados en futuras versiones como parte de la especificación oficial.

5.7 Roles en la Plataforma

Esta sección describe los roles que hacen posible el desarrollo de aplicaciones J2EE.

5.7.1 Proveedor de Producto J2EE

Un proveedor de productos J2EE es quién implementa y suministra los productos J2EE, es decir, el contenedor de componentes, APIs de la plataforma J2EE, y otros requerimientos definidos en la especificación. Por lo general un proveedor de productos J2EE, también provee sistemas operativos, servidores Web, bases de datos, etc.

5.7.2 Proveedor de Componentes de Aplicación

Existen múltiples roles para Proveedores de Componentes de Aplicaciones, estos incluyen: diseñadores de documentos HTML, programadores de documentos y desarrolladores de EJB. Estos roles hacen uso de herramientas para producir aplicaciones y componentes J2EE.

5.7.3 Ensamblador de Aplicaciones

El Ensamblador de Aplicaciones toma un conjunto de componentes desarrollados por el Proveedor de Componentes de Aplicaciones y los ensambla en una aplicación J2EE, entregándola en formato de archivo Enterprise Archive (.ear). El Ensamblador de Aplicaciones generalmente hace uso de una herramienta en forma de interfaz gráfica de usuario proporcionada por el Proveedor de Plataforma o Proveedor de Herramientas.

El Ensamblador de Aplicaciones es responsable de entregar las instrucciones de ensamblaje describiendo dependencias externas de la aplicación para ser consideradas en la instalación.

5.7.4 Instalador

El instalador (deployer) toma los componentes provistos por el ensamblador y los despliega, asegurando el cumpliendo de todas las dependencias

El despliegue es típicamente un proceso de tres fases:

1. Instalación. Durante esta fase el desplegador traslada los medios de la aplicación al servidor, genera las clases adicionales específicas para el contenedor e interfaces que posibilitan al contenedor administrar los componentes de aplicación en tiempo de ejecución, e instala los componentes de aplicación, clases e interfaces en un contenedor J2EE apropiado.
2. Configuración. Durante esta fase son resueltas las dependencias externas declaradas por el Proveedor de Componentes de Aplicaciones y se siguen las instrucciones de ensamblaje de la aplicación proporcionadas por el Ensamblador de Aplicaciones.
3. Ejecución. Puesta en marcha de la aplicación previamente instalada y configurada.

En algunos casos un Instalador experimentado puede incluso personalizar la lógica de negocio de los componentes de aplicación o cambiar la apariencia de páginas JSP.

La función principal del Instalador es entregar como resultado de su labor, aplicaciones Web, EJB, Applet y aplicaciones clientes funcionando en un ambiente operacional objetivo.

5.7.5 Administrador de Sistema

El Administrador de Sistema es responsable de la configuración y administración de la infraestructura de computación y red de una empresa, asegurando que las aplicaciones J2EE se ejecuten correctamente. El Administrador de Sistema, por lo general, hace uso de herramientas de monitoreo y administración proporcionadas por el proveedor de plataformas J2EE.

5.7.6 Proveedor de Herramientas

El Proveedor de Herramientas proporciona herramientas para el desarrollo y empaquetamiento de aplicaciones. Las herramientas independientes de la plataforma pueden ser usadas en todas las fases del desarrollo de las aplicaciones. Por otro lado, herramientas de administración, monitoreo y puesta en marcha de aplicaciones talvez dependan más de la plataforma para las cuales fueron construidas.

Especificaciones futuras de J2EE, esperan definir interfaces adicionales de tal forma que estas últimas herramientas sean independientes de la plataforma.

5.7.7 Proveedor de Componentes de Sistema

El Proveedor de Componentes de Sistema es el encargado de desarrollar componentes, que según la especificación de conectores, permiten comunicar a la plataforma con otros sistemas de la empresa incluyendo bases de datos y sistemas de mensajería.

6. Seguridad en J2EE

6.1 Introducción

El modelo de programación de aplicaciones J2EE libera al desarrollador de la implementación de mecanismos específicos de seguridad relacionada con la aplicación. Esto se debe a que se pretende reforzar el concepto de *Portabilidad* de las aplicaciones, de tal manera que una aplicación pueda ejecutarse sobre cualquier ambiente de seguridad.

Las aplicaciones J2EE están compuestas de componentes, los cuales pueden ser ejecutados en diferentes contenedores, conformando de esta forma aplicaciones de negocios multi-capas. El objetivo de la seguridad en J2EE es lograr niveles de seguridad entre los diferentes componentes y capas de las aplicaciones.

Las capas pueden contener tanto recursos protegidos como sin proteger. Los recursos protegidos sólo deben estar disponibles para usuarios autorizados. La Autorización provee de acceso controlado a los recursos protegidos. La Autorización esta basada en identificación y autentificación. La Identificación es el proceso que hace posible que un sistema reconozca una entidad, y la Autentificación es el proceso que verifica que el usuario o entidad es quien dice ser. En otras palabras, el usuario o entidad debe en primer lugar ser autentificado y luego identificado para autorizarle el uso de recursos protegidos.

La especificación no pretende limitar los productos J2EE en cuanto a mecanismos de seguridad se refiere, al contrario, lo que presenta la especificación son sólo los requerimientos básicos que la plataforma debe incluir para controlar el acceso a recursos, asegurar la integridad y confidencialidad de los datos, y proveer de mecanismos de autentificación y autorización de usuarios a recursos.

6.2 Arquitectura de Seguridad

Las siguientes características son propuestas para la arquitectura de seguridad de J2EE:

1. Portabilidad. La arquitectura debe apoyar el concepto "*Write Once, Run Anywhere*", es decir, se escribe una sola vez y se ejecuta en cualquier lugar.

2. Transparencia. Los Proveedores de Componentes de Aplicaciones no deben saber nada sobre seguridad para escribir una aplicación.

3. Aislamiento. La plataforma debe realizar la autentificación y control de acceso de acuerdo a instrucciones establecidas en los atributos de despliegue al momento de ser instalada y configurada la aplicación.

4. Extensibilidad. El uso de servicios de seguridad por parte de aplicaciones no debe comprometer su portabilidad.

5. Flexibilidad. Los mecanismos de seguridad y declaraciones usadas por aplicaciones bajo la especificación no deben imponer una política de seguridad particular.

6. Abstracción. Los requerimientos de seguridad de un componente de aplicación deben ser especificados utilizando descriptores de despliegue. En los descriptores de despliegue se deben declarar los roles de seguridad y requerimientos de acceso. Estas especificaciones deberán ser consistentes con las propiedades del ambiente en el cual se ejecute el componente, por lo cual pueden sufrir cambios. Se debe especificar qué propiedades de seguridad pueden sufrir cambios y cuales no.

7. Independencia. Los requerimientos de seguridad deben poder ser implementados utilizando la variedad de tecnologías de seguridad existentes.

8. Prueba de compatibilidad. Se debe poder determinar inequívocamente si una implementación es o no compatible con la arquitectura de seguridad.

9. Interoperabilidad segura. Los componentes de aplicaciones que se ejecutan en una plataforma J2EE determinada deben poder invocar servicios de otra plataforma, de otro proveedor, con las mismas o distintas políticas de seguridad. Los servicios pueden ser proporcionados por componentes Web o EJB.

6.3 Seguridad en Contenedores.

Un contenedor posee básicamente dos métodos para apoyar la seguridad de los componentes y aplicaciones en lo que se refiere a autorización:

1. Seguridad declarativa
2. Seguridad programada

6.3.1 Seguridad Declarativa.

La seguridad declarativa se refiere al mecanismo utilizado para expresar la estructura de seguridad de una aplicación incluyendo roles de seguridad, control de acceso y requerimientos de autentificación externos. El “Deployment Descriptor” o Descriptor de Despliegue es el vehículo principal para la seguridad declarativa en J2EE, dado que en él se especifican las políticas de seguridad relacionadas con el ambiente en que se ejecutará la aplicación.

En tiempo de ejecución, el Contenedor utiliza las políticas de seguridad establecidas para ejecutar mecanismos de seguridad específicos sobre un componente o un grupo de componentes.

6.3.2. Seguridad Programada.

Cuando la Seguridad Declarativa no basta para expresar el modelo de seguridad deseado para una aplicación, se hace uso de la Seguridad Programada, la cual utiliza aplicaciones externas especializadas en seguridad. Existe una API para la seguridad programada que consiste de dos métodos para los EJB (Enterprise Java Bean) y dos para los Servlet, la tabla 1 presenta los métodos relacionados con cada componente.

Componente	Métodos
EJB	isCallerInRole(EJBContext) para permisos getCallerPrincipal(EJBContext) para identidad
Servlet	isUserInRole(HttpServletRequest) getUserPrincipal(HttpServletRequest)

Tabla 1: Métodos utilizados en seguridad programada

Estos métodos permiten a los componentes tomar decisiones de lógica de negocios basados en roles de seguridad para los invocadores o usuarios remotos.

6.4 Seguridad Distribuida

Un ambiente se considera *distribuido* cuando los contenedores de distintos tipos de componentes se encuentran distribuidos en distintos servidores, dado que el proveedor de la plataforma permite esta configuración. En este escenario se debe tomar en cuenta que la

comunicación entre componentes de distintos contenedores se puede ver afectada a ataques de seguridad.

Para resolver este problema de amenaza se utilizan las *asociaciones seguras.* Una asociación segura es información de estado de seguridad compartida, que establece la base para asegurar la comunicación entre componentes. Una asociación segura puede involucrar: autentificación del cliente, negociación de calidad de protección (confidencialidad, integridad) y configuración del contexto para la asociación entre componentes. Todo lo anterior lo establecen los Contenedores.

6.5 Modelo de Autorización

El Modelo de Autorización para la plataforma J2EE esta basado en el concepto de roles de seguridad. Un rol de seguridad es una agrupación de usuarios definidos por el Proveedor de Componentes de Aplicaciones o el Ensamblador de Aplicaciones. Los roles de seguridad son utilizados tanto en la seguridad declarativa como en la programada.

La autorización declarativa puede ser usada para controlar el acceso a los métodos de un EJB, esto se puede lograr generando un listado de métodos, en el descriptor de despliegue, a los cuales puede tener acceso un determinado rol de seguridad. De esta forma al rol de seguridad se le permite el uso de un subconjunto específico de métodos de un determinado EJB.

7. Administración de Transacciones en J2EE

7.1 Introducción

Una aplicación de negocios por lo general accede y almacena información en una base de datos. Esta información debe ser fiable y actualizada para el correcto funcionamiento de la aplicación. La información almacenada en la base de datos puede ser distorsionada si muchas aplicaciones, o usuarios, acceden y actualizan los datos simultáneamente. También puede suceder que la información sea mal almacenada debido a una falla en la aplicación que manipula los datos. Para solucionar estos problemas se utilizan las *transacciones*, las que se encargan de coordinar el acceso simultáneo a los datos desde múltiples clientes y asegurar la integridad de los datos por eventuales fallas de las aplicaciones.

Una transacción de negocios involucra uno o más pasos que deben ser realizados con éxito. Por ejemplo, una transacción financiera para girar dinero desde un cajero automático puede seguir los siguientes pasos, los cuales se listan en seudo código:

```
Inicio de Transacción
       Verificación de debito en la cuenta corriente
       Actualización de montos cuenta corriente
       Actualización de historialde giros
       Entregar dinero
Finalización de Transacción
```

Para que la transacción sea exitosa se deben completar correctamente los cuatro pasos descritos anteriormente. Si uno de ellos falla se perderá la integridad de los datos, por lo cual se debe producir una vuelta atrás de los procesos involucrados con el fin de recuperar el estado inicial de la información. Por lo tanto, una transacción es una unidad de trabajo indivisible que unifica uno o más pasos o subtrabajos.

Una transacción termina de dos posibles formas, exitosamente (commit) o con fracaso (rollback). En el caso de que la transacción termine exitosamente todos los cambios realizados en la información por parte de cada una de los pasos son grabados en la base de datos. Por el contrario, si la transacción falla, dado de que uno o más pasos fallan, se debe deshacer los cambios hechos en cada paso volviendo al estado inicial que existía antes de comenzar la transacción. De esta forma se asegura la integridad de los datos.

7.2 Administración de transacciones

Las transacciones pueden ser administradas de dos formas:

- Transacción Administrada por el Contenedor
- Transacción Administrada por el Bean.

A continuación se describen en detalle cada uno de estos métodos de administración.

7.2.1 Transacciones Administradas por el Contenedor

Si las transacciones de un EJB son administradas a nivel del Contenedor, es el propio Contenedor quién fija los límites de la transacción. Esta modalidad es utilizada por cualquier tipo de EJB: Sesión, Entidad o dirigidos por mensajes. Las Transacciones Administradas por el Contenedor facilitan el desarrollo de un EJB ya que en el código no se explicitan los límites de la transacción, es decir, no se incluyen instrucciones de inicio y fin de la transacción.

Por lo general el Contenedor inicia una transacción antes de que se ejecute un método de un EJB. No se permiten transacciones anidadas dentro de un método. Es posible, en el momento que se desarrolla un EJB, indicar qué métodos estarán asociados a una transacción, fijando los atributos de la transacción.

7.2.1.1 Atributos de una transacción

Los atributos de una transacción controlan el alcance de ésta. En la figura 4, el método A inicia una transacción e invoca el método B de Bean 2. Cuando el método B se ejecuta ¿se ejecuta dentro del alcance del método A o inicia una nueva transacción? La respuesta a esta pregunta dependerá de los atributos de la transacción del método B.

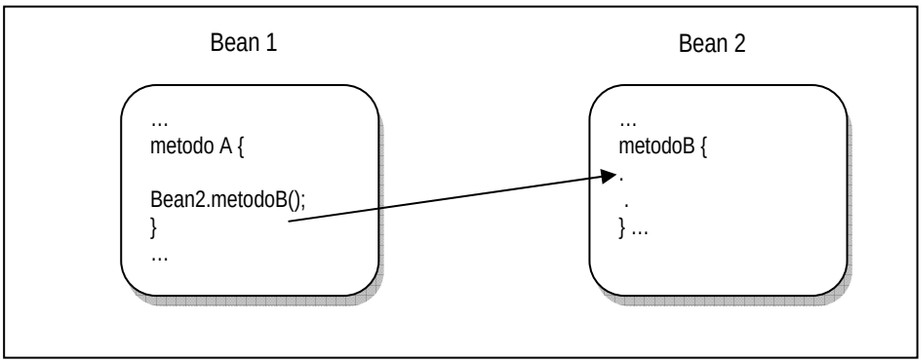


Fig. 4 Alcance de una transacción

Un atributo de transacción puede tener los siguientes valores:

- Required
- RequiresNew
- Mandatory
- NoSupported
- Supports
- Never

7.2.1.1.1 Required

Si el cliente se ejecuta dentro de una transacción e invoca un método de un Bean de negocio, el método se ejecuta dentro de la transacción del cliente. En el caso de que el cliente no

este asociado con una transacción, el Contenedor inicia una nueva transacción antes de la ejecución del método.

7.2.1.1.2 RequiresNew

Si el cliente se ejecuta dentro de una transacción e invoca un método de un Bean de negocio, el contenedor sigue los siguientes pasos:

1. Suspende la transacción del cliente.
2. Inicia una nueva transacción.
3. Ejecuta el método invocado.
4. Reanuda la transacción del cliente después de que el método sea completado.

En el caso de que el cliente no este asociado con una transacción, el Contenedor inicia una nueva transacción antes de la ejecución del método.

7.2.1.1.3 Mandatory

Si el cliente se ejecuta dentro de una transacción e invoca un método de un Bean de negocio, el método se ejecuta dentro de la transacción del cliente. En el caso de que el cliente no este asociado con una transacción, el Contenedor lanza una excepción, *TransactionRequiredException*.

7.2.1.1.4 NotSupported

Si el cliente se ejecuta dentro de una transacción e invoca un método de un Bean de negocio, el contenedor suspende la transacción del cliente y luego invoca el método. Luego de completar la ejecución del método, el contenedor reanuda la transacción del cliente.

En el caso de que el cliente no este asociado con una transacción, el Contenedor no inicia una nueva transacción.

7.2.1.1.5 Supports

Si el cliente se ejecuta dentro de una transacción e invoca un método de un Bean de negocio, el método se ejecuta dentro de la transacción del cliente. En el caso de que el cliente no este asociado con una transacción, el Contenedor no inicia una nueva transacción antes de la ejecución del método.

7.2.1.1.6 Never

Si el cliente se ejecuta dentro de una transacción e invoca un método de un Bean de negocio, el Contenedor lanza una excepción, *TransactionRequiredException*. En el caso de que el cliente no este asociado con una transacción, el Contenedor no inicia una nueva transacción antes de la ejecución del método.

7.2.1.2 Rollback de Transacciones Administradas por el Contenedor

Existen dos maneras de deshacer los cambios efectuados por una transacción que falla, disolviendo por ejemplo los cambios hechos por sentencias SQL a los datos almacenados en una base de datos. La primera se gatilla automáticamente desde el contenedor si ocurre una excepción de sistema. La segunda ocurre al invocar el método *setRollbackOnly* de la interfaz *EJBContext*, por medio de código fuente de un EJB. El segundo caso se puede utilizar cuando ocurre una excepción a nivel de la aplicación.

7.2.2 Transacciones Administradas por el Bean

En una transacción administrada por el EJB, el código fuente del EJB de sesión es quién demarca los límites de la transacción. En un EJB entidad, la transacción no puede ser administrada por en EJB, este debe utilizar transacciones administradas a nivel del Contenedor. Sólo los EJB Sesión y Dirigidos por Mensajes pueden administrar una transacción.

Cuando se decide administrar la transacción a nivel de EJB se debe elegir utilizar transacciones JDBC o JTA. Las transacciones JDBC son administradas por el DBMS.

7.3 Transacciones JTA

JTA es la abreviación de API de Transacciones Java (Java Transaction API). Esta API permite demarcar transacciones de una manera independiente de la implementación del administrador de transacciones. Una transacción JTA, es administrada por el administrador de transacciones J2EE.

Un administrador de transacciones de un DBMS particular puede que no funcione con otras bases de datos. El administrador de transacciones J2EE, puede funcionar con diferentes bases de datos, pero tiene la limitación de que debe trabajar con una a la vez, es decir no soporta transacciones anidadas.

7.4 Transacciones en Componentes Web

Tanto en Servlets como en páginas JSP se utilizan las interfaces *java.sql.Connection* o *javax.transaction.UserTransaction* para demarcar una transacción. Estas interfaces deben ser implementadas por el proveedor de la plataforma.

7.5 Transacciones JDBC

La plataforma debe apoyar la tecnología JDBC como un administrador de recursos transaccionales. La plataforma debe disponer de APIs que permitan a los componentes Web y EJB acceder a las transacciones JDBC.

8. Sistema de Nombrado en J2EE

8.1 Introducción

Tanto los EJB como los Componentes Web pueden acceder a una amplia variedad de recursos, como por ejemplo: bases de datos, objetos para servicios de mensajería, URLs, etc. Por lo anterior, se ha establecido un mecanismo en común para acceder a estos recursos, por lo cual los contenedores de estos componentes deben proporcionar un ambiente de nombramiento para referirse a sus recursos.

JNDI es el acrónimo de la interfaz de programación para Java Naming and Directory Interface (Interfaz de Directorios y Nombramiento Java), el que proporciona funcionalidades de nombrado y directorio a las aplicaciones escritas en Java. JNDI esta definido para ser independiente de cualquier implementación de servicio de directorios, por lo que proporciona una interfaz unificada para múltiples servicios de nombramiento y directorio.

Un *nombramiento JNDI* es un nombre “amistoso” para referirse a un objeto. El servicio de JNDI proporcionado por el servidor J2EE dispone de mecanismos para poder asociar un objeto a un nombramiento y directorio.

Un *Connection Factory* (fábrica de conexiones) es un objeto que produce objetos de conexión que permiten a los componentes J2EE acceder a recursos. Por ejemplo el Conection Factory para una base de datos es el objeto *javax.sql.DataSource* el cual crea un objeto *java.sql.Connection*.

Una *referencia de recurso* es un elemento en un descriptor de despliegue que identifica el nombre codificado del componente para el recurso. Más específicamente, el nombre codificado

hace referencia a una Connection Factory para el recurso. En el ejemplo anterior, el nombre de referencia al recurso es el jdbc/SavingsAccountDB.

Específicamente en J2EE los requerimientos de JNDI están dirigidos a los siguientes usos:

- Para que el desarrollador o ensamblador de aplicaciones pueda personalizar la lógica de negocio de la aplicación, sin tener que acceder al código fuente de la aplicación.
- Para que las aplicaciones puedan acceder a recursos e información externa en su ambiente operacional, sin tener la necesidad de conocer cómo la información externa es nombrada u organizada.

8.2 Contexto del nombrado JNDI

El ambiente de nombramiento de un componente de aplicación permite personalizar la lógica de negocio de un componente de aplicación durante su despliegue o ensamblaje, sin la necesidad de intervenir en su código fuente.

El contenedor de componentes debe proporcionar una implementación de JNDI que almacenará las variables de ambiente del componente de la aplicación. El contenedor debe además proveer de herramientas que permitan crear y administrar el ambiente de cada componente de aplicación. Por otro lado, los proveedores de componentes deben utilizar el descriptor de despliegue para declarar los valores de configuración del ambiente para un determinado componente, los valores se definen por mediante una descripción, nombre, tipo de dato y su valor. Cada componente tiene su propia configuración de ambiente, la cual no puede ser modificada en tiempo de ejecución, por esta razón el contenedor debe asegurar que instancias de componentes de aplicación sólo tengan acceso de lectura a las variables de ambiente.

8.3 Referencia a Enterprise JavaBeans

Los EJB cuentan con su propio formato y uso de descriptores que le permiten ser referenciados, estos se conocen como "*referencias EJB*". Las referencias EJB son entradas especiales en el ambiente de nombramiento de los componentes de aplicaciones.

El descriptor de despliegue también permite al ensamblador de aplicaciones enlazar una referencia EJB declarada en un componente de aplicación, con un EJB contenido en un archivo JAR perteneciente a la misma aplicación.

8.4 Responsabilidades del Proveedor de Componentes de Aplicaciones.

El Proveedor de Componentes de Aplicaciones debe utilizar la referencia EJB para localizar la *interfaz Home* del EJB. Para lo anterior se debe asignar una entrada en el ambiente de componente de aplicación para la referencia, el elemento entrada en el descriptor de aplicaciones debe ser ejb-ref el cual tiene un elemento opcional description y los elementos obligatorios ejb-ref-name, ejb-ref-type,ejb-link, home y remote. La tabla 2 presenta la función de cada elemento en el descriptor de despliegue.

Elemento	Modalidad	Función
Description	Opcional	Descripción de la referencia
ejb-ref-name	Obligatorio	Especifica el componente que hará referencia a un bean.
ejb-ref-type	Obligatorio	Especifica el tipo esperado de bean (entidad o sesión)
Ejb-link	Opcional	Se utiliza para enlazar una referencia EJB a un bean específico.
home	Obligatorio	Especifican los tipos de datos de las interfaces remote y home del entreprise bean referenciado.
remote	Obligatorio	

Tabla 2: Elementos del descriptor de despliegue

8.5 Responsabilidades del Ensamblador de Aplicaciones

El ensamblador de aplicaciones puede hacer uso del elemento ejb-link del descriptor de despliegue para enlazar una referencia EJB.

8.6 Responsabilidades del Desarrollador

El desarrollador debe asegurar que todas las referencias EJB declaradas estén enlazadas a las interfaces *home* de los EJB que existen en el ambiente operacional. Debe asegurar además que los tipos de beans (entidad o sesión) sean compatibles con los declarados en el descriptor.

8.7 Responsabilidades del proveedor de la plataforma J2EE

El proveedor de plataformas debe proporcionar herramientas que permitan el desempeño de las tareas descritas anteriormente. Estas herramientas deben permitir preservar información sobre el ensamblaje e informar sobre referencias erróneas para su corrección.

Capítulo III

Tecnologías Java para J2EE

1. Introducción

En el siguiente capítulo se presentan las principales tecnologías Java para la creación de aplicaciones Web. Las tecnologías que a continuación se describen, se ejecutan bajo el ambiente operacional de la plataforma J2EE, haciendo uso de todas las características y beneficios vistos en el capitulo anterior.

2. Java Server Pages

2.1 Introducción

JSP es el acrónico de Java Server Pages, tecnología basada en Java que simplifica el proceso de desarrollo de sitios Web dinámicos. Las páginas JSP son documentos basados en texto, que combinan código Java con algún lenguaje de marcas, como HTML ó XML, para generar dinámicamente el contenido de un documento.

2.2 Sitios Web Dinámicos

El World Wide Web, como originalmente fue diseñado, es un medio estático. El Hipertext Transfer Protocol (HTTP) o Protocolo de Transferencia de Hipertexto define un proceso general para la solicitud de páginas Web. El cliente (navegador o browser Web) invoca una página, luego el servidor Web responde entregando un flujo de datos, que por lo general son contenidos HTML e imágenes asociadas a este documento. Esta respuesta constituye la entrega de documentos (imágenes, HTML y otros) estáticos que se encuentran almacenados en forma de archivos en el servidor Web, estos nunca cambian dependiendo de quién lo solicite.

Por el contrario, los sitios Web dinámicos tienen capacidad de generar contenido cada vez que se solicite un documento. Es posible generar texto, imágenes y otros contenidos multimedia en forma dinámica. La generación de contenido dinámico requiere que el servidor Web procese cada solicitud para generar respuestas personalizadas. La personalización de los contenidos puede variar dependiendo de valores de parámetros incluidos en la solicitud de un documento.

Como parte de la familia de la tecnología Java, con JSP podemos desarrollar aplicaciones Web independientes de la plataforma.

Una característica importante de JSP es que nos permite desarrollar aplicaciones multicapas, en donde es posible separar los datos, la lógica del negocio y la lógica de presentación. Esto es posible ya que las páginas JSP pueden acceder directamente a componentes Java Beans o EJB, creando instancias de éstos, estableciendo sus propiedades e invocando sus métodos directamente para obtener la información necesaria para desplegar la interfaz usuario.

La figura 5 muestra el caso típico de cómo las páginas JSP se utilizan para generar contenido en forma dinámica mediante la conexión a una base de datos. La lógica de generación de contenidos reside en las páginas JSP.

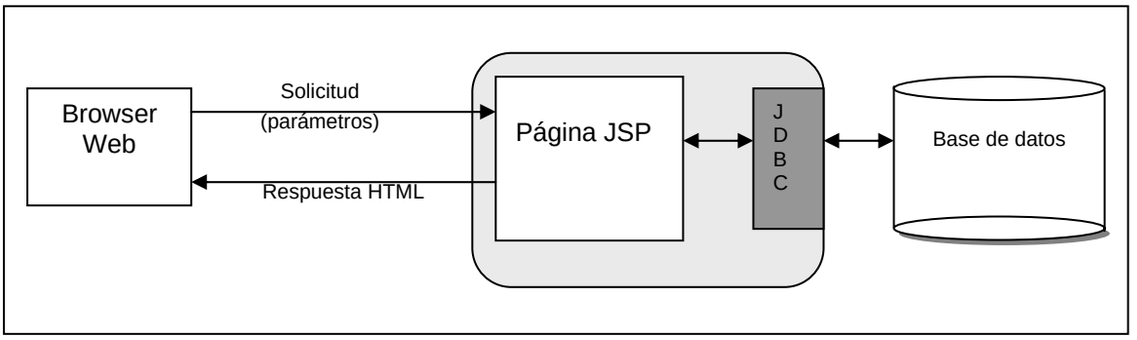


Fig. 5 Esquema general de conexión JSP y Base de Datos

Cuando el navegador del cliente solicita la página JSP, esta se ejecuta en el servidor, específicamente en el contenedor de componentes Web, generando contenido HTML en forma dinámica a través de datos extraídos desde una base de datos. La respuesta puede variar según parámetros entregados al momento de realizar la solicitud (parámetros POST o GET)

2.3 Objetos implícitos de las páginas JSP

Las páginas JSP contienen objetos implícitos, que permiten simplificar las tareas básicas de una aplicación, como por ejemplo gestionar la sesión de un usuario. No es necesario declarar ni crear instancias los objetos implícitos. Entre estos objetos podemos mencionar:

- request. Solicitud HTTP enviada a la página JSP desde un cliente.
- response. Objeto respuesta generado por la página JSP.
- out. Flujo de salida enviado de vuelta al cliente. En este objeto se almacena la salida generada, que por lo general es contenido en formato HTML.
- aplication. Es una instancia de la clase java.servlet.ServletContext, y su ámbito de utilización es de tipo application. Representa la aplicación Web en el que se está ejecutando la página JSP.
- config. Es una instancia de la clase javax.servlet.ServletConfig y maneja todo lo relacionado a la configuración del servlet generado al solicitar la página JSP. Su ámbito de funcionamiento es de tipo page.
- exception. Es una instancia de java.lang.Throwable. Representa un objeto excepción que provoca la invocación de una página de error. Este objeto solamente está disponible en la página de error (es la página que tiene errorPage=true en la directiva page). Tiene alcance page.
- pageContext. Es una instancia de javax.servlet.jsp.PageContext. Encapsula el contexto de la página para un JSP específico. Tiene alcance page.

- session. Es una instancia de la clase javax.servlet.http.HttpSesion. La función de este objeto es manejar todas las acciones relacionadas a la sesión del usuario, por lo tanto su ámbito de utilización es de tipo sesión. Una sesión es creada automáticamente, a menos que se especifique lo contrario, cuando un usuario solicita una página JSP, de esta manera podemos almacenar información relativa a ese usuario. Este objeto actúa como un contenedor de otros objetos, los cuales pueden ser incorporados, consultados y eliminados mientras dure activa la sesión del usuario.

Un elemento clave del modelo de aplicaciones J2EE es el uso del Web como el mecanismo preferido para la entrega de información, adoptando HTML como interfaz de aplicaciones de negocio. La ventaja de este modelo es que el navegador Web ha sido establecido como un estándar de facto para acceder y compartir datos a través de Internet e Intranet. De esta forma, no es necesario que el usuario instale software adicional para ejecutar una aplicación, ni siquiera es necesario cuando la versión de la aplicación cambia, ya que al ser instalada en el servidor queda a disposición de todos los clientes en forma automática.

2.4 Ciclo de vida de una página JSP

Como se muestra en la figura 6, en el momento en que un cliente invoca una página JSP desde un navegador, la página JSP es convertida automáticamente en un servlet, para posteriormente ser cargado en memoria y ejecutado en el contenedor.

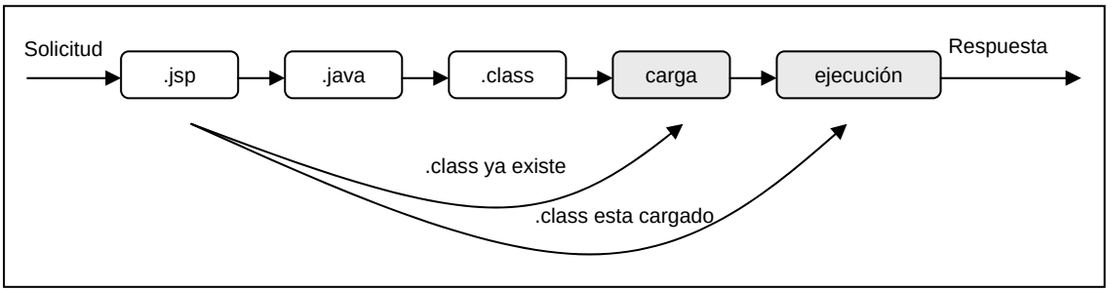


Fig. 6 Ciclo de vida de una página JSP

Los siguientes pasos describen las acciones que el contenedor de componentes Web desarrolla al ser solicitado un componente JSP [All, 00]:

1. La página JSP (archivo .jsp) es traducida a código fuente java (archivo .java)
2. El código fuente es compilado, produciendo como resultado un Servlet como clase (archivo .java y .class)
3. El Servlet Java es cargado en memoria.
4. El Servlet es ejecutado.

Estos pasos ocurren cuando se llama por primera vez a una página JSP. En las solicitudes siguientes, el contenedor compara la antigüedad de la página JSP con la clase compilada, en caso de no haber diferencias se verifica si clase esta cargada en memoria y se ejecuta. Cuando se realizan cambios en la página JSP, se vuelven a realizar los pasos 1, 2, 3 y 4 con la primera solicitud del componente, para mostrar la última versión desarrollada.

3. Servlets

3. 1 Introducción

Los servlets son componentes que se ejecutan en el servidor Web, específicamente en el contenedor para componentes Web. Están escritos en Java y son un mecanismo simple y eficiente para extender las funcionalidades de un servidor Web mediante el acceso a la familia entera de APIs Java, como por ejemplo, conexión a bases de datos por medio de API JDBC.

A diferencia de los CGI (Common Gateway Interface), los cuales son escritos en C++ o Perl para plataformas especificas, los servlet son implementados en Java y heredan todas las características, y en especial la portabilidad, de este lenguaje.

Los servlets son utilizados en la implementación de aplicaciones Web ya que ellos soportan el protocolo HTTP (solicitud – respuesta).

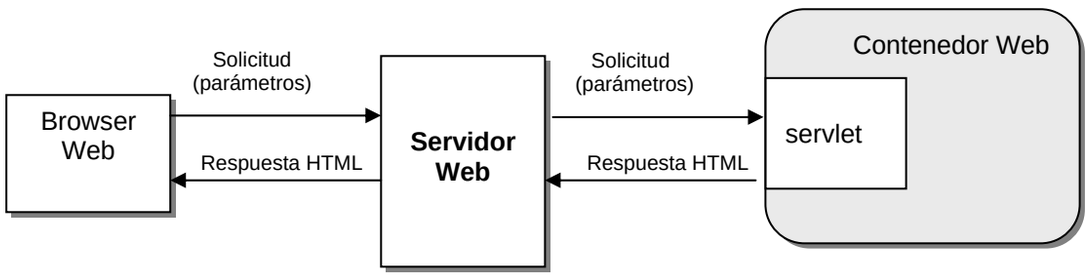


Fig. 7 Invocación de un servlet

En la figura 7, se puede observar que el servidor Web recibe una solicitud HTTP, este determina que la solicitud hace referencia a un servlet y envía la solicitud al contenedor de componentes Web, el que invoca el servlet apropiado. El servlet procesa la solicitud y entrega

una respuesta, que el servidor Web se encarga de hacer llegar al cliente en forma de documento HTML o XML.

3.2 Solicitudes y respuestas HTTP

Un servlet es invocado cuando una solicitud HTTP lo referencia directamente como un Java servlet o indirectamente como una página JSP. Una de las tareas más comunes de los servlet es acceder a información almacenada dentro de la solicitud HTTP, procesar la información y entregar un resultado al cliente como parte de la respuesta HTTP.

La solicitud HTTP contiene información enviada al servlet desde el cliente. Por ejemplo, si el servlet es invocado desde un formulario Web, el servlet debería en primer lugar acceder a los datos almacenados el la solicitud antes de realizar cualquier proceso.

En Java se puede acceder a datos almacenados en la solicitud HTTP a través del objeto javax.servlet.HttpServletRequest.

El servlet responde a la solicitud construyendo una respuesta HTTP y enviando la respuesta al cliente. Para la construcción de la respuesta se utilizan los métodos definidos en el objeto javax.servlet.HttpServletResponse.

3.3 Trabajando con sesiones de usuarios

Por naturaleza HTTP es un protocolo sin estado. Un servidor Web recibe una solicitud y entrega una respuesta, una vez que la respuesta es entregada se termina la conexión con el cliente. El servidor Web no mantiene información sobre el cliente, por lo cual no puede determinar cuando una solicitud proviene del mismo cliente. La incapacidad de realizar el seguimiento de

cada cliente, y su navegación a través del sitio Web, torna muy complicado la realización de transacciones. Sin embargo, los contenedores de componentes Web tienen la capacidad de trabajar con sesiones de usuarios, lo cual permite gestionar las interacciones entre el cliente y el servidor. Para el correcto funcionamiento del objeto *session* es preciso que el navegador (cliente) cuente con soporte de cookies o que eventualmente el cliente no las haya deshabilitado. El manejo de sesiones se realiza a través del objeto javax.servlet.http.HttpSession.

La gran desventaja de los sevlet es que es muy tedioso entregar respuestas en formatos HTML al cliente, haciendo engorroso y poco optimo, la mezcla de interfaz usuario con la lógica de la aplicación. El siguiente ejemplo ilustra lo mencionado:

```
public class HelloWorld extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res)   throws ServletException, IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter( );
        out.println("<html>");
        out.println("<head><title>Hello World</title></head>");
        out.println("<body>");
        out.println("<h1>Hello world !</h1>");
        out.println("</body></html>");
     out.close();
}
public String getServletInfo( ) {
   return "This is HelloWorld servlet.";
}
}
```

3.4 Beneficios de los Servlets

Los servlet cuentan con muchos beneficios, algunos de estos están asociados con el lenguaje de programación Java, mientras que otros son propios de la tecnología servlet, entre estos se pueden citar:

- Seguridad

  Dado que los servlet son invocados a través de servidores Web, la lógica de negocios de la aplicación no queda expuesta directamente al cliente. Además los servlets se encuentran aislados entre si, es decir, un error causado por un servlet corrupto no afecta el funcionamiento de otro.

- Desempeño

  Una de las diferencias más destacadas entre los servlet y aplicaciones que se ejecutan el servidor, como por ejemplo CGIs, es el desempeño. Los servlet son cargados en memoria una sola vez, al momento de ser solicitados, y no necesitan ser cargados una y otra vez, como ocurre con los CGI. Además los servlet son multihilos. y se ejecutan en procesos del servidor servlet o contenedor.

- Portabilidad

  Dado que los servlet tienen su propia especificación y forman parte integral de la plataforma J2EE, nos aseguran que las aplicaciones puedan funcionar en cualquier servidor J2EE o contenedor de servlet que cumplan con las especificaciones entregadas por Sun Microsystem. La portabilidad es importante para los desarrolladores de servlet, ya que esta característica implica que no deben implementar diferentes versiones según la plataforma en que se ejecutaran sus productos.

- Estabilidad

  Los servlet se ejecutan fuera del proceso del servidor Web, por lo tanto si un servlet falla, sólo el proceso que ejecuta el servlet se verá afectado.

4. Enterprise Java Beans

4.1 Introducción

Los componentes Enterprise Java Beans o EJB proporcionan mecanismos para implementar servicios de la capa de negocio o middleware de una aplicación. Los EJB definen un modelo de componentes de software mediante el cual la lógica de las aplicaciones se desarrolla utilizando servidores que cumplan con la especificación de los EJB.

Los EJB definen un escenario en el cual diferentes aplicaciones de negocio pueden funcionar y convivir en el mismo servidor, sin causar conflictos.

Se entiende por aplicaciones de negocio aquellas aplicaciones informáticas de gestión que se implantan en grandes corporaciones, con múltiples oficinas distribuidas por toda la geografía nacional (o internacional). Suelen ser aplicaciones distribuidas, usadas por múltiples clientes, que hacen un uso intensivo de transacciones, bases de datos y políticas de seguridad.

La arquitectura Enterprise JavaBeans es el núcleo de J2EE y hace uso intensivo de librerías incluidas en esta plataforma. Algunos ejemplos de APIs Java usados por la arquitectura EJB son los siguientes:

- JNDI para acceder a recursos
- JMS para tratar con mensajes
- JTA para programar explícitamente las transacciones

La tecnología Enterprise JavaBeans esta siendo apoyada por la mayor parte de las empresas de informática que se orientan al mundo de la empresa, entre ellas IBM, ORACLE y

SUN, y también por los clientes de estas empresas, que valoran de forma positiva, entre otros aspectos, el carácter abierto de la arquitectura.

En Marzo de 1998 Sun Microsystems propuso la especificación 1.0 de la arquitectura Enterprise JavaBeans. Esta especificación comienza con la siguiente definición: “La arquitectura Enterprise JavaBeans esta basada en componentes para el desarrollo y puesta en marcha de aplicaciones empresariales distribuidas y orientadas a objetos. Las aplicaciones escritas usando la arquitectura Enterprise JavaBeans son escalables, transaccionales y seguras en ambientes multiusuarios. Estas aplicaciones se escriben una vez, y se implantan en cualquier servidor que soporte la especificación Enterprise JavaBeans”.

Aunque se han introducido nuevas versiones de la especificación, que incorporan muchas mejoras, la definición de la arquitectura sigue siendo la misma.

La tabla 4 presenta un resumen con la evolución de las especificaciones de la arquitectura EJB.

Especificación EJB	Fecha	Principales novedades
EJB 1.0	Marzo 1998	Propuesta inicial de la arquitectura EJB. Se introducen los beans de sesión y los de entidad (de implementación opcional). Persistencia manejada por el contendedor en los beans de entidad. Manejo de transacciones. Manejo de seguridad.
EJB 1.1	Diciembre 1999	Implementación obligatoria de los beans de entidad. Acceso al entorno de los beans mediante JNDI.
EJB 2.0	Agosto 2001	Manejo de mensajes con los beans dirigidos por mensajes. Relaciones entre beans manejadas por el contenedor. Uso de interfaces locales entre beans que se encuentran en el mismo servidor. Consultas de beans declarativas, usando el EJB QL.
EJB 2.1	Agosto 2002	Soporte para servicios Web. Temporizador manejado por el contenedor de beans. Mejora en el EJB QL.

Tabla 4. Evolución de las especificaciones de la arquitectura Enterprise JavaBeans

Entre los objetivos que se enumeraban en ese primer documento de especificación 1.0 se encuentran los siguientes:

- Definir una arquitectura de componentes estándar con la cual construir aplicaciones de negocio (business applications) distribuidas y orientadas a objetos en el lenguaje de programación Java. Los Enterprise JavaBeans permitirán construir aplicaciones, combinando componentes desarrollados por herramientas de distintas compañías, lo que significa libertad de elección de productos.

- La arquitectura Enterprise JavaBeans hará fácil la construcción de aplicaciones. Los desarrolladores de aplicaciones no tendrán que entender los detalles de bajo nivel, referidos al manejo de transacciones, de estados, multi-hilos, pooling de recursos y otras APIs complejas y de bajo nivel, lo cual les permitirá concentrarse en la implementación de la lógica de negocios de la aplicación.

- Las aplicaciones Enterprise JavaBeans seguirán la filosofía “escribe una vez, ejecuta en cualquier lugar” del lenguaje de programación Java. Una vez implementado el EJB, éste puede ejecutarse en múltiples plataformas sin necesidad de recompilarlo o modificar su código fuente.

Los EJB, como componentes de la arquitectura de aplicaciones J2EE, se ejecutan en un entorno especial denominado contenedor EJB. El contenedor hospeda y maneja un EJB de la misma forma que un contenedor de componentes Web hospeda un servlet. Un EJB no puede funcionar fuera de un contenedor EJB. El contenedor EJB maneja cualquier aspecto del EJB en tiempo de ejecución, incluyendo acceso remoto éste, seguridad, persistencia, transacciones, concurrencia y acceso. El contenedor EJB también suele proporcionar servicios relacionados con la escalabilidad de la aplicación, como son la definición de clusters de contenedores, balanceo de carga y tolerancia a fallos.

El contenedor aísla el EJB del acceso directo de las aplicaciones cliente. Cuando una aplicación cliente invoca un método remoto en un EJB, el contenedor intercepta la invocación para asegurar que la persistencia, transacciones y seguridad se apliquen correctamente. De esta forma, el desarrollador de EJB puede concentrarse en encapsular correctamente la lógica de negocio, mientras que el contenedor se encarga de todo lo demás.

El contenedor EJB se ejecuta a su vez en una máquina virtual Java, con lo que tiene acceso a toda la infraestructura proporcionada por este lenguaje de programación.

Los contenedores manejan múltiples EJB simultáneamente. Para reducir el consumo de memoria y el procesamiento, los contenedores manejan (pool) los recursos y los ciclos de vida de los beans de forma muy cuidadosa. Cuando un EJB no está siendo usado, el contenedor lo situa en un almacén o pool para ser reutilizado por otro cliente, o lo eliminará de la memoria y sólo lo recuperará cuando sea necesario.

Debido a que las aplicaciones clientes no tienen acceso a los EJB, la aplicación cliente desconoce completamente las actividades de manejo de recursos del EJB. Por ejemplo, un EJB que no está siendo usado puede ser eliminado de la memoria, mientras que su referencia remota en el cliente permanece intacta. Cuando el cliente invoca un método sobre la referencia remota, el contenedor simplemente recupera el EJB para dar servicio a la petición. Para la aplicación cliente resulta transparente este proceso.

Un EJB depende del contenedor para todo lo que necesita. Si un EJB tiene que acceder a una conexión JDBC o a otro EJB, lo hace a través del contenedor; si un EJB tiene que acceder a la identidad de su invocador, obtener una referencia a él mismo, o acceder a propiedades, lo hace a través del contenedor.

Los componentes EJB encapsulan por lo general un proceso o una entidad de negocio. Un EJB, por ejemplo, podría calcular el sueldo e imposiciones de un trabajador, o encapsular la información sobre una cuenta bancaria que se encuentra físicamente en una base de datos relacional. Una solicitud de un cliente realiza una llamada a los métodos de negocio en el EJB y esta llamada provoca una invocación remota que llega al contenedor EJB.

Existen tres tipos de EJB:

1. Los EJB de sesión, realizan una tarea a petición de un cliente (request), pero no se corresponden con ninguna entidad persistente de negocio. Pueden ser a su vez de dos tipos: con estado (stateful) y sin estado (stateless). El estado lo almacenan localmente en la memoria, no en almacenamiento secundario, y dura el tiempo que está activa la sesión con el cliente.

2. Los EJB de entidad, representan objetos persistentes de negocio, normalmente datos existentes en una o más bases de datos. Por lo general, un EJB de entidad representa una fila de una tabla o vista de una base de datos relacional.

3. Los EJB dirigidos por mensajes, permiten procesar mensajes asíncronos generados por otros EJB o por aplicaciones externas que necesitan conectarse al sistema.

El cliente que usa EJB es una aplicación Java independiente, por ejemplo, un servlet o una página JSP, en definitiva, se trata de código Java que tiene acceso a las interfaces definidas para cada EJB. Más adelante se explica con detalle cada tipo de EJB.

Existen dos tipos de acceso a los EJB:

- Acceso remoto. El cliente usa RMI-IIOP, Invocación de Métodos Remotos Java que se ejecuta sobre el Protocolo Internet Inter-Orb desarrollado por IBM y Sun, para comunicarse con el EJB. RMI-IIOP combina las mejores características de RMI y Corba, permitiendo acelerar el desarrollo de aplicaciones distribuidas basadas en Java [URL3]. El desarrollador del EJB define dos tipos de interfaces: la mencionada *interfaz remota*, en la que se definen la interfaz de los procesos de negocio, y la interfaz *home*, en la que se

define la interfaz de los métodos de gestión (creación, borrado, etc.) de las instancias del EJB.

- Acceso local. Si el cliente se ejecuta en la misma máquina virtual Java que el EJB, puede acceder a versiones locales de las interfaces. De esta forma no es necesario serializar los parámetros ni los valores devueltos por el EJB, mejorándose de esta forma la eficiencia de las llamadas.

4.2 Tipos de EJB

Como se mencionó anteriormente existen tres tipos de EJB que se detallan a continuación [URL4].

4.2.1 EJB de sesión

Los EJB de sesión representan sesiones interactivas. Los EJB de sesión pueden mantener un estado, pero sólo durante el tiempo que el cliente interactúa con el EJB. Esto significa que los EJB de sesión no almacenan sus datos en una base de datos después de que el cliente termina el proceso. Por ello se suele decir que los EJB de sesión no son persistentes.

A diferencia de los EJB de entidad, los EJB de sesión no son compartidos entre más de un cliente, es decir, existe una correspondencia uno-uno entre EJB de sesión y un cliente. Por esta razón, el contenedor EJB no necesita implementar mecanismos de manejo de concurrencia en el acceso a estos EJB.

Existen dos tipos de EJB de sesión: con estado y sin estado:

4.2.1.1 EJB de sesión sin estado (stateless).

Los EJB de sesión sin estado no son modificados con las llamadas de los clientes. Los métodos que ponen a disposición de las aplicaciones clientes son llamadas procedurales que reciben datos y devuelven resultados, pero que no modifican internamente el estado del EJB. Esta propiedad permite que el contenedor EJB pueda crear un almacén (pool) de instancias, todas ellas del mismo EJB de sesión sin estado y asignar cualquier instancia a cualquier cliente

Cuando un cliente invoca un método de un EJB de sesión sin estado, el contenedor EJB obtiene una instancia del almacén. Cualquier instancia servirá, ya que el EJB no puede guardar

ninguna información referida al cliente. Tan pronto como el método termina su ejecución, la instancia del EJB está disponible para otros clientes. Esta propiedad hace que los EJB de sesión sin estado sean muy escalables para un gran número de clientes. El contenedor EJB no tiene que mover sesiones de la memoria a un almacenamiento secundario para liberar recursos, simplemente puede obtener recursos y memoria destruyendo las instancias.

Es apropiado usar EJB de sesión sin estado cuando una tarea no está ligada a un cliente específico. Un ejemplo de utilización podría ser un EJB que calcula las cuotas de un crédito de consumo de un banco, pasándole como parámetros los meses y cantidad solicitada.

Para mejorar la eficiencia de la aplicación, se debería usar EJB de sesión sin estado si se cumple alguna de las siguientes condiciones:

- El estado del EJB no contiene ningún dato para un cliente específico.
- En una única invocación de un método, el EJB realiza una tarea genérica que puede ser solicitada por cualquier cliente.
- El EJB obtiene desde una base de datos un conjunto de datos de sólo lectura que se usan a menudo por los clientes. Un EJB de este tipo, por ejemplo, podría obtener las filas de tabla que representan los productos que están a la venta.

4.2.1.2 Beans de sesión con estado (stateful).

En un EJB de sesión con estado, las variables de instancia del EJB almacenan datos específicos obtenidos durante la conexión con el cliente. Cada bean de sesión con estado, por tanto, almacena el estado conversacional de un cliente que interactúa con el EJB. Este estado conversacional se modifica conforme el cliente invoca los métodos de negocio del EJB. El estado conversacional no se guarda cuando el cliente termina la sesión. Un ejemplo de EJB de sesión

con estado podría ser un carro de la compra de una tienda virtual, en donde el cliente va guardando uno a uno los artículos que va comprando.

En general, se debería usar un EJB de sesión con estado si se cumplen las siguientes circunstancias:

- El estado del EJB representa la interacción entre el EJB y un cliente específico.
- El EJB necesita mantener información del cliente a lo largo de un conjunto de invocaciones de métodos.
- El EJB hace de intermediario entre el cliente y otros componentes de la aplicación, presentando una vista simplificada al cliente.

4.2.2 EJB de entidad

Los EJB de entidad modelan conceptos de negocio que puede expresarse como nombres. Esto es una regla sencilla más que un requisito formal, pero ayuda a determinar cuándo un concepto de negocio puede ser implementado como un EJB de entidad. Los EJB de entidad representan “cosas”: objetos del mundo real como hoteles, habitaciones, expedientes o estudiantes. Un EJB de entidad puede representar incluso cosas abstractas como una reserva o un pago. Los beans de entidad describen tanto el estado como la conducta de objetos del mundo real y permiten a los desarrolladores encapsular las reglas de datos y de negocio asociadas con un concepto específico. Por ejemplo un EJB Estudiante encapsula los datos y reglas de negocio asociadas a un estudiante. Esto hace posible manejar de forma consistente y segura los datos asociados a un concepto.

Los EJB de entidad se corresponden con datos en un almacenamiento persistente (base de datos, sistema de archivos, etc.). Las variables de instancia del EJB representan los datos en las columnas de la base de datos. El contenedor debe sincronizar las variables de instancia del EJB con la base de datos. Los EJB de entidad se diferencian de los EJB de sesión en que las variables de instancia se almacenan de forma persistente.

Son muchas las ventajas de usar EJB de entidad en lugar de acceder a la base de datos directamente. El uso de EJB de entidad para transformar en objetos los datos, proporciona a los programadores un mecanismo más simple para acceder y modificar información. Es mucho más fácil, por ejemplo, cambiar el nombre de un estudiante llamando a estudiante.setNombre() que ejecutando un comando SQL contra la base de datos. Además, el uso de objetos favorece la reutilización del software. La representación de los datos como EJB de entidad hace que el desarrollo sea más sencillo y menos costoso.

Los EJB de entidad se diferencian de los EJB de sesión, principalmente, en que los primeros son persistentes, permiten el acceso compartido, tienen clave primaria y pueden participar en relaciones con otros EJB de entidad.

Persistente significa que el estado del EJB de entidad existe más tiempo que la duración de la aplicación o del proceso del servidor J2EE.

Los EJB de entidad tienen dos tipos de persistencia: Persistencia Gestionada por el EJB (BMP, Bean-Managed Persistence) y Persistencia Gestionada por el Contenedor (CMP, Container-Managed Persistence). En el primer caso (BMP) el EJB de entidad contiene el código que accede a la base de datos. En el segundo caso (CMP) la relación entre las columnas de la base de datos y el EJB se describe en el fichero de propiedades del EJB, y el contenedor EJB se ocupa de la implementación. En este caso el código del EJB no contiene ninguna llamada SQL

4.2.3 EJB dirigidos por mensajes

Estos EJB permiten que las aplicaciones J2EE reciban mensajes JMS de forma asíncrona. Así, el hilo de ejecución de un cliente no se bloquea cuando está esperando que se complete algún método de negocio de otro EJB. Los mensajes pueden enviarse desde cualquier componente J2EE (una aplicación cliente, otro EJB, o un componente Web) o por una aplicación o sistema JMS que no use la tecnología J2EE.

Las instancias de un EJB dirigido por mensajes no almacenan ningún estado conversacional ni datos del cliente. Todas las instancias de los EJB dirigidos por mensajes son equivalentes, lo que permite al contenedor EJB asignar un mensaje a cualquier instancia. El contenedor puede almacenar estas instancias para permitir que los flujos de mensajes sean procesados de forma concurrente. Un único EJB dirigido por mensajes puede procesar mensajes de múltiples clientes.

4.3 Tipos de acceso a los EJB

Al diseñar aplicaciones J2EE, una de las primeras decisiones que hay que tomar es el tipo de acceso que van a utilizar en los EJB, es decir, decidirse por *remoto* o *local*.

4.3.1 Acceso remoto

Un cliente remoto de un EJB tiene las siguientes características

- Puede correr en una máquina física distinta o en una máquina virtual Java (JVM) distinta de aquella en la que se encuentra el EJB al que está accediendo.
- Puede ser un componente Web, una aplicación cliente J2EE o también otro EJB.

Para crear un EJB con un acceso remoto es necesario codificar una interfaz remota y una interfaz *home*. La interfaz remota define los métodos de negocio que podrán ser respondidos por las instancias del EJB. La interfaz *home* proporciona los métodos necesarios para gestionar el ciclo de vida de las instancias del EJB, como *create* y *remove*. Para los EJB de entidad, la interfaz *home* también define métodos de búsqueda y métodos *home* (equivalentes a los métodos de clase en Java). Los métodos de búsqueda se usan para localizar instancias del EJB de entidad a partir de su clave primaria. Los métodos *home* son métodos de negocio cuya invocación afecta a todas las instancias de un EJB.

4.3.2 Acceso local

Un cliente local tiene las siguientes características:

- Debe correr en la misma JVM que el EJB al que está accediendo.
- Puede ser un componente Web o también otro EJB.
- Para el cliente local, la localización del EJB al que está accediendo no es transparente.
- A menudo el cliente local es otro EJB de entidad, que tiene una relación gestionada por el contenedor con otro EJB de entidad.

Al igual que en el acceso remoto, para construir un EJB que permita acceso local se debe codificar la interfaz local y la interfaz *home* local. La interfaz local define los métodos de negocio del EJB y la interfaz *home* local los métodos que gestionan el ciclo de vida de las instancias del EJB.

4.4 Ventajas de los EJB

4.4.1 Simplicidad. Debido a que el contenedor de aplicaciones libera al programador de realizar las tareas de bajo nivel, la escritura de un EJB es casi tan sencilla como la escritura de una clase Java. Como resultado, el desarrollador de aplicaciones se concentra en la lógica de negocio y en el dominio específico de la aplicación.

4.4.2 Portabilidad de la aplicación. Una aplicación EJB puede ser instalada y ejecutada en cualquier servidor de aplicaciones compatible con las especificaciones de J2EE.

4.4.3 Reutilización de componentes. Una aplicación EJB está formada por componentes. Cada componente es un bloque de construcción reutilizable. Un EJB desarrollado puede ejecutarse en distintas aplicaciones, adaptando sus características a las necesidades de las mismas. También un EJB que se esta ejecutando puede ser usado por múltiples aplicaciones cliente simultáneamente.

4.4.4 Construcción de aplicaciones complejas. La arquitectura EJB simplifica la construcción de aplicaciones complejas. Al estar basada en componentes y en un conjunto claro y bien establecido de interfaces, se facilita el desarrollo en equipo.

4.4.5 Separación de Lógica de presentación y Lógica de negocio. Un EJB encapsula por lo general un proceso o una entidad de negocio (un objeto que representa datos del negocio), haciéndolo independiente de la lógica de presentación. El programador de EJB no necesita preocuparse de cómo formatear la salida; será el programador que desarrolle la interfaz Web el que se ocupe de ello, usando los datos de salida que proporcionará el EJB. Esta separación hace

posible desarrollar distintas lógicas de presentación para la misma lógica de negocio o cambiar la lógica de presentación sin modificar el código que implementa el proceso de negocio.

4.4.6 Implantación en muchos entornos operativos. Entendemos por entornos operativos el conjunto de aplicaciones y sistemas (bases de datos, sistemas operativos, aplicaciones ya en marcha, etc.) que están instaladas en una empresa. Al detallarse claramente todas las posibilidades de implantación de las aplicaciones, se facilita el desarrollo de herramientas que asistan y automaticen este proceso. La arquitectura permite que los EJB de entidad se conecten a distintos tipos de sistemas de bases de datos.

4.4.7 Arquitectura distribuida. La arquitectura EJB hace posible que las aplicaciones se ejecuten de forma distribuida entre distintos servidores de una red, haciendo uso de RMI-IIOP.

4.4.8 Interoperabilidad entre aplicaciones. La arquitectura EJB hace más fácil la integración de múltiples aplicaciones de diferentes vendedores. La interfaz del EJB con el cliente sirve como un punto bien definido de integración entre aplicaciones.

4.4.9 Integración con sistemas no Java. Las APIs relacionadas, como las especificaciones Connector y Java Message Service (JMS), así como los EJB dirigidos por mensajes, hacen posible la integración de los EJB con sistemas no Java, como sistemas ERP (Enterprise Resources Planning).

4.4.10 Recursos educativos y herramientas de desarrollo. El hecho de que la especificación EJB sea un estándar hace que exista una creciente oferta de herramientas y formación, que facilita el trabajo del desarrollador de aplicaciones EJB.

4.4.11 Otras ventajas. Entre las ventajas que aporta esta arquitectura al cliente final, se destaca la posibilidad de elección del servidor, la mejora en la gestión de las aplicaciones, la integración con las aplicaciones y datos ya existentes y la seguridad.

4.4.11.1 Elección del servidor. Debido a que las aplicaciones EJB pueden ser ejecutadas en cualquier servidor J2EE, el cliente no queda ligado a un vendedor de servidores.

4.4.11.2 Gestión de las aplicaciones. Las aplicaciones son mucho más sencillas de manejar (iniciar, parar, configurar, etc.) debido a que existen herramientas de control más elaboradas.

4.4.11.3 Integración con aplicaciones y datos ya existentes. La arquitectura EJB y otras APIs de J2EE simplifican y estandarizan la integración de aplicaciones EJB con aplicaciones no Java y sistemas en el entorno operativo del cliente. Por ejemplo, un cliente no tiene que cambiar un esquema de base de datos para encajar en una aplicación. En lugar de ello, se puede construir una aplicación EJB que encaje en el esquema.

5. JSP y XML

5.1 Introducción

XML, Extensible Markup Language o Lenguaje de Marcado Extensible se ha convertido en un estándar para representación de datos y documentos en Internet gracias a su portabilidad, independencia de la plataforma y su fácil almacenamiento en archivos con formato texto. XML es la solución perfecta para comunicar sistemas heterogéneos dentro de una o más empresas. La característica de independencia también es proporcionada por Java, por lo cual ambas forman un conjunto ideal para la construcción de aplicaciones Web, teniendo como resultado de esta fusión datos y código fuente portables.

En XML la información es expresada bajo una estructura, su formato de almacenamiento basado en texto puede ser fácilmente transmitido, transformado e interpretado por entidades que entiendan la estructura del documento.

Existen diversos analizadores de XML escritos en lenguaje Java, que forman una completa colección de APIs que han sido construidas para el desarrollo de aplicaciones basadas en XML. Las páginas JSP pueden hacer uso de estas APIs para generar y procesar documentos XML.

5.2 Breve introducción a XML

XML es un metalenguaje que sirve para componer documentos que tienen una estructura de datos definida. Las características y beneficios de XML pueden ser agrupados en las siguientes áreas:

- Extensible. Como metalenguaje permite crear sus propios lenguajes de marcado.
- Estructura precisa. Todo documento XML tiene una estructura bien definida. Cada documento posee un elemento raíz que anida a otros elementos en forma jerárquica.
- Existen dos tipos de documentos:
    1. Documentos válidos. Un documento XML válido esta definido mediante un archivo de Definición de Tipos de Datos (DTD), el cual especifica la gramática para tipos de elementos, atributos y entidades que pueden formar parte del documento. Los DTD hacen que los documentos XML sean portables, ya que en éste se definen reglas de cómo procesar el documento.
    2. Documentos bien formados. Un documento bien formado no necesita estar ligado a un DTD. Estos deben cumplir con las siguientes reglas:
        - Cada elemento debe estar entre una etiqueta de apertura y cierre.
        - Debe existir un único elemento raíz que contenga los demás elementos.
        - Extensiones. Es posible utilizar Extensible Stylesheet Languaje (XSL) para definir la forma o estilo de presentación de un documento XML.

5.2.1 XML versus HTML

Tanto XML como HTML son lenguajes de marcado, en donde los tags o etiquetas son utilizadas para anotar datos. Las principales diferencias entre estos dos lenguajes son:

- HTML por si sólo puede ser utilizado para crear documentos visibles, ya que incorpora tanto el contenido como su presentación. XML permite definir la sintaxis de un documento.
- Los documentos HTML no son bien formados ya que no todas las etiquetas están cerradas, es decir, no siempre existe una etiqueta de apertura y otra de cierre, por ejemplo: </BR> para salto de línea. Los documentos XML deben ser siempre bien formados.
- En XML los nombres de las etiquetas son sensibles a las letras mayúsculas y minúsculas, pero no en HTML, por ejemplo en HTML <*Title*> es lo mismo que <*TITLE*>.

5.3 Utilizando XML con JSP

La tecnología JSP proporciona variadas características que son ideales para trabajar con XML. Las páginas JSP pueden contener cualquier tipo de datos basados en texto, de esta manera se puede fácilmente generar documentos XML. También las páginas JSP pueden utilizar el poder de la plataforma Java para analizar y transformar mensajes y documentos XML, mediante APIs que permitan trabajar con XML.

Las páginas JSP pueden:

- Consumir documentos XML
- Transformar documentos XML
- Generar documentos XML

5.3.1 Utilizando fuentes de datos XML en JSP

La figura 8 muestra cómo una página JSP se conecta a fuentes de datos heterogéneos a través de objetos Bean o etiquetas personalizadas.

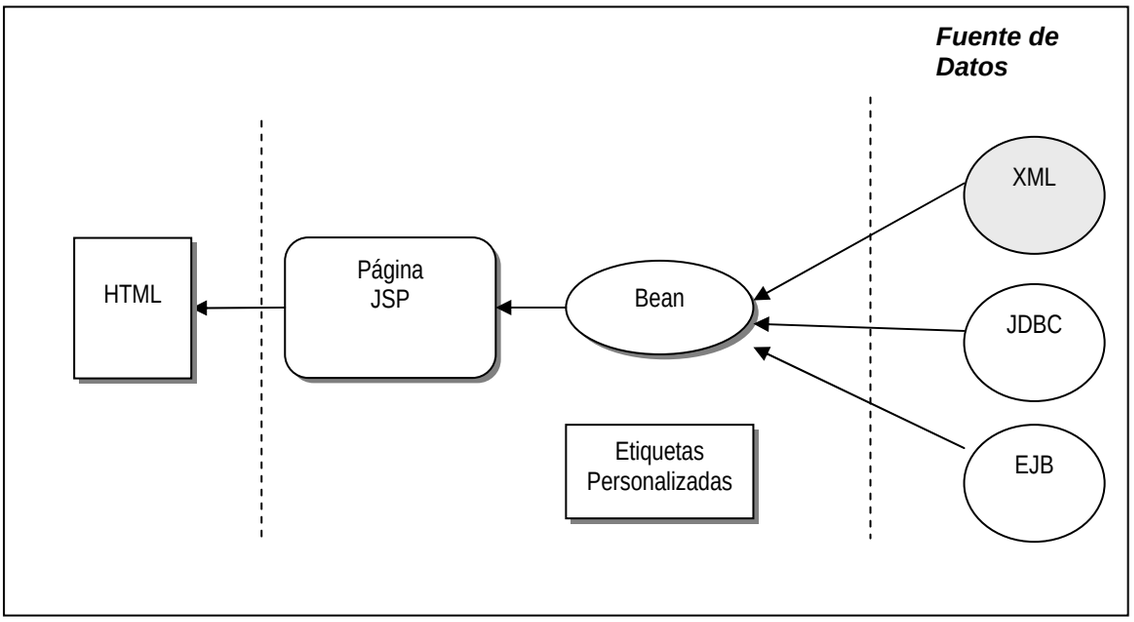


Fig.8 Fuentes de datos en JSP

Una forma de utilizar como fuente de datos un documento XML es creando objetos que representen la información almacenada en el documento XML. Para esto es necesario crear clases Java (JavaBeans) que hagan uso de las API SAX o DOM y encapsular los datos en éstos componentes. La otra opción es hacer uso de etiquetas personalizadas que permiten acceder al contenido almacenado en el archivo XML. Estas etiquetas son utilizadas directamente desde las páginas JSP.

5.3.2 Conversión de XML utilizando la transformación XSLT

Otra forma de utilizar documentos XML en páginas JSP es aplicando una transformación sobre la fuente de datos XML, con el fin de extraer los datos o crear un nuevo formato. Esta transformación puede ser llevada a cabo utilizando diferentes mecanismos y accediendo a los datos a través de etiquetas personalizadas.

XSLT es un lenguaje de transformación estandarizado en W3C que se utiliza para transformar datos XML en formatos HTML, PDF u otros formatos XML. Para transformar datos XML en formato HTML se pueden utilizar hojas de estilo XLS.

5.3.2 Generación de XML utilizando JSP

Anteriormente se mencionó que las páginas JSP pueden utilizar XML como fuente de datos para generar contenido dinámico. Las páginas JSP también pueden generar datos en formato XML, estos datos pueden ser extraídos desde otros documentos XML e incluso desde bases de datos. La figura 9 ilustra lo descrito anteriormente.

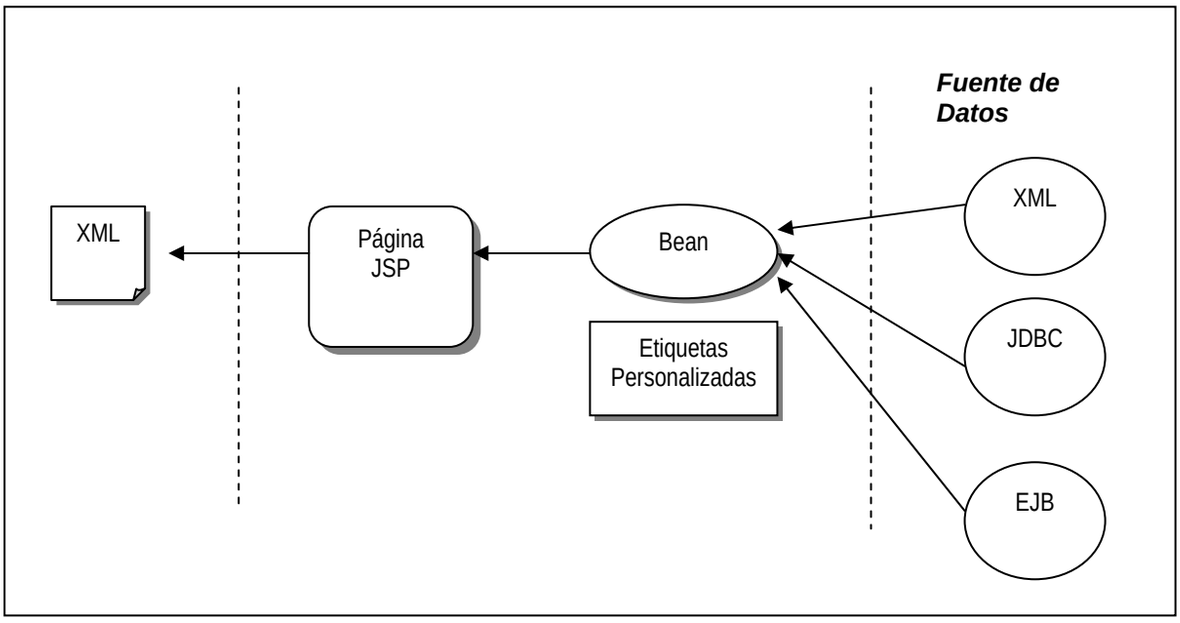


Fig.9 Generación de XML desde JSP

5.4 APIs de Java para XML

Las APIs de Java para XML permiten implementar aplicaciones Web que trabajen con XML. Estas APIs se agrupan en dos categorías: Las que tratan directamente con el documento XML y aquellas que tratan con procedimientos [URL5].

- Orientadas al documento
  - JAXP. API Java para Procesar XML, procesa documentos XML utilizando varios analizadores.
  - JAXB. Arquitectura Java para Uniones XML, mapea elementos XML a clases Java.
- Orientadas al Procedimiento
  - JAXM. API Java para Mensajería XML, envía mensajes SOAP sobre Internet de una forma estándar.
  - JAXR. API Java para Registros XML, proporciona una forma estándar para acceder a registros de negocios que comparten información.
  - JAX-RPC. API Java para RPC basado en XML, envía llamadas a métodos SOAP a partes remotas sobre Internet y recibe los resultados.

La característica más importante de las APIs de Java para XML es que todas cumplen con los estándares de la industria, así aseguran la interoperabilidad con otras tecnologías y sistemas. Varios grupos de estandarización, como el "World Wide Web Consortium" (W3C) y la "Organization for the Advancement of Structured Information Standards" (OASIS), han estado definiendo formas estándar de cómo hacer las cosas, para que las empresas se basen estos estándares y puedan hacer que sus datos y aplicaciones funcionen entre ellas.

Los APIs Java para XML definen requerimientos de compatibilidad estricta para asegurarse que todas las implementaciones siguen la funcionalidad estándar, pero también le dan a los desarrolladores una gran libertad para proporcionar implementaciones hechas a medida para usos específicos.

Capítulo IV

Un Proceso para el Desarrollo de Aplicaciones Web

1. Introducción

El siguiente capítulo propone una metodología de desarrollo de aplicaciones Web basada en técnicas de Análisis y Diseño Orientado a Objetos (ADOO) para construir, desde los requisitos al diseño, el núcleo de una aplicación: El Modelo de Negocio. Una vez definidas las clases que implementarán la lógica de negocio, la metodología hace uso del patrón arquitectónico MVC o Modelo-Vista-Controlador, con el fin de obtener aplicaciones Web escalables y fáciles de mantener.

Una de las actividades consideradas dentro del proceso de desarrollo, es el uso de los *Casos de Prueba*, los cuales permiten asegurar que durante todo el proceso de construcción se realicen pruebas funcionales a las clases que implementan la lógica de la aplicación.

Una de las principales características de esta metodología es que se basa en un modelo conceptual derivado del dominio del problema. De esta forma el proceso parte realizando una documentación y análisis detallado de los requerimientos, utilizando para este fin una técnica sistemática e intuitiva basada en casos de uso, para extraer luego los conceptos presentes y sus asociaciones. Tomando como base estos conceptos se construye el conjunto de clases que implementarán la lógica de negocio de la aplicación. La lógica de negocio es utilizada posteriormente para construir una solución codificable mediante el patrón arquitectónico MVC sobre la plataforma J2EE.

El proceso propuesto es iterativo e incremental. Tiene como estrategia abordar el producto software en pequeños pasos manejables o mini proyectos, formando cada uno de estos una iteración. Cada iteración produce resultados tangibles que aportan un incremento al producto final. En proyectos de desarrollo de sistemas complejos es imposible abordar en forma secuencial

el problema, diseñar la solución, construirla y probarla, por lo cual es necesaria una metodología iterativa para comprender el problema a través de refinamientos sucesivos e incrementales.

En cada iteración se utilizan y construyen artefactos. Artefacto es el término general con el cual se hace referencia a cualquier tipo de información tangible creada, intercambiada o utilizada por los involucrados en el proyecto de desarrollo. Ejemplos de artefactos pueden ser diagramas UML y textos asociados a ellos o prototipos de interfaz usuario.

La figura 10 ilustra la composición en iteraciones y flujos de trabajo del ciclo de vida del proceso de desarrollo propuesto en este trabajo.

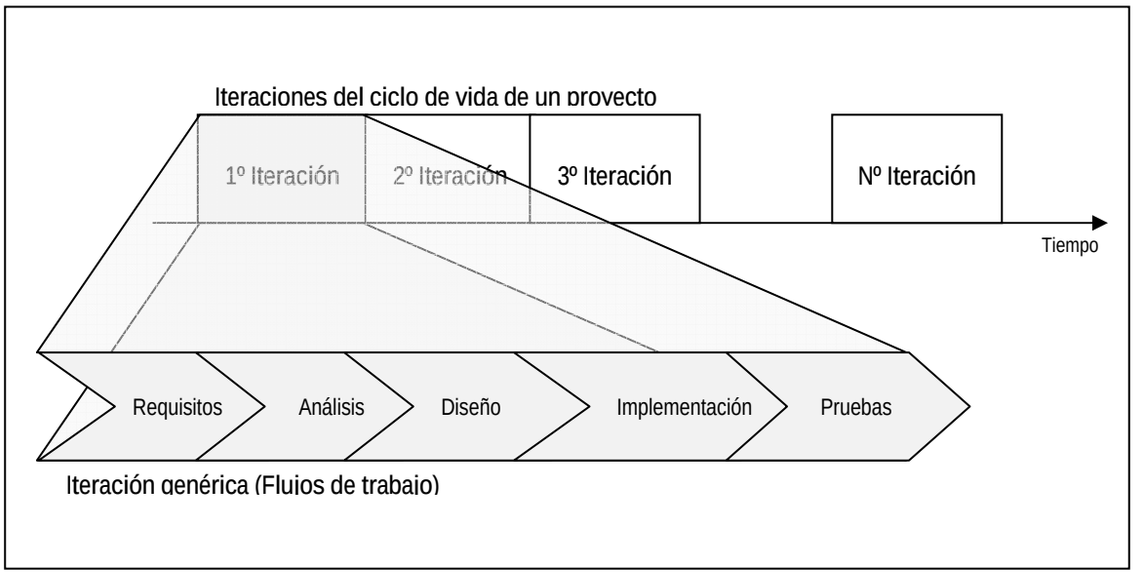


Fig.10 Ciclo de vida del proceso de desarrollo

2.2 Captura de requisitos

2.2.1 Introducción

El propósito fundamental del flujo de trabajo de captura de los requisitos es guiar el desarrollo hacia un sistema correcto. Lo anterior se alcanza desarrollando una descripción de los requisitos del sistema, es decir las condiciones y capacidades que el sistema debe cumplir. La descripción debe ser suficientemente buena como para llegar a un acuerdo entre el cliente (incluyendo los usuarios) y los desarrolladores. El conjunto de requerimientos del sistema a construir debe ser expresado de tal forma que el cliente pueda leerlos y comprenderlos. Para lograr esto se debe utilizar el lenguaje del cliente en la descripción de los requerimientos.

Una técnica muy útil para identificar los requisitos es mediante los casos de uso, los cuales permiten documentar los requisitos funcionales del sistema en términos de secuencias de eventos que los actores realizan al utilizarlo. Los requisitos no funcionales, los cuales no son definidos en los casos de uso, deben ser especificados por separado en una lista de atributos del sistema, ejemplos de esto son: rendimiento, facilidad de uso y fiabilidad.

Dado que los requisitos cambian todo el tiempo, es necesario actualizar de manera controlada cada uno de los artefactos que se construyen, ya que cada iteración reflejará algún cambio o refinamiento en el conjunto de requisitos y artefactos creados.

Una vez obtenida la primera aproximación de los requerimientos utilizando los casos de uso y su descripción, éstos deben ser clasificados con el fin de abordar primero los más significativos para la aplicación. Por lo general, también se suele abordar primero las

funcionalidades que representan un alto nivel de riesgo para el proyecto (uso de nuevas tecnologías, requerimientos funcionales y no funcionales), de esta forma el proceso se asegura de minimizar el riesgo el las primeras iteraciones y no al final de proyecto.

Por último, no basta con tener una visión aislada de cada caso de uso. Es necesario representar en un único diagrama todos los casos de uso y cómo estos se relacionan.

2.2.2 Artefactos

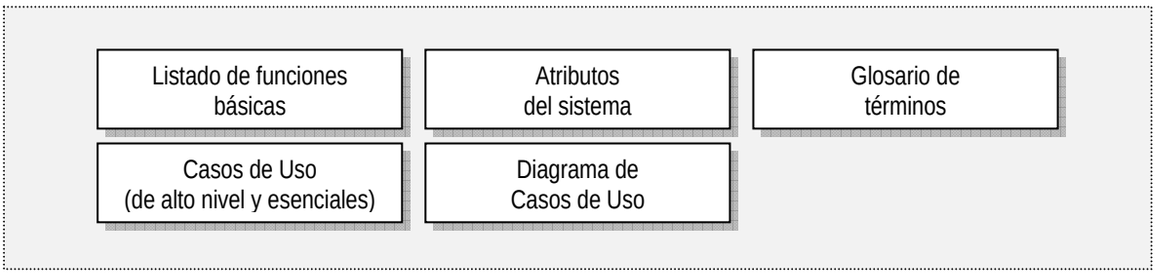


Fig.11 Artefactos de Captura de Requisitos

2.2.3 Definición del flujo de trabajo “Captura de los Requisitos”

2.2.3.1 Funcionalidades básicas del sistema

El proceso de captura de requisitos comienza creando un listado de las ***funcionalidades básicas del sistema***. Estas funcionalidades deben ser capturadas por analistas de sistemas y jefes de proyectos en sesiones con los futuros usuarios del sistema y clientes.

Se debe utilizar la tabla 5 para listar las funcionalidades básicas del sistema:

Ref. #	Funcionalidad	Categoría
R1.1	Descripción de la funcionalidad	Evidente/Oculta/Opcional
…		Evidente/Oculta/Opcional
Rn.m		Evidente/Oculta/Opcional

Tabla 5: Funcionalidades básicas del sistema.

La tabla permite además clasificarlos en categorías evidente, oculta y opcional. La categoría Evidente corresponde a funcionalidades visibles y conocidas por el usuario, como por ejemplo efectuar pago a través de tarjeta de crédito. La categoría Oculta corresponde a funcionalidades desconocidas por el usuario, como lo son, métodos en almacenamiento en medios persistentes o transacciones. La categoría Opcional hace alusión a funcionalidades optativas de implementar, dependiendo si estas no afectan significativamente el costo o a otras funcionalidades.

Al momento de definir las funcionalidades básicas es posible encontrase con requerimientos no funcionales o atributos del sistema. Estos atributos son características o dimensiones del sistema y deben ser documentados utilizando la tabla 6:

Atributo	Detalles	Aplica a funcionalidad
Nombre del atributo	Valores o descripción	Ref. funcionalidad

Tabla 6: Atributos del sistema.

Los atributos del sistema pueden ser aplicados, en algunas ocasiones, a funcionalidades básicas del sistema, si es así, estos deben ser asociados a dichas funcionalidades. Ejemplos de atributos pueden ser: facilidad de uso, plataformas, tiempo de respuesta, etc.

2.2.3.2 Glosario de términos

Junto con describir las funcionalidades del sistema se debe desarrollar el glosario de términos, el cual constituye un documento simple que lista y define todos los términos que requieren clarificación para mejorar la comunicación entre los participantes del proyecto, disminuyendo de esta forma el riesgo por confusión.

2.2.3.3 Casos de uso

Una vez descritas las funcionalidades básicas del sistema se deben identificar y describir los casos de uso. Los casos de uso representan una excelente técnica para entender y organizar los requerimientos. Un caso de uso es un documento narrativo que describe la secuencia de eventos de un actor (o agente externo) que utiliza el sistema para completar un proceso [Jac, 92]. Un caso de uso involucra la descripción de varios pasos dentro de un proceso, desde su inicio a fin, no describe pasos o actividades aisladas. Existen dos modalidades para describir un caso de uso:

1. Descripción de alto nivel de un caso de uso (útil para requerimientos iniciales)
2. Descripción expandida de un caso de uso

A continuación se presentan las plantillas para describir los dos tipos de modalidades:

Descripción de alto nivel de un caso de uso	
Caso de uso:	Nombre del caso de uso
Actores:	Lista de actores, se debe indicar quién inicia el caso de uso
Tipo:	1. Primario, secundario u opcional - 2. Esencial o real
Descripción:	Descripción del curso de eventos o interacción entre el actor y el sistema

Descripción expandida de un caso de uso	
Caso de uso:	Nombre del caso de uso
Actores:	Lista de actores, se debe indicar quién inicia el caso de uso
Propósito:	Intención del caso de uso
Resumen:	Descripción del curso de eventos o interacción entre el actor y el sistema
Tipo:	1. Primario, secundario u opcional 2. Esencial o real
Referencias cruzadas:	Relacionada con otros casos de uso y funcionalidades del sistema

Curso de eventos típico

Acciones del actor	Respuesta del Sistema
1.- Este caso de uso comienza cuando…	

Cursos alternativos

- Línea N: …

Tipo de caso de uso		Descripción
1	Caso de uso Primario.	Representa el proceso más importante
	Caso de uso Secundario	Representa proceso menos significativos o que se realizan con menor frecuencia.
	Caso de uso Opcional	Representa procesos que pueden no ser realizados.
2	Caso de uso esencial	Su descripción es muy abstracta y no esta descrita en detalle. Este tipo de caso de uso se utiliza por lo general al comenzar un proyecto, en la descripción de alto nivel, no esta descrito en términos de tecnología.
	Caso de uso real	Describe en detalle la realización de un caso de uso. Por lo general, este tipo de casos de uso se utilizan en la fase de diseño y desarrollo dado que incluyen en su descripción términos de tecnología que se ha de utilizar.

Tabla 7: Tipos de casos de uso.

Un método útil para identificar los casos de uso es:

1. Identificar los actores relacionados con el sistema u organización.
2. Para cada uno de los actores, identificar los procesos que ellos inician o en los cuales participan.

Un actor es una entidad externa al sistema (usuarios, otros sistemas o dispositivos), los cuales de alguna manera interactúan con el sistema. La descripción de cómo interactúan se encuentra descrita en los casos de uso [Lar, 98].

Una vez definido los casos de uso en forma individual y aislada, estos deben ser representados en un diagrama de casos de uso. Un diagrama de casos de uso ilustra un conjunto de casos de uso, actores y relaciones entre los actores y casos de uso [Lar, 98]. El diagrama de

casos de uso se representa mediante óvalos, actores y líneas. Las líneas ilustran la comunicación entre actores y casos de uso (flujos de información o estímulos).

La figura 12 muestra un diagrama de casos de uso compuesto por los casos de uso: Comprar producto, Devolver producto e Iniciar sesión en los que se puede observar la participación de los actores Vendedor y Cliente.

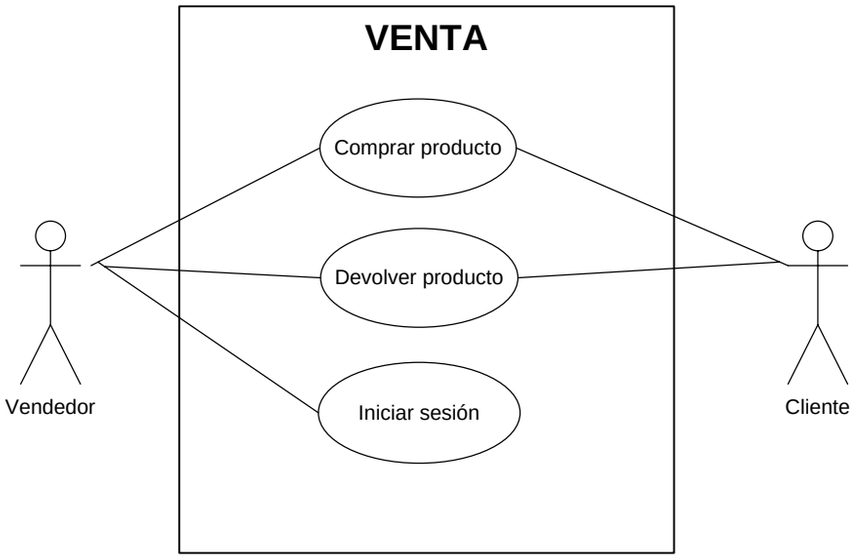


Fig.12 Ejemplo de Diagrama de Casos de Uso

2.2.3.4 Priorización de los casos de uso y planificación de iteraciones

Una vez definida la especificación de los casos de uso estos deben ser clasificados con el fin de planificar qué casos de uso se abordaran primero.

Se evalúa cada una de las siguientes características para cada caso de uso:

a. Posee un impacto significativo sobre el diseño arquitectónico.
b. Contiene información significativa.
c. Incluye alto nivel de riesgo, tiempo o funciones complejas.
d. Involucra investigación, o uso de tecnología que no es dominada.

e. Representa un proceso significativo del negocio.

f. Incrementa los ingresos o disminuye los costos.

La evaluación de cada una de estas características nos entregará la clasificación en la que cae el caso de uso: Alto, medio o bajo.

Se debe utilizar la tabla 8 y asignar un valor a cada característica: (ejemplo de 0 a 5)

Caso de uso	A	B	C	D	E	F	Suma	Clasificación
Nombre del caso de uso	0	1	5	2	0	4	12	Alto

Tabla 8: Clasificación de casos de uso

Una vez evaluado cada caso de uso con su respectivo resultado (suma), los que obtengan valores mayores corresponderán a clasificación “Alta”, los menores a clasificación “baja” y los restantes a “Media”.

Se debe comenzar la primera iteración con el o los casos de uso que tengan una clasificación “Alta”.

En algunas ocasiones, se puede utilizar más de un ciclo de desarrollo para un caso de uso, esto ocurre cuando el caso de uso requiere demasiado tiempo para ser completado por ser muy complejo. Si es así, es recomendable redefinir el caso de uso en varios casos de uso, uno por cada ciclo, los cuales, al finalizar un ciclo de desarrollo entregarán funcionalidades incrementales hasta completar el caso de uso original [Lar, 98].

La figura 13 muestra un ejemplo de localización de los casos de uso X e Y en los ciclos de desarrollo.

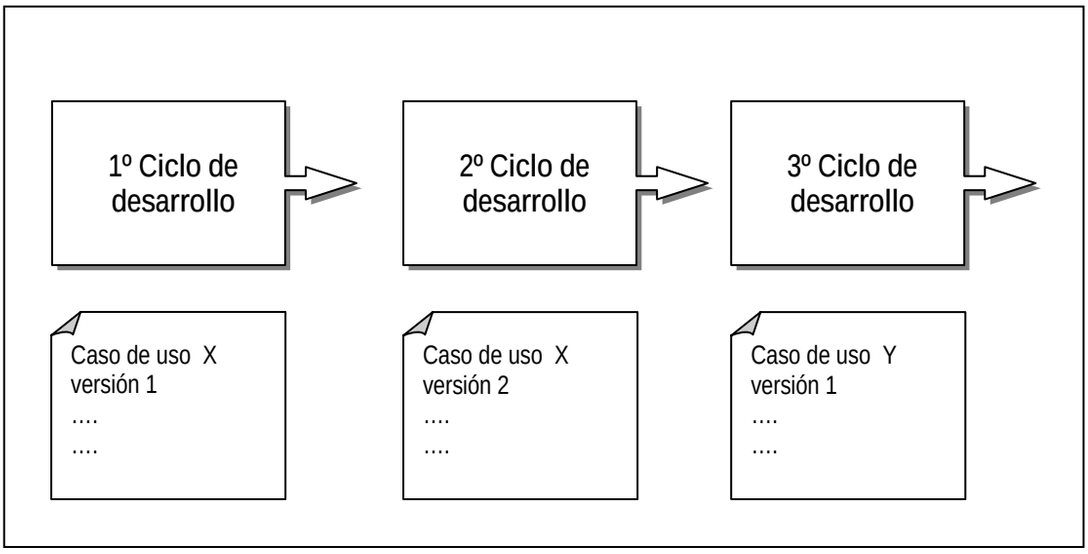


Fig.13 Ejemplo de localización de casos de uso en ciclos de desarrollo.

Cada ciclo de desarrollo debe tener definido el tiempo que se demorará en completarse, por ejemplo: 2 a 3 semanas o 2 meses. Se puede hacer uso de cartas Gantt para planificar los ciclos de desarrollo.

Existen técnicas útiles para estimar el tiempo y los recursos utilizados en el desarrollo de cada caso de uso o ciclo de desarrollo, tales como COCOMO o COCOMO II. Estas técnicas de estimación están fuera del alcance de este proyecto. Vale la pena mencionar que al momento de estimar los tiempos y recursos utilizados en el desarrollo, estos no deben ser calculados únicamente en base a los casos de uso ya que existen otras actividades de desarrollo y gestión que no están contemplados en estos artefactos, como por ejemplo: documentación, administración, investigación, preparación del ambiente de desarrollo, reclutamiento de personal y adquisiciones, pruebas, implantación o capacitación.

Una vez desarrollado y aceptado el modelo de requisitos se comienza el desarrollo del modelo de análisis.

2.3 Fase de análisis

2.3.1 Introducción

El objetivo de la fase de análisis es identificar los conceptos involucrados en el ciclo de desarrollo actual y desarrollar un modelo conceptual inicial. Un modelo conceptual es una representación o descripción estática de conceptos en el dominio del problema. El modelo conceptual debe representar:

- Conceptos
- Asociaciones entre conceptos
- Atributos de los conceptos

En el análisis no se especifican los métodos de los conceptos ni elementos de software, ya que estos son definidos posteriormente en el diseño.

En el análisis se refina el glosario de términos. El glosario debe ser creado inicialmente en la captura de requerimientos, pero debe ser refinado y mantenido durante el análisis.

En esta fase se debe definir además el comportamiento dinámico del sistema mediante el uso del diagrama de secuencias, diagrama de estados y los contratos.

El diagrama de secuencias describe los eventos entre el actor y el sistema, estos eventos deben ser extraídos desde el flujo normal de eventos descritos en los casos de uso. Los contratos son documentos ayudan a definir el comportamiento del sistema, describen el efecto de las operaciones sobre el estado del sistema.

En la fase de análisis, los esfuerzos se centran en comprender y delimitar el problema en estudio, no en la definición de la solución.

2.3.2 Artefactos

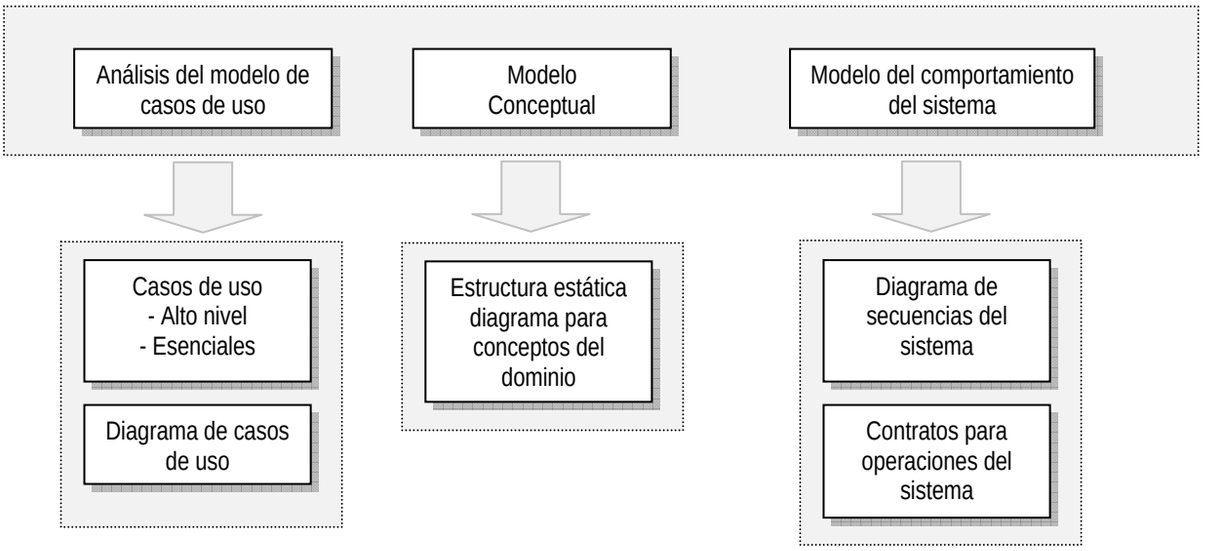


Fig.14 Artefactos para la Fase de Análisis

2.2.3 Definición del flujo de trabajo “Análisis”

Asumiendo que la fase de requerimientos ha sido completada y que los casos de uso han sido identificados, clasificados y planificados podemos iniciar la fase de análisis.

2.2.3.1 Modelo Conceptual

La fase de análisis comienza con la construcción del modelo conceptual, el cual ilustra los conceptos más importantes en el dominio del problema. En UML, el modelo conceptual es representado mediante un diagrama de estructura estática del sistema, en el cual no se definen operaciones. En la definición del modelo conceptual se debe identifica en primer lugar los conceptos, luego las relaciones entre los conceptos y finalmente los atributos de cada concepto.

Por lo general los conceptos se presentan como sustantivos en la descripción narrativa de los casos de uso. Una estrategia simple para verificar si nuestro supuesto concepto es correcto es verificando si cae en alguna de las siguientes categorías:

1. Objetos físicos o tangibles
2. Especificación, diseño o descripción de cosas
3. Lugares
4. Transacciones
5. Líneas ítem de una transacción
6. Roles de personas
7. Contenedor de otras cosas
8. Cosas en un contenedor
9. Otros computadores o sistemas externos al sistema
10. Organizaciones
11. Eventos
12. Procesos
13. Reglas y políticas
14. Catálogos
15. Registros financieros, trabajos, contratos
16. Servicios e instrumentos financieros
17. Manuales, libros

Una vez identificado los conceptos del dominio del problema, se deben seguir los siguientes pasos para crear el modelo conceptual [Lar, 98]:

1. Listar los conceptos candidatos obtenidos mediante la lista de categoría de conceptos y sustantivos en frases de descripción de casos de uso.
2. Dibujar en un modelo conceptual los conceptos identificados.

3. Agregar las asociaciones necesarias para registrar las relaciones entre conceptos.
4. Agregar los atributos de conceptos necesarios para satisfacer las necesidades de la información.

Se debe tener especial cuidado al crear el modelo conceptual en no confundir atributos y conceptos. Si no se piensa en un concepto X como si fuese un número o texto en el mundo real, probablemente X es un concepto y no un atributo.

La figura 15 muestra un ejemplo de asociación entre los objetos Venta e Item, de la cual se entiende que un concepto Venta contiene uno o más Ítem.

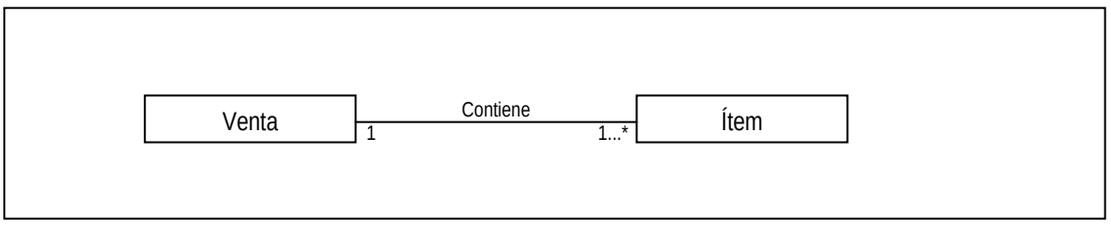


Fig.15 Ejemplo de conceptos y asociaciones

Las asociaciones se utilizan para describir relaciones entre conceptos que necesitan ser preservadas por algún momento y son descritas mediante una línea entre dos conceptos y su descripción. Las asociaciones son intrínsicamente bidireccionales. El inicio y final de la línea describe la multiplicidad de la relación.

Se debe utilizar la siguiente lista de categorías de asociaciones para identificar las asociaciones dentro del modelo conceptual:

1. A es parte física de B
2. A es parte lógica de B
3. A se encuentra físicamente contenido en B
4. A se encuentra lógicamente contenido en B

5. A es una descripción de B
6. A es una línea de ítem de B
7. A es conocido, capturado o registrado por B
8. A es miembro de B
9. A es una sub-unidad organizacional de B
10. A utiliza o administra a B
11. A se comunica con B
12. A esta relacionado con una transacción B
13. A es una transacción relacionada con otra transacción B
14. A esta a continuación de B
15. A es poseído por B

Es más relevante identificar los conceptos que sus asociaciones, por consiguiente no se debe destinar demasiado tiempo ni esfuerzo en la búsqueda de asociaciones entre conceptos ya que sus beneficios son marginales. Muchas de las asociaciones identificadas en el modelo conceptual, durante la fase de análisis, serán finalmente implementadas como software ya sea como rutas de navegación o visibilidad (esto será explicado más adelante), las asociaciones restantes definitivamente no serán implementadas. Análogamente, durante la fase de diseño, pueden descubrirse asociaciones que necesitan ser implementadas y que no se encuentren presentes en el modelo conceptual, de ser así, el modelo conceptual debe ser actualizado, incorporando las nuevas asociaciones.

Otro factor a considerar en la búsqueda de asociaciones es que un modelo conceptual sobrecargado de asociaciones tiende a confundir más que a ayudar.

La figura 16 muestra los diferentes tipos de multiplicidad de asociaciones entre conceptos:

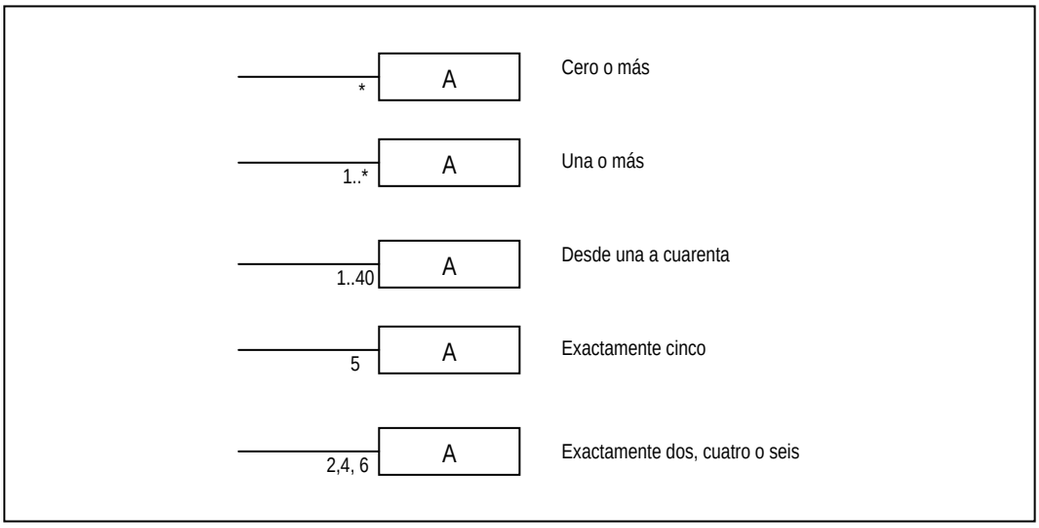


Fig.16 Multiplicidad de asociaciones entre conceptos

Los **atributos** de conceptos constituyen valores lógicos que necesitan ser identificados para satisfacer los requerimientos de información. Los atributos, en un modelo conceptual, deben ser preferiblemente atributos o valores simples como por ejemplo: fechas, números, textos, color, etc. Un atributo no debería ser un concepto complejo del dominio del problema.

Un criterio para identificar atributos útiles es:

- Aquellos en que los requerimientos indican o conllevan la necesidad de conservar información.
- Deben ser seleccionados exclusivamente en base a los requerimientos, no agregar cosas que no existan.

2.2.3.2 Diagramas de secuencia

Una vez construida la primera aproximación del modelo conceptual, con sus conceptos, asociaciones y atributos, el siguiente paso es modelar el comportamiento del sistema mediante el diagrama de secuencia del sistema.

Los diagramas de secuencias ilustran eventos que el actor realiza con el sistema, por lo cual su construcción es considerada como parte de la investigación del sistema que se ha de construir, por consiguiente es incluido dentro del modelo de análisis. El comportamiento del sistema debe ser una descripción de lo que el sistema debe hacer, sin profundizar en cómo lo hará.

El diagrama de secuencia del sistema corresponde a un modelo del comportamiento del sistema que representa, en un determinado escenario, la secuencia de eventos generados por actores externos al sistema (como caja negra). La secuencia de eventos corresponde al curso de eventos típicos de un caso de uso y otros cursos de interés.

La figura 17 ilustra un diagrama de secuencia de sistema para el caso de uso Comprar producto:

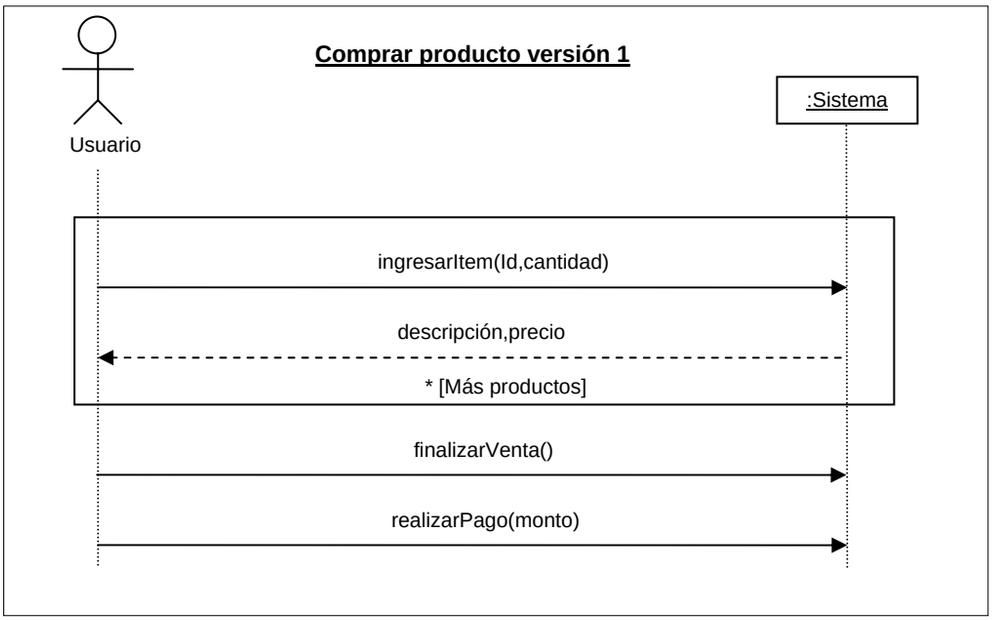


Fig.17 Ejemplo de Diagrama de Secuencia de Sistema

En el ejemplo, el tiempo precede desde arriba hacia abajo y el orden de los eventos deben ser los mismos que especifican los casos de uso en su curso de eventos típicos. Los eventos pueden incluir parámetros. En este caso, sólo se muestra un actor, en otras situaciones se deben incluir todos los actores que participan en el caso de uso.

En el ejemplo, el rectángulo que encierra el evento *ingresarItem*, ilustra que el evento se puede repetir muchas veces con la finalidad de ingresar más productos.

Un evento de un sistema corresponde a un estimulo externo generado por un actor a un sistema. Una operación de sistema es una operación que el sistema ejecuta como respuesta al evento recibido. Por ejemplo cuando el Usuario genera el evento *ingresarItem,* este causa la ejecución de la operación del sistema *ingresarItem,* los nombres de la operación y el evento son

los mismos, la diferencia es que el evento es el nombre del estimulo y la operación es la respuesta.

La identificación de eventos genera como resultado el conjunto de operaciones que se requiere que el sistema realice. En UML pueden agruparse estas operaciones como Operaciones de tipo Sistema, los parámetros son opcionales, la figura 18 muestra, mediante UML, las operaciones del sistema para el caso de uso Comprar producto:

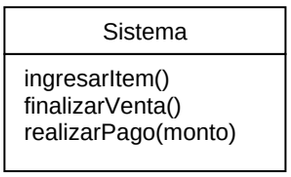


Fig.18 Ejemplo de Operaciones de Sistema

Los pasos para crear un diagrama de secuencias, a partir del curso típico de eventos para un caso de uso es el siguiente [Lar, 98]:

1. Dibujar una línea representando el sistema como una caja negra.
2. Identificar cada actor que opera o interactúa directamente sobre el sistema. Dibujar una línea para cada actor.
3. A partir de la descripción del curso de eventos típicos de un caso de uso, identificar los eventos que cada actor genera sobre el sistema. Ilustrar estos sobre el diagrama.

Una vez definidos los diagramas de secuencia para los casos de usos presentes en la iteración, se crear los contratos para las operaciones del sistema.

2.2.3.3 Contratos de operaciones del sistema

Los contratos son documentos útiles que ayudan a definir el comportamiento del sistema ya que estos describen los efectos de una operación sobre el sistema. Un contrato define qué pasará después de que el sistema realiza una operación, sin describir cómo este la lleva a cabo. El contrato es expresado en términos de precondiciones y poscondiciones.

El formato del documento utilizado para describir los contratos es el siguiente:

Contrato	
Nombre:	Nombre de la operación y parámetros
Responsabilidades:	Descripción informal de las responsabilidades que la operación debe cumplir.
Tipo:	Nombre del tipo (Concepto, clase, interfaz)
Referencias Cruzadas:	Número de referencia de función del sistema, casos de uso, etc.
Notas:	Notas de diseño, algoritmos, etc.
Excepciones:	casos excepcionales
Salidas:	Salidas que son enviadas fuera del sistema tales como mensajes (No salidas de interfaz de usuario)
Precondiciones:	Supuestos sobre el estado del sistema antes de ejecutar la operación
Poscondiciones:	Estado del sistema después de ejecutar la operación.

El siguiente ejemplo ilustra un contrato para la descripción de la operación ingresarItem() del caso de uso comprar productos.

Contrato	
Nombre:	ingresarItem(id, cantidad)
Responsabilidades:	Introducir (registrar) la venta de un ítem y agregar esta a la venta, Desplegar la descripción del ítem y su precio
Tipo:	Sistema
Referencias Cruzadas:	Funciones de sistemas: R1.1, R1.2 Casos de uso: Comprar productos
Notas:	Utiliza acceso rápido a bases de datos
Excepciones:	Si id no es válido indicar con mensaje de error
Salidas:	
Precondiciones:	El id del ítem es conocido por el sistema
Poscondiciones:	1. Si se trata de una nueva venta, se creó una nueva Venta (creación de instancia) 2. Si se trata de una nueva venta, la nueva Venta fue asociada con el Punto de venta 3. Se creó un Ítem (creación de instancia) 4. El Ítem fue asociado a …

Las Poscondiciones describen cómo el sistema cambia de estado una vez que la operación es finalizada, esta sección no describe como se realiza la operación en tiempo de ejecución. No existe una forma estricta para declarar las poscondiciones pero se deben tener en cuenta los siguientes cambios ocurridos (expresados en el contexto del modelo conceptual):

1. Creación y eliminación de instancias.
2. Modificación de atributos.
3. Creación y eliminación de asociaciones.

La sección de Notas esta reservada para posibles comentarios sobre algoritmos conocidos que pueden ser aplicados a la operación. Esta sección se utiliza con mayor frecuencia en la fase de diseño.

En general, se deben seguir los siguientes pasos en creación de los contratos [Lar, 98]:

Para cada caso de uso:

1. Identificar las operaciones del sistema a partir del diagrama de secuencias de sistema.
2. Para cada operación de sistema, construir un contrato.
3. Comenzar escribiendo las Responsabilidades.
4. Completar las Poscondiciones en forma declarativa, describiendo el cambio de estado que ocurrió con los objetos en el modelo conceptual en base a:
    a. Creación y eliminación de instancias.
    b. Modificación de atributos.
    c. Creación y eliminación de asociaciones.

2.2.3.1 Conclusión

La fase de análisis se centra en el entendimiento de los requerimientos, conceptos y operaciones relacionadas con el sistema que ha de construir. De esta forma los artefactos antes nombrados responden a las siguientes preguntas:

Artefacto de Análisis	Responde a
Casos de uso	¿Cuáles son los procesos del dominio?
Modelo conceptual	¿Cuáles son los conceptos y términos?
Diagrama de secuencia del sistema	¿Cuáles son los eventos y operaciones del sistema?
Contratos	¿Qué hacen las operaciones del sistema?

Tabla 9: Respuestas entregadas por los artefactos del análisis

2.4 Fase de diseño

2.4.1 Introducción

Completada la fase de análisis y continuando con el ciclo de desarrollo iterativo, es posible pasar a la fase de diseño una vez construidos todos los artefactos relacionados con el análisis.

La fase de diseño tiene como finalidad diseñar una solución lógica basada en el paradigma de orientación objetos. La esencia de esta solución es crear diagramas de interacción, los cuales ilustran cómo los objetos se comunican entre si para satisfacer los requerimientos.

En esta fase se utilizan además los diagramas de clases de diseño, los que muestran una versión resumida de las clases e interfaces que serán finalmente implementadas como software. La creación de estos diagramas requiere gran cantidad de esfuerzo, creatividad, aplicación de principios de asignación de responsabilidades y utilización de patrones de diseño.

Un gran apoyo para diseñar la solución lógica que cumpla con los requerimientos descritos en el o los casos de uso presentes en la iteración actual, lo constituyen los patrones de diseño.

2.4.2 Artefactos

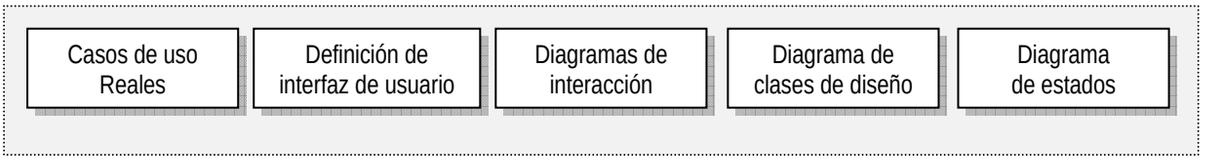


Fig.19 Artefactos para la fase de diseño

2.4.3 Definición del flujo de trabajo "Diseño"

2.4.3.1 Casos de uso reales y Storyboards

La primera actividad dentro de la fase de diseño es la creación de los casos de uso reales. Un caso de uso real muestra un diseño concreto para la realización de un caso de uso esencial. Un caso de uso real describe un diseño de un caso de uso en términos de entradas y salidas concretas basadas en alguna tecnología, en este caso, basados en tecnologías correspondientes a la plataforma J2EE y más específicamente para aplicaciones Web, por lo cual debemos considerar diagramas que ilustren la organización y aspecto de las páginas JSP y HTML que constituirán finalmente la interfaz de los usuarios del sistema. Los componentes antes nombrados, no deben ser necesariamente creados, se pueden utilizar storyboards (secuencia de pantallas) de interfaz de usuario sin profundizar en el detalle de su implementación, de esta forma se logra un acuerdo preliminar de la apariencias y forma uso de dichas interfaces.

2.4.3.2 Diagramas de Interacción

Una vez creados los casos de usos reales, con los bosquejos de interfaz de usuario cuando se amerite, el siguiente paso es crear los diagramas de interacción. Los diagramas de interacción, incluidos en UML, describen cómo los objetos interactúan mediante mensajes para completar las tareas descritas en los casos de uso reales, desde el punto de vista de software. La figura 20 muestra las dependencias entre artefactos para la actividad de creación de los diagramas de interacción.

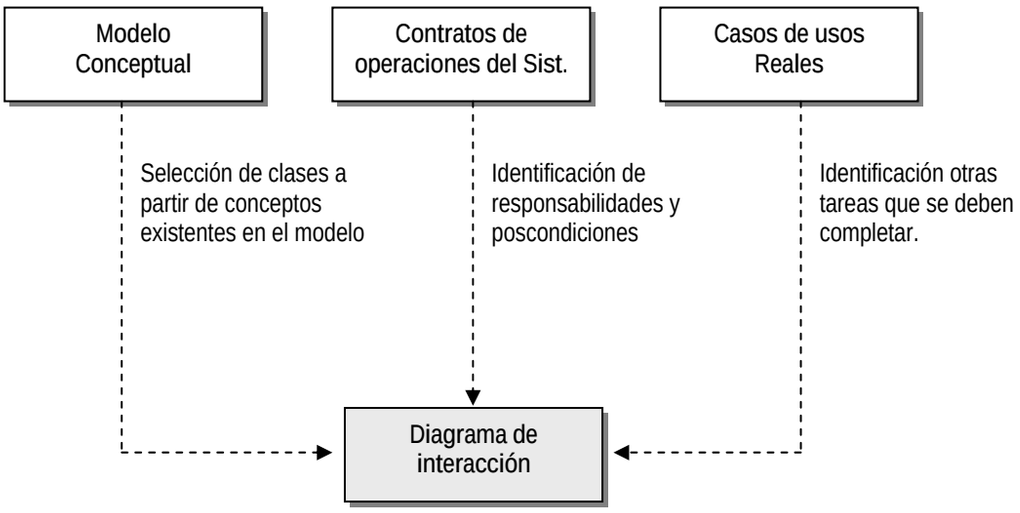


Fig.20 Artefactos necesarios para la creación del Diagrama de interacción

Un diagrama de interacción ilustra el intercambio de mensajes entre instancias y clases en el modelo de clases de diseño. UML define dos tipos de diagramas de interacción:

1. Diagramas de colaboración
2. Diagramas de secuencia

Los diagramas de colaboración, cómo el de la figura 21, ilustran la interacción entre objetos en un gráfico con formato de red.

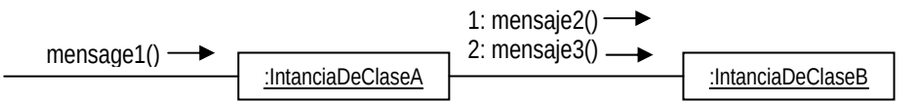


Fig.21 Ejemplo de diagrama de colaboración

Los diagramas de secuencia, cómo el de la figura 22, ilustran las interacciones en orden cronológico, en donde el tiempo transcurre desde arriba hacia abajo y los mensajes son enviados de izquierda a derecha con flechas rellenas. Opcionalmente, las respuestas pueden ser representadas por flechas en líneas punteadas con cabeza de fechas sin rellenar.

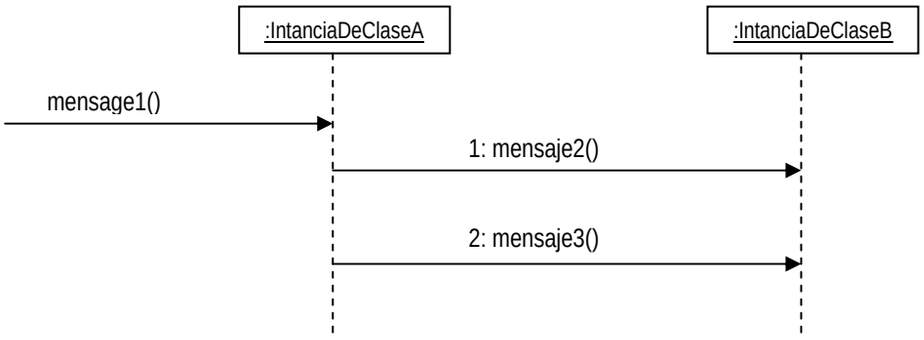


Fig.22 Ejemplo de diagrama de secuencia

Los diagramas de colaboración presentan una economía de espacio al dibujarlo, sin embargo, en la literatura J2EE existente, se utiliza con mayor frecuencia el diagrama de secuencia ya que ilustra con mayor claridad el orden cronológico de ocurrencias de las interacciones.

Los diagramas de interacción son uno de los artefactos más importantes creados en el análisis y diseño OO, por lo cual se debe destinar bastante tiempo y esfuerzo en su construcción. Para su correcto diseño es conveniente y recomendable utilizar patrones de diseño, como también poseer un alto entendimiento y manejo de estos diagramas.

Para construir un diagrama de colaboración se pueden seguir las siguientes pautas [Lar, 98]:

1. Crear un diagrama para cada operación de sistema involucrado en la iteración actual del ciclo de desarrollo. Para cada mensaje de operación de sistema, construir el diagrama con el mensaje inicial.
2. Utilizando las responsabilidades y poscondiciones descritas en el contrato de la operación, y descripción de casos de uso como punto de partida, diseñar un sistema de interacción de objetos que permita completar la realización de la operación. Aplicar patrones GRASP u otros patrones, como por ejemplo los patrones del catalogo de patrones de J2EE, para obtener un buen diseño.

Otro factor muy importante al momento de crear los diagramas de colaboración es determinar la visibilidad entre objetos. Visibilidad es la capacidad que posee un objeto de tener referencia sobre otro. Un objeto que envía un mensaje a otro debe tener cierta visibilidad sobre el objeto receptor con el fin de hacer posible la interacción. Se distinguen los siguientes tipos de visibilidad entre un objeto A que interactúa con un objeto B (A tiene visibilidad sobre B):

1. Visibilidad por atributo. B es un atributo de A, visibilidad permanente mientras exista A y B.
2. Visibilidad por parámetro. B es un parámetro de un método de A, visibilidad temporal dado que esta presente dentro del alcance del método de A.

3. Visibilidad por declaración local. B está declarado como un objeto local en algún método de A, visibilidad temporal dentro del alcance del método de A.
4. Visibilidad global. B es de alguna forma globalmente visible, visibilidad permanente mientras exista A y B.

UML incluye una notación para especificar la visibilidad entre objetos, la figura 23 ilustra los diferentes tipos de visibilidad.

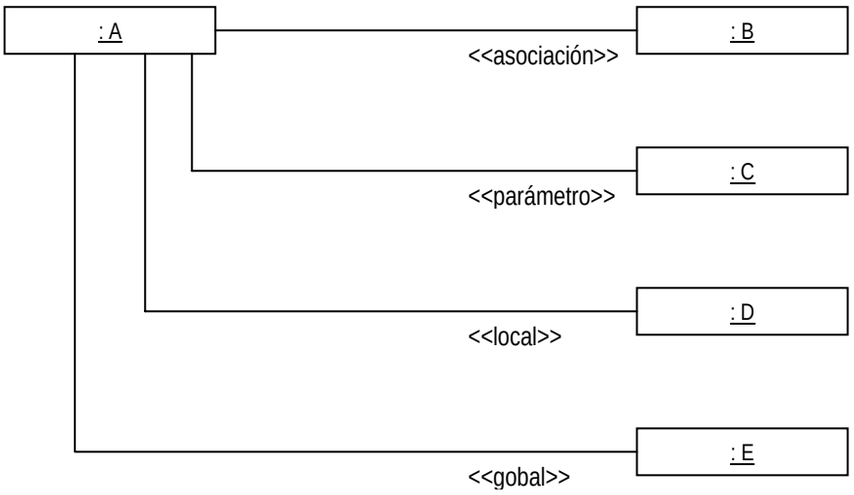


Fig.23 Notación UML para visibilidad entre objetos

Se utiliza <<asociación>> para describir el tipo de visibilidad por atributo.

2.4.3.3 Patrones J2EE

Algunos autores definen patrón como una solución recurrente a un problema, en un contexto determinado. El contexto hace referencia a condiciones ambientales o situaciones en la que se enmarca el problema. En otras palabras, un patrón define una solución probada (en Sun Java Center más de tres veces en proyectos diferentes) sobre un problema recurrente, por lo cual podemos reutilizarla una y otra vez en problemas o escenarios similares.

Los patrones son documentados en base a plantillas definidas por el autor, con el fin de contar con un formato genérico para el grupo de patrones. Lo anterior lleva a la situación de que existen tantas plantillas como autores o tecnologías hay, lo importante es que existe un esfuerzo por normalizar la documentación de patrones.

El 8 de Marzo del 2001, el Centro Java de Sun publicó la versión beta de su catálogo de patrones. En los primeros 60 días en los cuales estos patrones estuvieron a disposición de los desarrolladores, en formato HTML, fueron accedidos más de 120.000 veces, lo que evidencia el éxito de estas soluciones dentro de la comunidad Java [URL6].

En esencia, estos patrones contienen las mejores soluciones para ayudar a los desarrolladores a diseñar y construir aplicaciones para la plataforma J2EE. Aunque estos patrones son representados desde un nivel lógico de abstracción, todos contienen en sus descripciones estrategias que ofrecen pautas para su implementación.

La figura 24 ilustra la fragmentación en capas de aplicaciones J2EE. Cada una de estas capas cuenta con sus propias responsabilidades y por lo tanto con sus propios patrones, es decir, patrones orientados específicamente a solucionar problemas que se observan continuamente en estas capas [URL6].

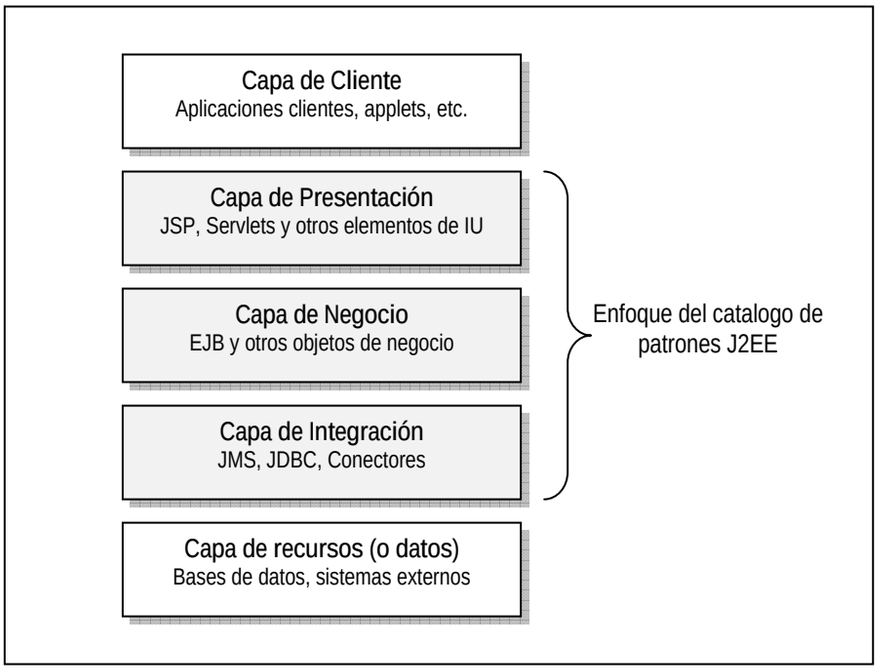


Fig.24 Fragmentación en capas de una aplicación J2EE

La capa de cliente representa todos los dispositivos o sistemas clientes que acceden a la aplicación. Un cliente puede ser un cliente Web, una aplicación Java o un teléfono móvil con acceso a WAP.

La capa de presentación encapsula toda la lógica de presentación requerida para servir a los clientes que acceden al sistema. Esta capa intercepta las solicitudes de los clientes, administra las sesiones, controla el acceso a servicios de negocio, construye y entrega la respuesta al cliente [URL6].

La capa de negocio proporciona los servicios de negocio requeridos en la aplicación. Esta capa contiene datos y lógica de negocios. Se utilizan por lo general componentes de negocio para implementar, centralizadamente en esta capa, la lógica de negocio o dominio de la aplicación [URL6].

La capa de integración es responsable de la comunicación con sistemas y recursos externos, por ejemplo datos almacenados en sistemas de base de datos relacionales. [URL6]

La capa de recursos contiene datos de negocio y recursos externos al sistema tales como bases de datos o sistemas de integración B2B. [URL6]

En las tablas 10, 11, 12 se listan los patrones más utilizados, asociados a las tres capas a las que prestan solución.

Nombre	Descripción
Intercepting Filter	Facilita el preprocesamiento y posprocesamiento de una solicitud.
Front Controller	Proporciona un controlador centralizado para la administración de solicitudes
View Helper	Encapsula la lógica que no esta relacionada con la presentación formateando mediante componentes Helper.
Composite View	Crea una visión agregada desde subcomponentes atómicos.
Service To Worker	Combina un componente Dispatcher con los patrones Front Controller y View Helper.
Dispatcher View	Combina un componente Dispatcher con los patrones Front Controller y View Helper.

Tabla 10: Patrones J2EE para Capa de Presentación

Nombre	Descripción
Business Delagate	Permite aislar las capas de presentación y de servicio proporcionando una interfaz a los servicios.
Value Object	Facilita el intercambio de datos entre capas reduciendo la sobre utilización de la red.
Session Facade	Oculta la complejidad de los objetos de negocio, centraliza la manipulación de flujos de trabajo.
Composite Entity	Representa una mejor práctica para diseñar beans de entidad agrupando objetos dependientes en un solo bean de entidad
Value Object Assembler	Ensambla un objeto valor compuesto desde múltiples fuentes de datos.
Value List Handler	Administra la ejecución de consultas, resultados y procesamientos.
Service Locator	Encapsula la complejidad de las operaciones de búsqueda y de creación de servicios de negocio.

Tabla 11: Patrones J2EE para Capa de Negocio

Nombre	Descripción
Data Access Object	Fuente abstracta de datos, proporciona el acceso transparente a los datos.
Service Activator	Facilita el procesamiento asincrónico para componentes EJB.

Tabla 12: Patrones J2EE para Capa de Integración

2.4.3.4 Patrones GRASP

Los patrones GRASP (General Responsibility Assignement Patterns) o patrones generales para asignación de responsabilidades, son un conjunto de patrones que ayudan a identificar responsabilidades sobre los objetos. No se encuentran ligados a una tecnología.

Booch y Raumbaugh definen responsabilidad como “un contrato o responsabilidad de un tipo o clase” [Boo+, 97]. La responsabilidad debe ser tomada desde el punto de vista de obligaciones en términos de comportamiento. Estas responsabilidades pueden ser de dos tipos:

1. Conocer
2. Hacer

La responsabilidad “Hacer” de un objeto implica:

- Hacer algo para si mismo
- Iniciar una acción sobre otros objetos.
- Coordinar y controlar actividades en otros objetos.

La responsabilidad “Conocer” involucra:

- Conocer acerca de datos privados encapsulados
- Conocer referencias a objetos
- Conocer cosas que pueden derivarse o calcularse.

Las responsabilidades son asignadas a los objetos durante el diseño orientado a objetos. Es importante no confundir las responsabilidades con los métodos. Los métodos son implementados

para llevar a cabo responsabilidades de un objeto. Mediante métodos los objetos pueden colaborar y comunicarse para cumplir con una función. En algunas ocasiones es necesario comunicar varios objetos con el fin de completar una responsabilidad, de esta forma podemos hablar de responsabilidades parciales entre objetos.

Al momento de asignar responsabilidades es muy importante tener en cuenta factores tales como bajo acoplamiento y alta cohesión, de esta forma podemos lograr que el sistema tenga las siguientes características, que habitualmente son deseables en un sistema:

- Encapsulación, los objetos utilizan su propia información para completar las tareas.
- Bajo acoplamiento, baja dependencia entre clases y alta probabilidad de reutilización. Un cambio en un componente no afecta a otro.
- Facilidad de entender y mantener.

El principal objetivo de los patrones de diseño es capturar buenas prácticas que nos permitan mejorar la calidad del diseño de sistemas, determinando objetos que soporten roles útiles en un contexto específico, encapsulando complejidad, y haciéndolo más flexible. Además los patrones incrementan nuestro vocabulario de diseño, ayudándonos a diseñar desde un mayor nivel de abstracción.

2.4.3.5 Diagramas de clases de diseño.

Una vez creado el diagrama de interacción, habiendo definido las responsabilidades de cada objeto y visibilidad entre estos, es posible crear el diagrama de clases de diseño, para lo cual es necesario identificar las clases, métodos y asociaciones. La figura 25 ilustra los artefactos necesarios para la creación del diagrama de clases de diseño.

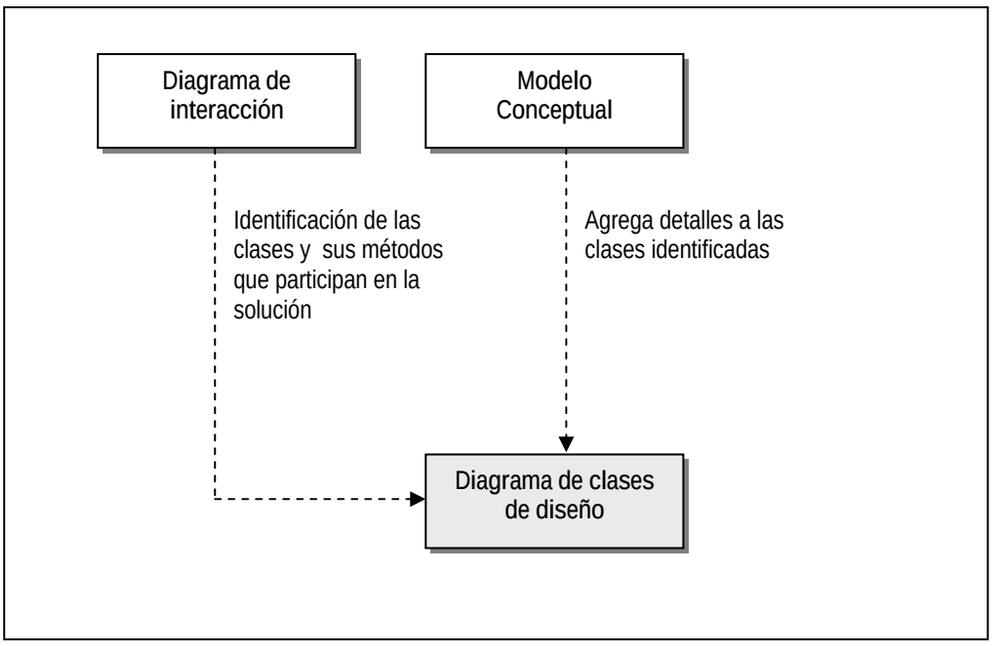


Fig.25 Artefactos involucrados en la creación del diagrama de Clases de Diseño

Un diagrama de clases de diseño ilustra la especificación de las clases de software y sus interfaces en la aplicación. La información incluida es la siguiente:

- Clases, asociaciones y atributos
- Interfaces con sus operaciones y constantes
- Métodos
- Atributos (tipos de información)
- Navegabilidad
- Dependencias

UML no incluye un elemento llamado específicamente “clases de diseño”, sólo incluye los diagramas de clases. El término agregado “diseño”, se utiliza para especificar que este artefacto es propio de la fase de diseño.

Se deben seguir los siguientes pasos para crear el diagrama de clases de diseño:

1. Identificar todas las clases participantes en la solución de software, analizando los diagramas de interacción (de la iteración actual).
2. Dibujar las clases identificadas en un diagrama de clases.
3. Agregar los atributos presentes en el modelo conceptual a cada clase.
4. Agregar a cada clase los métodos especificados en el diagrama de interacción.
5. Agregar los tipos de información a atributos y métodos.
6. Agregar las asociaciones necesarias para permitir la visibilidad de atributos entre clases.
7. Agregar las flechas de navegabilidad a las asociaciones para indicar la indicar la dirección de visibilidad de atributos de clases.
8. Agregar líneas de asociaciones de dependencias para indicar atributos visibles.

Es importante notar que la diferencia entre el modelo conceptual y diagrama de clases de diseño radica en que el modelo conceptual muestra una abstracción de conceptos de interés presentes en el mundo real. Por otro lado, el diagrama de clases de diseño muestra la definición de clases de software que implementarán la aplicación.

Hasta el momento se ha generado una primera aproximación del conjunto de clases de diseño que permitirá implementar la realización del caso de uso abarcado en la iteración en curso. Las actividades realizadas y los artefactos construidos hasta el momento forman una buena base de diseño que permitirá posteriormente ser refinada con el fin crear un modelo de negocio más robusto. El refinamiento se basa en perfeccionar las clases progresivamente,

agregando métodos y características propias relacionadas con los componentes tecnológicos sobre el cual se desarrollarán, esto implica analizar cada concepto o clase y decidir su implementación según el dominio del problema.

Algunos aspectos que deben ser considerados para su implementación en la plataforma son:

1. ¿Es necesario mantener el estado de un objeto entre sucesivas solicitudes (request) del cliente, es decir, manejar el estado conversacional?
2. ¿Los objetos operan sobre datos compartidos?
3. ¿Los objetos participan en transacciones?
4. ¿Dan servicios a un gran número de clientes?
5. ¿Necesitan se concebidos como componentes reutilizables?

La lógica de negocio de una aplicación se representa mediante objetos de negocio. En un sentido muy amplio, es el conjunto de procedimientos o de métodos usados para manejar una función específica del negocio. El análisis y diseño orientado al objeto, permiten descomponer una función de negocio en un conjunto de componentes o de elementos llamados objetos de negocio. Como todo objeto, los objetos del negocio tienen estado (o datos) y comportamiento. Por ejemplo, un objeto *empleado* tiene datos tales como un *nombre*, *dirección*, *edad* y tiene métodos para asignarlo a un nuevo departamento o para cambiar su sueldo en cierto porcentaje.

Es aconsejable aislar la lógica de negocio con el entorno de una aplicación, en este caso un entorno Web, con el fin de lograr una mayor reutilización de las componentes de la lógica de negocio.

2.5 Fase de implementación y pruebas

2.5.1 Introducción

Una vez concluida la fase de diseño, donde se obtuvo una primera aproximación de las clases que compondrán la lógica de negocio relacionada con los casos de uso de la iteración actual, es posible pasar a la fase de implementación.

La fase de implementación y pruebas se inicia con el refinamiento de las clases que implementan la lógica de negocio de la aplicación, lo cual generará como resultado el diagrama de clases de implementación y su programación en Java. Una vez logrado lo anterior, se debe crear la interfaz Web y flujo de trabajo que permitirán comunicar a los usuarios con este conjunto de clases (lógica de negocio), con el objetivo de permitir la realización de uno o más casos de uso abordados en la iteración actual. Para este fin se ha elegido utilizar el framework Struts, el cual permite implementar de manera relativamente fácil el patrón de arquitectura MVC en aplicaciones Web.

En esta etapa se hace necesario crear clases que prestarán servicios de comunicación a la lógica de negocio con medios de almacenamiento persistentes de datos, como por ejemplo, conexión con base de datos utilizando algún método proporcionado por Java como JDBC.

Las aplicaciones que son implementadas utilizando el patrón de arquitectura MVC, obtienen beneficios tales como: modularidad, diseño claro y ampliamente aceptado, facilidad para crear distintas vistas del mismo modelo de negocio y extensibilidad de la aplicación.

Una actividad importante considerada a lo largo de esta fase es el uso de los casos de prueba. Esta práctica, que ha sido planteada en las metodologías ágiles de desarrollo de

software, permite manejar de manera controlada las pruebas sobre cada una de las clases programadas, permitiendo crear colecciones de pruebas sobre partes o el total del sistema. Como herramienta de apoyo a la automatización de las pruebas se propone el uso de JUnit, un framework para escribir casos de prueba cuya ejecución y comprobación puede realizarse de manera rápida y fácil.

2.5.2 Artefactos de la fase de Implementación y Pruebas

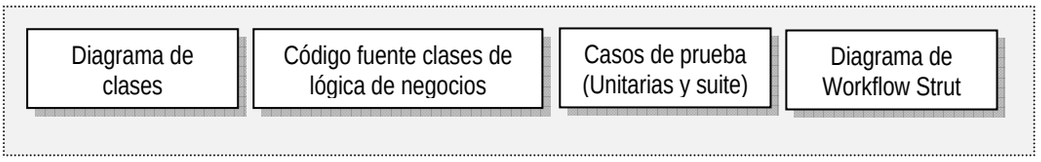


Fig.26 Artefactos para la fase de Implementación y Pruebas

2.5.3 Marco Teórico

2.5.3.1 El patrón de arquitectura MVC

El patrón de arquitectura MVC es un patrón muy utilizado para aplicaciones Web, dado que divide la aplicación en tres capas independientes: el Modelo (Objetos de Negocio), la Vista (interfaz con el usuario u otro sistema) y el Controlador (controlador del workflow de la aplicación).

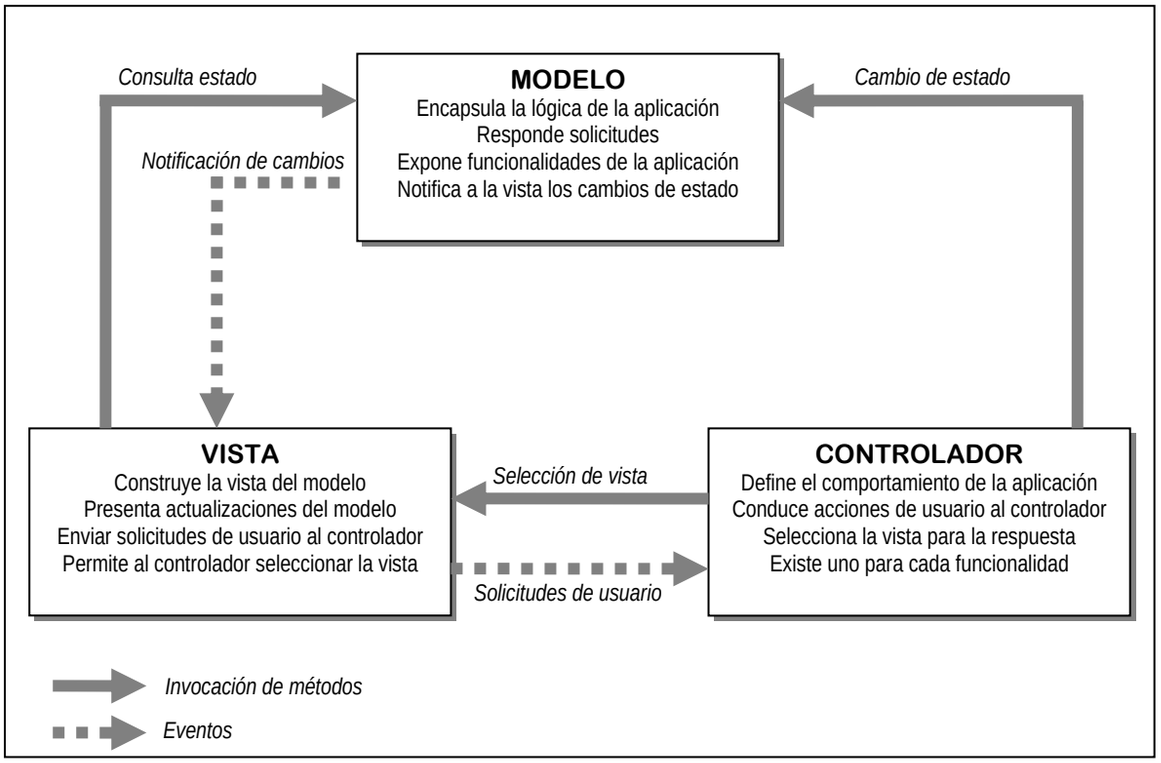


Fig.27 Esquema de interacción entre el Modelo, la Vista y el Controlador.

El Modelo representa datos y lógica de negocio u operaciones que acceden y modifican los datos de negocio. El modelo notifica a la vista cuándo su estado ha cambiado y le proporciona mecanismos para consultar su estado actual. También proporciona al controlador acceso a funcionalidades de la aplicación encapsuladas en el modelo. [URL8]

La Vista accede a los datos del modelo y especifica cómo estos deberían ser presentados, actualiza la presentación de los datos cada vez que el modelo cambia su estado. La Vista además representa un mecanismo para enviar las entradas del usuario al controlador [URL8].

El Controlador define el comportamiento de la aplicación, despacha las solicitudes del usuario y selecciona la vista adecuada para su presentación. El controlador interpreta las solicitudes de usuario y genera acciones para ser desarrolladas por el modelo. En aplicaciones Web esas solicitudes corresponden a solicitudes HTTP del tipo GET y POST. [URL8]

La separación de responsabilidades a través de los objetos Modelo, Vista y Controlador nos garantiza reducir la duplicación de código fuente en la aplicación y a la vez facilitar su mantención. Por otro lado, es fácil crear interfaces para nuevos tipos de clientes ya que no es necesario cambiar la lógica de negocio para cada nueva interfaz.

2.5.3.2 El Framewok Struts

Struts es un framework que implementa el patrón de arquitectura MVC (Model-View-Controller), en Java. Un framework es la extensión de un lenguaje de programación, mediante una o más jerarquías de clases que implementan una funcionalidad y que opcionalmente pueden ser extendidas por el programador.

Struts es parte del Proyecto Jakarta de Apache, patrocinado por la Apache Software Foundation y se encuentra disponible bajo la licencia "free-to-use-license" de la Apache Software Foundation.

Struts simplifica notablemente la implementación de una arquitectura según el patrón MVC, separa muy bien lo que es la gestión del workflow de la aplicación, del modelo de objetos de negocio y de la generación de interfaz de usuario.

Aunque el controlador ya se encuentra implementado por Struts, si fuera necesario se puede heredar y ampliar o modificar (extended), y el workflow de la aplicación se puede programar desde un archivo XML, por lo cual no es necesario modificar las clases para cambiar la lógica del workflow. Las acciones que se ejecutarán sobre el modelo de objetos de negocio, se implementan basándose en clases predefinidas por el framework y siguiendo el patrón Facade. La generación de interfaz se soporta mediante un conjunto de Tags o etiquetas predefinidas por Struts cuyo objetivo es evitar el uso de Scriplets (los trozos de código Java entre "<%" y "%>"), lo cual genera ventajas de mantención y de desempeño (pooling de Tags, caching, etc).

Struts separa claramente el desarrollo de interfaz de usuario y lógica de negocio, permitiendo desarrollar ambas en paralelo o con personal especializado, potencia la reutilización, soporte de múltiples interfaces de usuario (HTML, SHTML, WML, Aplicaciones de escritorio, etc.) y de múltiples idiomas.

En J2EE una aplicación Web utiliza un descriptor de despliegue para inicializar sus recursos, tales como servlet o etiquetas personalizadas. El descriptor es creado en formato XML en un archivo nombrado "web.xml". De la misma forma, Struts utiliza un archivo de configuración para inicializar sus propios recursos, "struts-config.xml", estos recursos incluyen:

- ActionForms, utilizados para recolectar las entradas de los usuarios.
- ActionMappings, para dirigir las entradas a los Actions (clases que definen dos métodos a ejecutar dependiendo del entorno servlet) de servidor.

- ActionForwards, para seleccionar las páginas que conformarán las respuestas a usuario.

Se debe tener especial cuidado al utilizar Struts, ya que uno de los errores más recurrentes es ligar la lógica de negocio con el entorno de una aplicación Web, es decir, codificar la lógica en algunos de las clases antes nombradas. Para evitar esto, la lógica de negocio debe ser encapsulada en beans de negocio, siendo transparente para estos beans su uso en entornos Web. Lo anterior nos asegura una mayor reutilización de los componentes de negocio.

La figura 28 muestra tipos de componentes mediante los cuales Struts implementa el patrón MVC. [URL11]

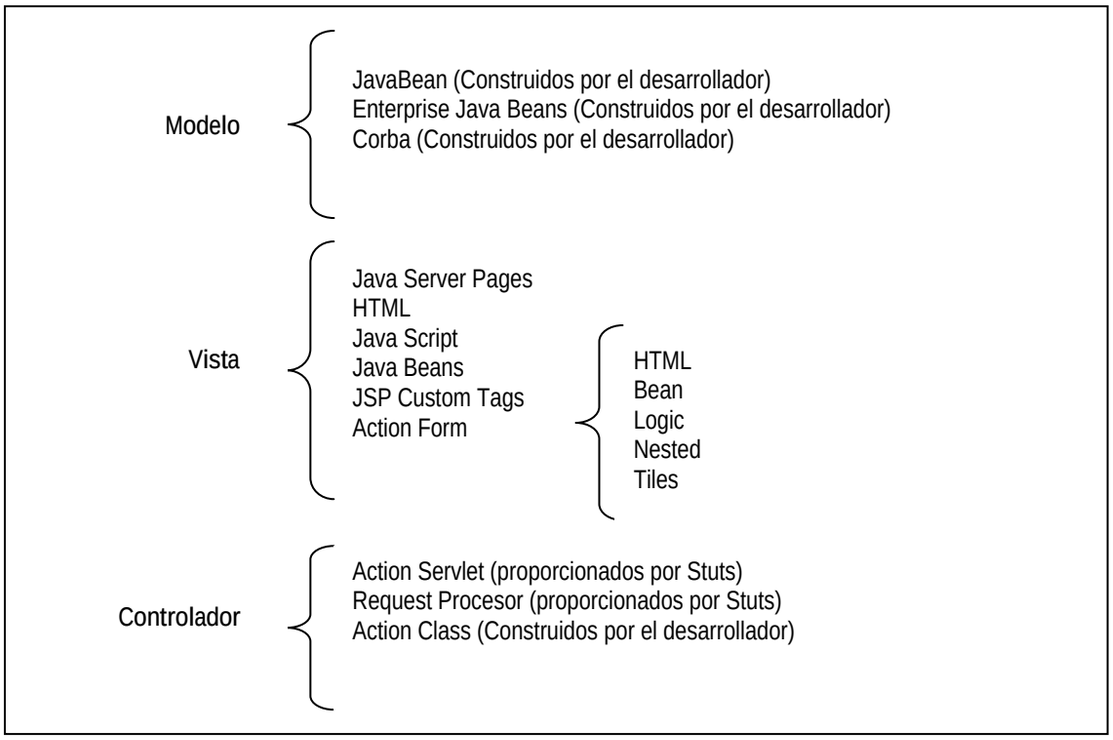


Fig.28 Componentes de Struts para implementación del patrón MCV

Las JSP Custom Tags, de la vista, son un conjunto de etiquetas disponibles para ser utilizadas desde las páginas JSP. La tabla X presenta las bibliotecas de etiquetas disponibles en Struts y su uso práctico.

Biblioteca de etiquetas	Utilidad
HTML	▪ Etiquetas utilizadas para crear formularios de entrada Struts (checkbox, submit, text, textarea).
Bean	▪ Utilizadas para acceder a JavaBeans y sus propiedades.
Logic	▪ Administración de generación condicional de salidas de texto. ▪ Iteraciones sobre objetos contenedores para generación de salidas repetitivas de texto. ▪ Administración del flujo de la aplicación.
Nested	▪ Proporcionan la capacidad de definir un modelo de objetos jerarquizado para representar y manejar ese modelo a través de etiquetas personalizadas desde páginas JSP.
Tiles	▪ Creación de páginas Web ensamblando piezas reutilizables.

Tabla 13. Etiquetas Personalizadas Struts.

Aunque las clases Action deben ser construidas por el desarrollador, Struts provee las siguientes clases preconstruidas [URL11]:

- ForwardAction
- DispatchAction
- LookupDispatchAction
- IncludeAction
- SwitchAction

La figura 29 ilustra los eventos que suceden al enviar un formulario Web que ha sido implementado con Struts.

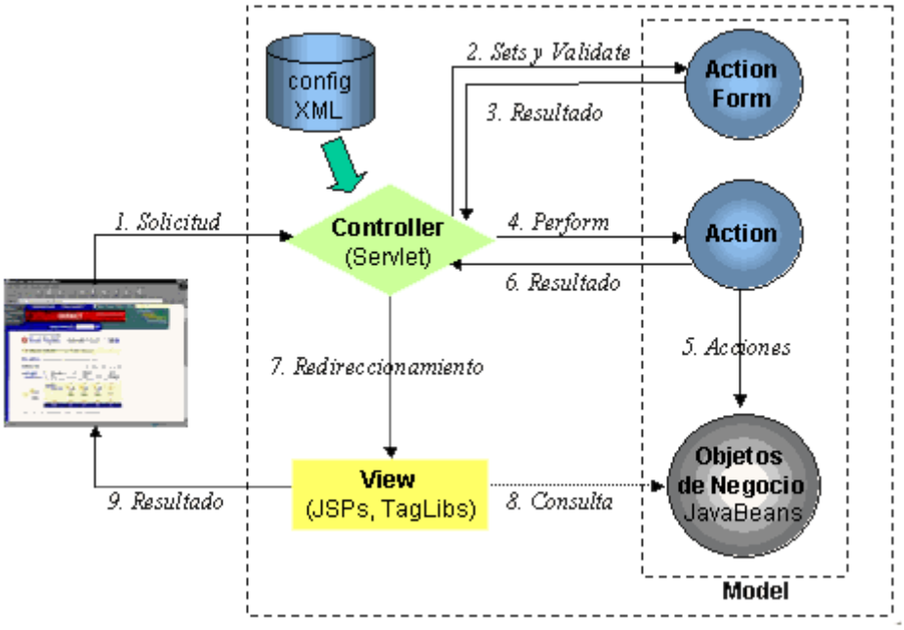


Fig.29 Flujo de interacción entre componentes Struts.

Cuando un usuario completa un formulario y lo envía, el Controlador busca el ActionFormBean correspondiente al formulario que se esta enviando y si no lo encuentra, lo crea. Esta relación esta configurada en el descriptor struts-config.xml. Luego, el Controlador realiza un set por cada entrada del formulario y finalmente llama al método *validate*. Si el método *validate* retornara uno o más errores, el Controlador llama a la página JSP del formulario para que ésta lo vuelva a generar, incluyendo los mensajes de error correspondientes. Si el método *validate* no retorna errores, llama al método *perform* del Action (también configurado en struts-config.xml) pasándole el ActionFormBean como parámetro para obtener los valores de los datos.

Si bien el ActionForm tiene características que corresponden al Modelo, los ActionForm pertenecen a la Vista. Justamente uno de estos puntos comunes es la validación de datos y a fines de evitar la duplicación de funcionalidad, si un ActionForm debe realizar controles de validación que se hubiesen implementado en un objeto de negocio entonces se debería utilizar una instancia de éste para efectuarlos.

2.5.3.3 Casos de prueba

El término “verificación de programas” hace referencia a cualquier tipo de proceso que trate de comprobar si un determinado programa, módulo o componente software cumple con los requisitos que se esperan de él. De esta manera, engloba muchos tipos de técnicas, incluyendo algunas que no requieren la ejecución – y ni siquiera la implementación – del programa, como las técnicas de verificación formal, así como otras que basan su comprobación en el análisis de la ejecución del programa implementado. De esta manera, podemos decir que la prueba es un tipo de verificación que se basa en la ejecución del código.

La prueba de programas es una aproximación dinámica a la verificación de los mismos, en la cual se ejecuta el software con unos datos o casos de prueba para analizar el buen funcionamiento de los requisitos esperados del programa

La prueba unitaria es la prueba de un módulo concreto dentro de un software que incluirá muchos otros módulos. Un módulo es una unidad de código que sirve como bloque de construcción para la estructura física de un sistema, por consiguiente podemos considerar como “módulo” una clase Java.

En los últimos años, ha aparecido una metodología de desarrollo de programas denominada Extreme Programming (XP), la cual es considerada como la más popular de las metodologías ágiles de desarrollo de software que haya sido planteada hasta el momento. XP da énfasis especial en las prácticas y técnicas de prueba unitaria, es por esto que se han creado una serie de frameworks para ayudar a realizar pruebas unitarias en diferentes lenguajes de programación. Al conjunto de esos frameworks se les denomina xUnit.

Como parte de la metodología de desarrollo se hará uso de JUnit, que es “el xUnit para Java”. Los conceptos fundamentales de los frameworks JUnit son los casos de prueba (test cases) y las suites de prueba (test suites).

Los casos de prueba son clases (o módulos, más en general) con una serie de métodos que ejecutan los métodos de una determinada clase (o módulo), que es el objeto de la prueba. De esta manera, cada clase tiene asociada otra clase que sirve para probarla. Para clases triviales, no merece la pena codificar un caso de prueba [URL7].

Este enfoque tiene las siguientes características:

- Las pruebas son también programas, y no simplemente especificaciones de datos de entrada y datos de salida esperados, de manera que pueden ejecutarse automáticamente, y puede quedar automatizada también la comprobación de si se ha pasado la prueba o no.
- Los casos de prueba no se descartan, sino que son un producto más del desarrollo, junto con el código fuente.
- Los casos de prueba se pueden estructurar en colecciones, de manera que se pueden ejecutar las pruebas relativas sólo a una parte del sistema o a todo el sistema.
- Según se va desarrollando la aplicación, se va generando una batería de pruebas más y más completa que sirve para realizar pruebas de regresión.

Esta última característica es especialmente importante desde el punto de vista de la calidad del software resultante. Cuando introducimos un cambio en una clase que ya ha sido probada, existe la posibilidad de que ese cambio genere un error en otra parte de la clase como efecto colateral. Para garantizar que el cambio no ha generado este tipo de efectos, debemos

volver a ejecutar todas las pruebas que se habían hecho anteriormente sobre la clase. A este proceso se le denomina prueba de regresión. [URL7]

La estructuración de los casos de prueba se realiza mediante suites. Una suite es una colección de casos de prueba que por lo general agrupa clases que están funcionalmente relacionadas. Por ejemplo, podemos agrupar en una suite todos los casos de prueba relacionados a un caso de uso, por ejemplo: Comprar producto o Registrar cliente. Ambas suites pueden a su vez estructurarse formando un árbol. De esta manera, las pruebas de nuestro sistema se estructuran en forma de árbol, siendo las hojas los casos de prueba, y permitiendo ejecutar cualquier subárbol a partir de cualquier nodo intermedio.

En esta técnica, es el mismo desarrollador de la clase quién codifica el caso de prueba unitaria, y lo hace como actividad inmediata al desarrollo de la clase e incluso paralela, de forma se garantiza que se hacen pruebas sobre todo el código que se escribe.

Por ejemplo, si la definición de la clase “Complejo” es la siguiente:

```
public class Complejo{

    private float parteReal;
    private float parteImaginaria;

 public Complejo(float parteReal, float parteImaginaria){
              parteReal = parteReal;
             parteImaginaria = parteImaginaria;
  }

 public float getParteReal(){
          return parteReal;
  }

  public float getParteImaginaria(){
          return parteImaginaria;
  }

  public Complejo sumar(Complejo c){
    return new Complejo( this.getParteReal() +  c.getParteReal(),
                                this.getParteImaginaria() + c.getParteImaginaria()  );
     }
}
```

El caso de prueba para la clase “Complejo” puede ser de cómo sigue:

```
import junit.framework.TestCase;

public class ComplejoTest extends TestCase{

    public ComplejoTest(String nombre){
        super(nombre);
    }

    public void testSumaComplejos(){
        // Crear dos instacias de Complejo
        Complejo c1 = new Complejo(3, 5);
        Complejo c2 = new Complejo(1, -1);
        // Sumar y asignar a nueva instancia
        Complejo resultado = c1.sumar(c2);
        // Verificar resultado real con valores esperados
        assertTrue(resultado.getParteReal()==4);
        assertTrue(resultado.getParteImaginaria()==4);
     }
    // Visualizar resultados en forma gráfica
    public static void main(String args[]) {
        String[] testCaseName = {ComplejoTest.class.getName()};
        junit.swingui.TestRunner.main(testCaseName);
    }
}
```

El resultado arrojado por JUnit fue el siguiente, figura 30:

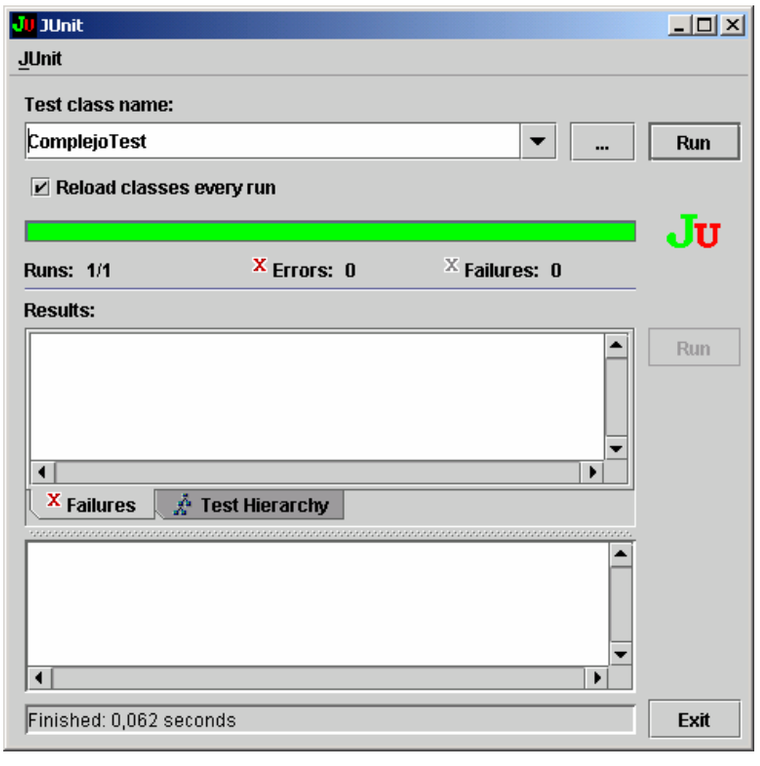


Fig.30 Interfaz gráfica JUnit.

Se puede observar que la clase “Complejo” se ejecutó correctamente para el caso de prueba construido.

En resumen, se deben seguir los siguientes pasos para realizar un caso de prueba [URL10]:

1. Crear una instancia de TestCase.
2. Crear un constructor que acepte un String como parámetro y pase éste a su superclase.
3. Sobrescribir el método runTest()
4. Utilizar el método assertTrue() y pasarle como parámetro un valor boolean que es verdadero si la prueba es exitosa.

2.5.4 Definición de flujo de trabajo “Implementación y pruebas”

2.5.4.1 Depuración de clases de la lógica de negocio.

La primera actividad de la fase de implementación y pruebas es la depuración del Diagrama de clases de diseño, obtenido en la fase de diseño. Dicha depuración se basa en obtener una definición de un conjunto de clases que permita ser codificada sin mayores inconvenientes en Java y que cumpla cabalmente con los requerimientos del caso de uso en curso. Para llegar a una definición precisa se hará uso de los Diagramas de clases, proporcionados por UML.

Un diagrama de clases sirve para visualizar las relaciones entre las clases que involucran el sistema, las cuales pueden ser asociativas, de herencia, de uso y de contenimiento.

Notación.

Un diagrama de clases esta compuesto por los siguientes elementos:

- Clase: atributos, métodos y visibilidad.
- Relaciones: Herencia, Agregación, Asociación y Uso.

En UML, una clase es representada por un rectángulo que posee tres divisiones, tal como lo muestra la figura 31:

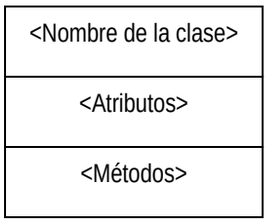


Fig.31 Estructura de una clase

En donde el rectángulo:

- Superior: Contiene el nombre de la clase.
- Intermedio: Contiene los atributos (o variables de instancia) que caracterizan a la clase (pueden ser private, protected o public).
- Inferior: Contiene los métodos u operaciones, los cuales son la forma como interactúa el objeto con su entorno (dependiendo de la visibilidad: private, protected o public).

Los atributos o características de una clase pueden ser de tres tipos, los que definen el grado de comunicación y visibilidad de ellos con el entorno, estos son:

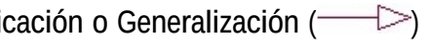
1. Public (+, ) Indica que el atributo será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.
2. Private (-, ) Indica que el atributo sólo será accesible desde dentro de la clase (sólo sus métodos lo pueden acceder).
3. Protected (#, ) Indica que el atributo no será accesible desde fuera de la clase, pero si podrá ser accedido por métodos de la clase además de las subclases que se deriven de ella.

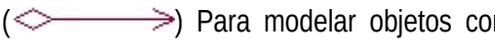
Los métodos u operaciones de una clase son la forma en como ésta interactúa con su entorno, éstos pueden tener las características

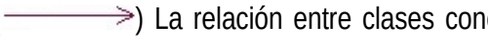
1. Public (+, ) Indica que el método será visible tanto dentro como fuera de la clase, es decir, es accesible desde otros lados.
2. Private (-, ) Indica que el método sólo será accesible desde dentro de la clase (sólo otros métodos de la clase pueden invocarlo).

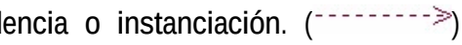
3. Protected (#,) Indica que el método no será accesible desde fuera de la clase, pero si podrá ser accedido por métodos de la clase además de métodos de las subclases que se deriven.

Las relaciones entre clases pueden se de cuatro tipos:

1. Herencia, Especificación o Generalización () Indica que una subclase hereda los métodos y atributos especificados por una Super Clase, por ende la Subclase además de poseer sus propios métodos y atributos, poseerá las características y atributos visibles de la Super Clase (public y protected).

2. Agregación. () Para modelar objetos complejos, n bastan los tipos de datos básicos que proveen los lenguajes: enteros, reales y secuencias de caracteres. Cuando se requiere componer objetos que son instancias de clases definidas por el desarrollador de la aplicación, tenemos dos posibilidades:
    i. Por Valor: Es un tipo de relación estática, en donde el tiempo de vida del objeto incluido esta condicionado por el tiempo de vida del que lo incluye. Este tipo de relación es comúnmente llamada Composición (el Objeto base se construye a partir del objeto incluido, es decir, es "parte/todo").
    ii. Por Referencia: Es un tipo de relación dinámica, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye. Este tipo de relación es comúnmente llamada Agregación (el objeto base utiliza al incluido para su funcionamiento).

3. Asociación. () La relación entre clases conocida como Asociación, permite asociar objetos que colaboran entre si. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro.

4. Dependencia o instanciación. () Representa un tipo de relación muy particular, en la que una clase es instanciada (su instanciación es dependiente de otro objeto/clase). Se denota por una flecha punteada. El uso más particular de este tipo de relación es para denotar la dependencia que tiene una clase de otra.

Una vez creado el diagrama de clases, éste debe ser programado en lenguaje de programación Java. Existen varias herramientas que permiten generar automáticamente el código base o estructura (sin lógica de los métodos) a partir del diagrama de clases, lo que permite agilizar considerablemente esta tarea y a la vez mantener documentado el diagrama de clases en un formato que es fácil de mantener e intercambiar entre los desarrolladores.

2.5.4.2 Ejecución de Casos de Prueba.

Una actividad paralela a la codificación de cada clase es la creación de los casos de prueba. Esta actividad tiene como fin asegurar el correcto comportamiento de cada clase. Los casos de prueba creados deben ser agrupados en una suite, con el fin de congregar casos de pruebas que guardan algún tipo de relación. La relación es evidente, ya que las clases probadas implementan uno o más casos de uso, relacionados con la iteración en curso.

2.5.4.3 Aplicación de patrón MVC utilizando Struts.

Una vez programadas y probadas las clases correspondientes a la lógica de negocio de la aplicación, estas deben ser utilizadas aplicando el patrón MVC. Para lo cual se deben seguir los siguientes pasos:

1. Diseñar un diagrama de flujo que describa el workflow entre páginas de la aplicación (formularios y/o respuestas) y relaciones con clases ActionForm o Action. Construir descriptor strut-config.xml a partir del diagrama creado. (ver Herramienta StrutsGUI para modelado visual y generación automática del descriptor).
2. Programar clases ActionForm y Action correspondientes.
    a. Los ActionForm deben tener atributos que correspondan con los campos del formulario asociado. Implementar método validate para validación de atributos.
    b. Los Action deben crear instancias de clases de la lógica de negocio para procesar las entradas de usuario y enbase a esto decidir la respuesta y siguiente página a desplegar. No implementar lógica de negocio en este componente.
3. Programar páginas JSP para las vistas.
4. Probar desarrollo con StrutsTestCase.

Como herramientas de apoyo a esta fase se propone el uso de StrutsGUI, StrutsTestCase y JMeter.

StrutsGUI es un esténcil para Microsoft Visio, con figuras que permiten modelar visualmente el workflow de una aplicación Struts descrito en descriptor strut-config.xml. Entre las características de esta herramienta se encuentran:

- Soporte de Apache Struts 1.3
- Generación automática de descriptor strut-config.xml
- Validación de correcta formación del diagrama.
- Ingeniería reversa. Convierte un descriptor strut-config.xml en un diagrama.

StrutsTestCase es una extensión de JUnit TestCase que proporciona mecanismos para probar código basado en el framework Struts. StrutsTestCase es compatible con las especificaciones 2.2 y 2.3 de servlet, Struts 1.1 y JUnit 3.8.1 [URL12]. Existen dos tipos de pruebas que se pueden realizar con StrutsTestCase:

1. Mock Testing. Prueba las clases simulando el contenedor en el que se ejecutan.
2. In-Container Testing. Prueba las clases ejecutándolas en un contenedor real.

2.5.4.4 Pruebas de stress y carga.

Una vez desarrollada la aplicación sobre Struts, opcionalmente se pueden realizar pruebas de rendimiento utilizando Apache JMeter. JMeter es un programa de la fundación Apache que permite observar el comportamiento de un servidor Web y medir su rendimiento [URL13]. Puede generar una demanda de intensidad variable para probar cómo se comporta un servidor sometido a diferentes niveles de carga. Entre otras cosas genera un análisis gráfico del rendimiento y guarda la historia de transferencias durante una prueba.

Conclusiones y mejoras

Conclusiones

1. La plataforma J2EE constituye una poderosa y completa base tecnológica para construir aplicaciones Web de carácter empresarial ya que provee todos los aspectos necesarios para implementarlas. Por otro lado, la independencia de plataformas de sistemas operativos y libre elección de proveedores, son características que están siendo valoradas por importantes empresas de nuestro país, al momento de realizar inversiones en el campo tecnológico.
2. Las técnicas de Análisis y Diseño Orientadas a Objetos, utilizadas en las actividades del proceso de desarrollo propuesto, nos ayudan a integrar a los usuarios y clientes en las primeras fases del proceso, produciéndose de esta forma un trabajo colaborativo entre ambas organizaciones. Asimismo, el uso de UML y patrones de diseño permiten al equipo desarrollador contar con un lenguaje en común para comunicar sus ideas y diseñar desde un mayor nivel de abstracción.
3. Las herramientas propuestas durante la fase de implementación, nos permiten disminuir notablemente los costos involucrados en el desarrollo de una aplicación Web. Estas herramientas, libres de uso, están ganado muchos adeptos dentro de la comunidad de desarrolladores de Tecnologías de Información. Una importante incubadora de estos proyectos lo constituye la Fundación Apache con su proyecto *Jakarta* que incluye alrededor de 19 sub-proyectos orientados a la plataforma Java, entre ellos el más conocido “Tomcat”, un contenedor de componentes Web.

4. En base a evidencia empírica obtenida al aplicar la metodología en el desarrollo de una aplicación real (ver Anexo A) se puede destacar que:
    a. UML como lenguaje de modelado, ayudó a mejorar la comunicación del grupo desarrollador, permitiendo plantear e intercambiar ideas que antes se transmitían en forma verbal, dando lugar a interpretaciones.
    b. Aunque en las etapas de diseño e implementación fue necesario invertir mayor tiempo que el de costumbre, el resultado final fue notablemente de mayor calidad, basándose en aspectos observados tales como:
        i. La documentación fue generada en paralelo con el desarrollo y no al final del proyecto, obteniendo una documentación más completa.
        ii. Se realizaron pruebas funcionales para cada componente desarrollado, lo cual garantiza que los componentes cumplen con los requerimientos.
        iii. El patrón arquitectónico MVC permitió construir una aplicación más robusta y fácil de mantener o modificar, en caso de cambios en los requerimientos.

Mejoras

1. Investigar más sobre técnicas formales que nos permitan enfrentar de mejor manera la fase de diseño, con el fin de obtener resultados de mayor calidad. Lo anterior puede ser logrado realizando una investigación más profunda sobre patrones de diseño y su aplicación práctica en Java.
2. Investigar y realizar evaluaciones prácticas de herramientas que nos ayuden a automatizar las actividades descritas en el proceso de desarrollo.
3. Investigar metodologías, que a partir de los casos de uso descritos en la fase de análisis, permitan estimar tiempos y costos involucrados en el proyecto de desarrollo de una aplicación Web.

Bibliografía

Libros

[Jac+,00] Ivar Jacobson, Grady Booch, James Rumbaugh
"El Proceso de Desarrollo de Software", Addison Wesley 2000.

[All, 00] Allaire Corporation, "Development Aplications with JRun 3.0", Allaire 2000.

[Boo+, 97] Booch, G.Jacobson, I., and Rumbaugh, J. "The UML specification documents". Santa Clara, CA.: Rational Software Corp. 1997

[Jac, 92] Jacobson, I. "Object-Oriented Software Engineering: A Use Case Driven Approach". Addison-Wesley 1992

[Arm+, 03] Eric Armstrong, Stephanie Bodoff, Debbie Carson, Ian Evans, Maydene Fisher, Dale Green, Kim Haase, Eric Jendrock, Monica Pawlan, Beth Stearns. "The J2EE™ 1.4 Tutorial". Sun Microsystems, Inc. 30 de Mayo 2003

[Rot+,03] Mark Roth, Eduardo Pelegrí-Llopart. "JavaServer Pages™ Specification Version 2.0", Sun Microsystems, Inc. 11 de Abril 2003

[Sha, 03] Bill Shannon, "Java™ 2 Platform Enterprise Edition Specification, v1.4", Sun Microsystems, Inc. 4 Noviembre 2003.

[Lar, 98] Larman, Craig, "Applying UML and Patterns: An introduction to Object-Oriented Analisys and Design", Prentice Hall, 1998.

Contenidos publicados en sitios Web

[URL1] http://www.ulpgc.es/otros/tutoriales/java/Cap1/puntero.html

[URL2] http://java.sun.com/j2ee/index.jsp

[URL3] http://java.sun.com/products/rmi-iiop/

[URL4] http://java.sun.com/developer/onlineTraining/EJBIntro/EJBIntro.html

[URL5] http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html

[URL6] http://java.sun.com/blueprints/corej2eepatterns/Patterns/

[URL7] http://www.dei.inf.uc3m.es/docencia/p_s_ciclo/ptelesup/01-02/junit/PruebaUnitaria.htm

[URL8] http://java.sun.com/blueprints/patterns/MVC-detailed.html

[URL9] http://junit.sourceforge.net/doc/testinfected/testing.htm

[URL10] http://www.junit.org/

[URL11] http://jakarta.apache.org/struts

[URL12] http://strutstestcase.sourceforge.net/

[URL13] http://jakarta.apache.org/jmeter/

Anexo A

Contenidos

INTRODUCCION

Con el fin de exponer el uso de la metodología de desarrollo propuesta en este proyecto de tesis, se demostrará su aplicación en un proyecto real de desarrollo de una aplicación Web. Por lo cual, el presente documento constituye el resultado de la implantación de la metodología en una empresa y un proyecto real, abarcando de esta manera todas las actividades implícitas en el proceso de desarrollo y los resultados obtenidos.

La implantación y prueba de proceso de desarrollo se enmarca dentro de la empresa Trío Ingeniería, empresa valdiviana que lleva más de un año prestando servicios de desarrollo de sistemas basados en tecnologías de información a empresas locales. Su distintivo son los proyectos basados en tecnología Web, es por esta razón que ha accedido tener la experiencia de probar el proceso de desarrollo de aplicaciones Web en su organización y equipo de trabajo.

ETAPA DE CAPTURA DE REQUISITOS

Resumen del problema.

A Trío Ingeniería, uno de sus clientes les ha encomendado la solución de un problema que hoy en día ha crecido exponencialmente y que afecta directamente al rendimiento de su negocio. El cliente, la empresa que necesita el servicio y que por razones estratégicas de mercado se mantendrá en el anonimato en este documento, ha decidido externalizar la fuerza de ventas para la comercialización de uno de sus servicios, pero se ha encontrado con una sorpresa. El volumen de ventas alcanzado en los últimos meses, que si bien es un gran logro y éxito del nuevo modelo operacional, ha ocasionado la saturación del procesamiento interno de los contratos de venta y habilitación de los servicios al cliente final.

El problema radica principalmente en que el cliente de Trío Ingeniería no contaba con suficiente personal, en BackOffice, para procesar los contratos de servicios y la decisión de aumentar el personal no esta dentro de sus planes. La lentitud del procesamiento ha causado una demora en la instalación del servicio al cliente final, causando el descontento de estos últimos. El procesamiento realizado manualmente a cada solicitud de servicios consiste en verificaciones de carácter técnico y aplicación de políticas comerciales vigentes, que se traduce una tarea tediosa y repetitiva, considerando que la información necesaria para esta toma de decisión se encuentra distribuida entre varios sistemas de información.

Una de las principales características del modelo de negocio del cliente de Trío Ingeniería, es que opera en forma distribuida geográficamente, con sucursales en las principales ciudades del Sur de Chile. Por esta razón es que ha debido contar con empresas aliadas que suministren el servicio de venta y comercialización de sus servicios, los distribuidores de servicios, llamados también dealers.

Considerando el escenario anteriormente expuesto, se decidió implementar una aplicación multiusuario basada en tecnologías Web, utilizando como canal de comunicación Internet, por ser considerado un servicio de bajo costo con el cual cuentan la mayoría de los participantes del proyecto.

Objetivos

- Agilizar el proceso de venta.
- Reducir al mínimo la intervención de personal involucrado en el procesamiento de solicitudes de servicio y su activación.
- Controlar el uso de contratos de servicios entregados a empresas distribuidoras.
- Generar informes para control de acciones de venta.

Funcionalidades del sistema

Ref. #	Funcionalidad	Categoría
R1.1	Ingresar solicitudes de servicio a través de Internet.	Evidente
R1.2	Verificar automáticamente el cumplimiento de políticas comerciales y restricciones técnicas de cada solicitud de servicio ingresada al sistema.	Oculta
R1.3	Mostrar el estado (aceptada o rechazada) de cada solicitud de servicio ingresada al sistema.	Evidente
R1.4	Editar solicitud de servicio ingresada al sistema.	Evidente
R1.5	Anular solicitud de servicio ingresada al sistema.	Evidente
R1.5	Generar automáticamente una orden de servicio en sistemas internos una vez aprobadas todas las políticas comerciales vigentes.	Oculta
R2.1	Iniciar sesión de usuario según rol asignado.	Evidente
R3.1	Asignación de formularios de contratos de servicios a cada distribuidor.	Evidente
R3.2	Anulación de formularios de contratos de servicio.	Evidente
R3.3	Manejar stock de formularios de contratos de servicio para cada distribuidor.	Oculta

Atributos del sistema

Atributo	Detalles	Aplica a funcionalidad
Tiempo de respuesta	Cuando se listan las solicitudes ingresadas al sistema, deben aparecer en menos de 10 segundos.	R1.3
Homologación de información	Al ingresar una solicitud de servicio, los nombres de ciudades, sectores y calles deben ser rescatados desde sistemas internos y no permitir que el distribuidor los ingrese manualmente con el fin de evitar errores.	R1.1
Software cliente del distribuidor	El distribuidor deberá hacer uso de un navegador Web para utilizar la aplicación.	
Procesamiento de solicitudes	El procesamiento de solicitudes ingresadas debe realizarse cada una hora y no en línea por motivos de seguridad.	R1.2 – R1.5

Glosario de términos

- Formulario de solicitud de servicio. Documento físico mediante el cual el vendedor captura todos los datos necesarios de un cliente que desee solicitar un servicio.

- Políticas comerciales. Reglas comerciales que deben ser verificadas antes de otorgar el servicio a un cliente (número de documentos adeudados en la empresa, monto máximo acumulado de la deuda o antecedentes financieros).

- Restricciones técnicas. Limitaciones técnicas que pueden ocurrir al momento de habilitar un servicio (factibilidad técnica verificada a través de SIG, Sistema de Información Geográfico).

- Orden de servicio. Transacción que gatilla la instalación e inicio de contratación del servicio para posteriormente ser cobrado.

- Formulario de contrato de servicio. Documento legal por el cual se genera un contrato entre el cliente y la empresa que presta el servicio.

Casos de uso

Actores relacionados con el sistema y procesos en los que participa

A continuación se presentan los actores relacionados con el sistema y lo procesos en que estos participan o inician.

Actor	Procesos en los que participa o inicia
Distribuidor de servicios	Iniciar sesión Ingresar solicitud de servicio Editar solicitud de servicio Anular solicitud de servicio Conocer estado de solicitudes de servicio Cerrar sesión Anular formulario de contrato de servicio
Asistente BackOffice	Iniciar sesión Cerrar sesión Asignar formularios de contrato de servicio Registrar nuevo distribuidor Editar distribuidor Crear cuenta de usuario Editar cuenta de usuario Configurar políticas comerciales

Descripción de casos de usos

A continuación se describen los casos en un formato de alto nivel de uso detectados.

Caso de uso:	INICIAR SESIÓN
Actores:	Distribuidor de servicios
Tipo:	Secundario - Esencial
Descripción:	Un usuario ingresa al sistema mediante algún mecanismo de identificación, de esta forma el sistema habilita las funcionalidades definidas para el rol que le fue asignado.

Caso de uso:	INGRESAR SOLICITUD DE SERVICIO
Actores:	Distribuidor de servicios
Tipo:	Primario - Esencial
Descripción:	El distribuidor de servicios ingresa al sistema todos los datos capturados en el *formulario de solicitud de servicio* para que el sistema los procese.

Caso de uso:	EDITAR SOLICITUD DE SERVICIO
Actores:	Distribuidor de servicios
Tipo:	Secundario - Esencial
Descripción:	El distribuidor de servicios selecciona una solicitud de servicio que aún no ha sido procesada por el sistema y edita sus datos.

Caso de uso:	ANULAR SOLICITUD DE SERVICIO
Actores:	Distribuidor de servicios
Tipo:	Secundario - Esencial
Descripción:	El distribuidor de servicios selecciona una solicitud de servicio que aún no ha sido procesada y la anula, de esta forma el proceso automático no la procesa, pero queda un registro de que esta fue ingresada al sistema.

Caso de uso:	CONOCER ESTADO DE SOLICITUDES DE SERVICIO
Actores:	Distribuidor de servicios
Tipo:	Secundario - Esencial
Descripción:	El distribuidor de servicios elige ver un listado con el estado de todas las solicitudes de servicios ingresadas en el mes actual. Puede ver el detalle de cada una como también seleccionar el historial de meses anteriores.

Caso de uso:	CERRAR SESIÓN
Actores:	Distribuidor de servicios
Tipo:	Secundario - Esencial
Descripción:	Un usuario que ha iniciado sesión elige abandonar el sistema.

Caso de uso:	ANULAR FORMULARIO DE CONTRATO DE SERVICIO
Actores:	Distribuidor de servicios
Tipo:	Secundario - Esencial
Descripción:	El distribuidor de servicios reporta la anulación de un formulario de contrato con el fin de que sea descontado de su stock disponible.

Caso de uso:	ASIGNAR FORMULARIOS DE CONTRATO DE SERVICIO
Actores:	Asistente BackOffice
Tipo:	Secundario - Esencial
Descripción:	El Asistente BackOffice asigna un nuevo rango de formularios de contratos a un distribuidor aumentando el stock de formularios disponibles a éste y dejándolo a disposición para su uso en sistema.

Caso de uso:	REGISTRAR NUEVO DISTRIBUIDOR
Actores:	Asistente BackOffice
Tipo:	Secundario - Esencial
Descripción:	El Asistente BackOffice ingresa una nueva empresa distribuidora al sistema, registrando toda la información relevante que la identifica.

Caso de uso:	EDITAR DISTRIBUIDOR
Actores:	Asistente BackOffice
Tipo:	Secundario - Esencial
Descripción:	El Asistente BackOffice selecciona una empresa distribuidora y actualiza su información.

Caso de uso:	CREAR CUENTA DE USUARIO
Actores:	Asistente BackOffice
Tipo:	Secundario - Esencial
Descripción:	El Asistente BackOffice ingresa un nuevo usuario asociándolo a una empresa distribuidora y zona de servicio. La cuenta debe ser enviada automáticamente al usuario.

Caso de uso:	EDITAR CUENTA DE USUARIO
Actores:	Asistente BackOffice
Tipo:	Secundario - Esencial
Descripción:	El Asistente BackOffice selecciona una empresa distribuidora y lista todos sus usuarios, selecciona uno específico y edita sus datos.

Caso de uso:	CONFIGURAR POLÍTICAS COMERCIALES
Actores:	Asistente BackOffice
Tipo:	Secundario - Esencial
Descripción:	El Asistente BackOffice lista las políticas comerciales vigentes y edita sus valores, actualizándolos en el instante para ser aplicados a las solicitudes de servicios sin procesar.

Diagrama de casos de uso

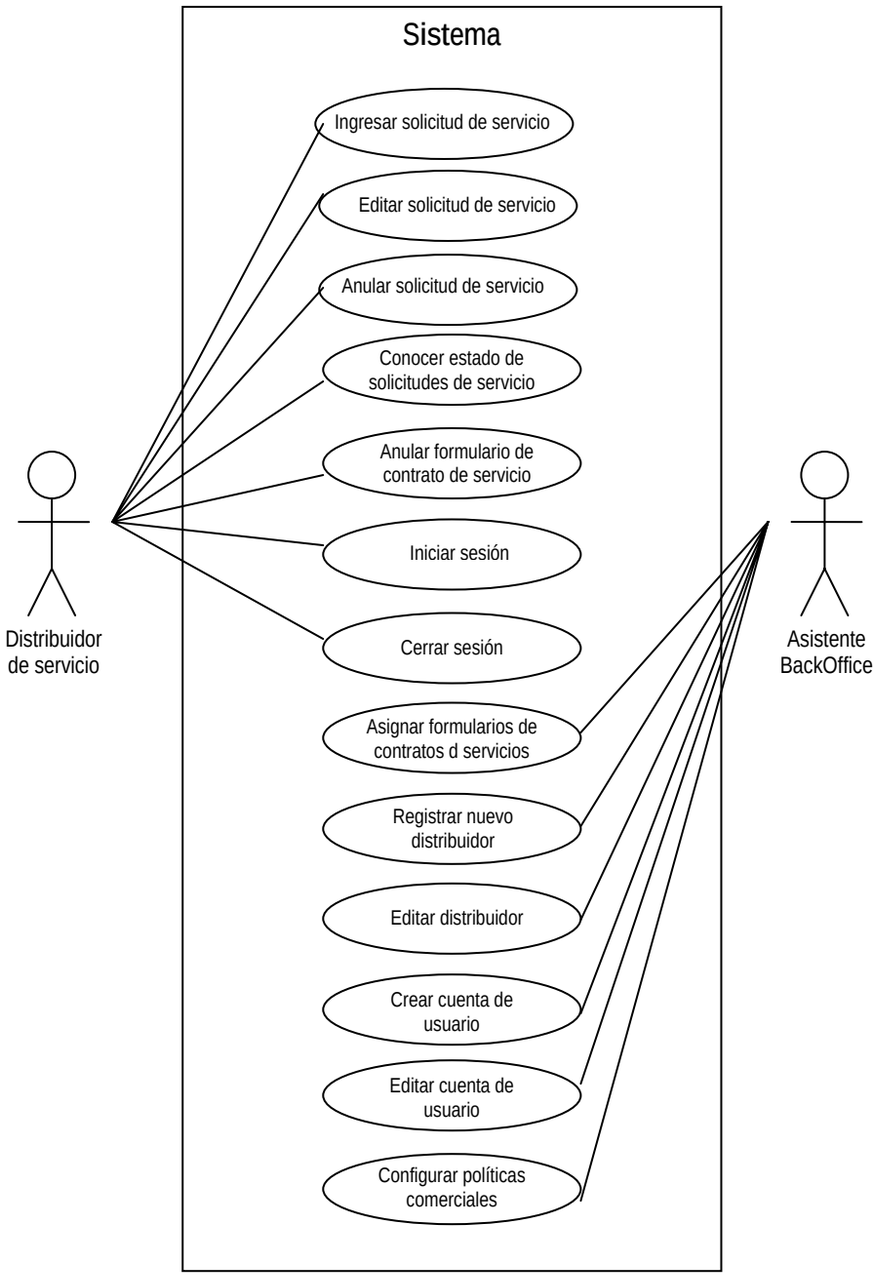


Fig. 1 Diagrama de casos de uso

Planificación de iteraciones

Clasificación de casos de uso

A continuación se presenta una evaluación de los casos de uso en base a seis aspectos, que permitirán clasificarlos con el objetivo de elegir los más importantes para ser desarrollados en las primeras iteraciones.

Casos de uso	a	b	c	d	e	f	Suma	Clasificación
Iniciar sesión	1	2	1	1	1	1	7	Bajo
Ingresar solicitud de servicio	4	5	3	2	5	3	23	Alto
Editar solicitud de servicio	1	2	1	1	1	1	7	Bajo
Anular solicitud de servicio	1	2	1	1	1	1	7	Bajo
Conocer estado de solicitudes de servicio	2	5	3	1	3	1	15	Medio
Cerrar sesión	1	2	1	1	1	1	7	Bajo
Anular formulario de contrato de servicio	1	2	1	1	1	1	7	Bajo
Asignar formularios de contrato de servicio	2	4	2	1	3	1	13	Medio
Registrar nuevo distribuidor	1	3	1	1	2	1	9	Bajo
Editar distribuidor	1	2	1	1	1	1	7	Bajo
Crear cuenta de usuario	1	3	1	1	2	1	9	Bajo
Editar cuenta de usuario	1	2	1	1	1	1	7	Bajo
Configurar políticas comerciales	1	4	3	1	3	1	13	Medio

Aspectos evaluados:

a. Posee un impacto significativo sobre el diseño arquitectónico.
b. Contiene información significativa.
c. Incluye alto nivel de riesgo, tiempo o funciones complejas.
d. Involucra investigación o uso de tecnología que no es dominada.
e. Representa un proceso significativo del negocio.
f. Incrementa los ingresos o disminuye los costos.

Escala de evaluación: 1 – 5

Clasificación: Bajo: 6 - 10 Medio: 11 – 22 Alto: 23 - 30

Planificación

		ITERACIÓN						
Casos de uso	**DURACIÓN**	**1°**	**2°**	**3°**	**4°**	**5°**	**6°**	**7°**
Ingresar solicitud de servicio	4 sem	■						
Configurar políticas comerciales	2 sem		■					
Asignar formularios de contrato de servicio	2 sem		■					
Conocer estado de solicitudes de servicio	1 sem			■				
Registrar nuevo distribuidor	1 sem			■				
Crear cuenta de usuario	1 sem				■			
Iniciar sesión	1 sem				■			
Cerrar sesión	1 sem				■			
Editar solicitud de servicio	1 sem					■		
Anular solicitud de servicio	1 sem					■		
Anular formulario de contrato de servicio	1 sem						■	
Editar distribuidor	1 sem						■	
Editar cuenta de usuario	1 sem							■

A modo de ejemplo se realizará la primera iteración con el caso de uso “Iniciar sesión”.

FASE DE ANALISIS

Descripción expandida del caso de uso “Iniciar sesión”

Descripción expandida	
Caso de uso:	Iniciar sesión
Actores:	Distribuidor de servicios
Propósito:	Iniciar sesión para disponer de las funcionalidades entregadas por el sistema.
Resumen:	Este caso de uso describe cómo un usuario ingresa al sistema con una sesión válida.
Tipo:	Secundario - Esencial
Referencias cruzadas:	

Curso de eventos normal

Acciones del actor	Respuestas del sistema
1.- Este caso de uso comienza cuando el distribuidor se conecta al SRPV (Sistema Remoto de Procesamiento de Ventas).	2.- Despliega la interfaz de inicio de sesión.
3.- El distribuidor ingresa su nombre de usuario y contraseña y los envía al SRPV.	4.- El sistema valida que el nombre de usuario y contraseña sean correctos. 5.- Aumenta el número de intentos de inicio de sesión en uno. 5.- Configura la aplicación en base al rol asociado al distribuidor que ingresó al sistema. 6.- Despliega un mensaje de bienvenida con los datos del distribuidor. 7. Registra la fecha y hora en que el distribuidor inició sesión en el sistema.
7.- El distribuidor se dispone a iniciar sus actividades.	

Cursos alternativos

Línea 4:	Nombre de usuario y contraseña inválidos. En este caso el sistema despliega un mensaje de error y debe permitir intentar nuevamente al usuario iniciar sesión. Se permitirá como máximo 3 intentos, si el usuario no puede iniciar sesión después de los tres intentos la cuenta del usuario será bloqueará por 24 hrs. y se enviará automáticamente su nombre de usuario y contraseña a su cuenta de correo electrónico.
Línea 4:	Cuenta deshabilitada. Si el sistema detecta que el nombre de usuario y contraseña son correctos pero que la cuenta esta deshabilitada, se debe presentar un mensaje de error para que el usuario se contacte con el administrador del sistema o espere 24 hrs. para su habilitación automática.

Modelo conceptual

A continuación se definirá el modelo conceptual, con el fin de ilustrar los conceptos más importantes en el dominio del problema.

Búsqueda de conceptos

Se identifican los conceptos presentes en la descripción narrativa del caso de uso “Iniciar sesión”.

Descripción expandida	
Caso de uso:	Iniciar sesión
Actores:	Distribuidor de servicios
Propósito:	Iniciar sesión para disponer de las funcionalidades entregadas por el sistema.
Resumen:	Este caso de uso describe cómo un usuario ingresa al sistema con una sesión válida.
Tipo:	Secundario - Esencial
Referencias cruzadas:	

Curso de eventos típico	
Acciones del actor	Respuestas del sistema
1.- Este caso de uso comienza cuando el distribuidor se conecta al SRPV (Sistema Remoto de Procesamiento de Ventas).	2.- Despliega la interfaz de inicio de sesión.
3.- El distribuidor ingresa su nombre de usuario y contraseña y los envía al SRPV.	4.- Valida que el nombre de usuario y contraseña sean correctos. 5.- Aumenta el número de intentos de inicio de sesión en uno. 5.- Configura las funcionalidades del SRPV en base al rol asociado al distribuidor. 6.- Despliega un mensaje de bienvenida con los datos del distribuidor. 7. Registra la fecha y hora en que el distribuidor inició sesión en el sistema.
7.- El distribuidor se dispone a iniciar sus actividades.	

Los siguientes son candidatos a conceptos:

DISTRIBUIDOR	SRPV	~~NOMBRE DE USUARIO~~
~~CONTRASEÑA~~	FUNCIONALIDAD	ROL
~~FECHA~~	~~HORA~~	~~NUMERO DE INTENTOS~~

Observaciones:

- Se elimina CONTRASEÑA, NOMBRE DE USUARIO y NUMERO DE INTENTOS por tratarse de atributos del concepto DISTRIBUIDOR.
- Se elimina FECHA y HORA ser atributos de DISTRIBUIDOR, corresponden al inicio de sesión, es decir cuando DISTRIBUIDOR se encuentra autentificado.

Modelo conceptual

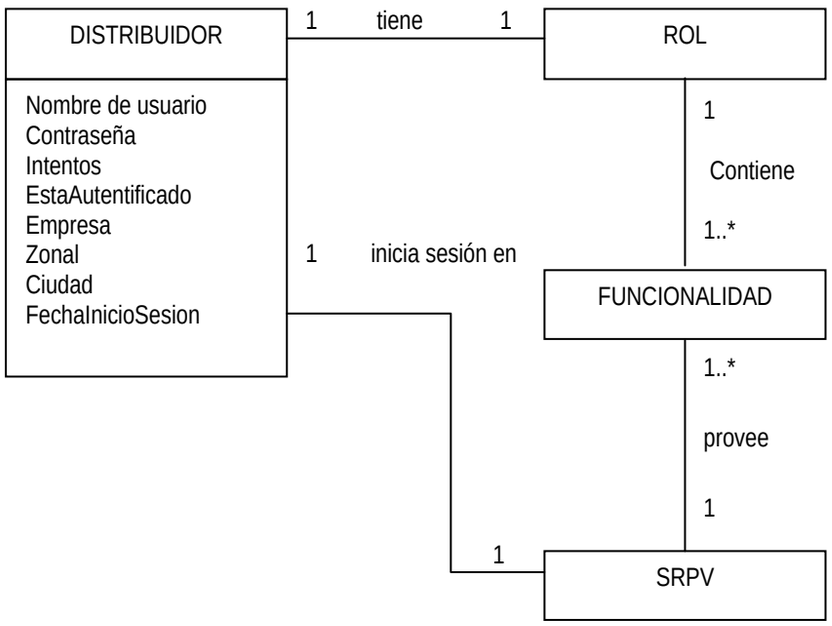


Fig.2 Modelo Conceptual

A continuación se explican cómo se derivaron algunas de las asociaciones presentes en el modelo conceptual.

- Distribuidor inicia sesión en SRPV
  Los requerimientos indican que el distribuidor debe utilizar el SRPV para sus actividades.

- Distribuidor posee un Rol
  Los requerimientos indican que se deben habilitar ciertas funcionalidades en el SRPV cuando un distribuidor inicia sesión. El Rol delimita las funcionalidades que estarán disponibles para el distribuidor.

- Rol contiene funcionalidades
  Rol actúa como un contenedor de funcionalidades.

Diagrama de secuencia de Sistema

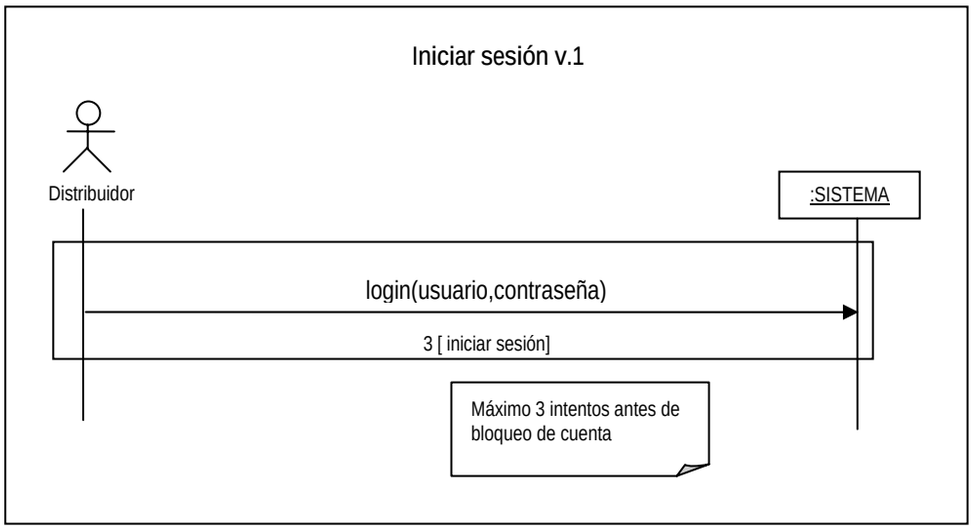


Fig.3 Diagrama de Secuencia de Sistema

Las operaciones del sistema son las siguientes:

Sistema
login(usuario,contraseña)

Contratos de operaciones del sistema

Nombre:	login(usuario,contraseña)
Responsabilidades:	Autentificar usuario para iniciar sesión en sistema
Tipo:	Sistema
Referencias Cruzadas:	Caso de uso “Iniciar sesión
Notas:	
Excepciones:	
Salidas	
Precondiciones:	
Poscondiciones:	1. Si se trata del primer intento de inicio de sesión, se creó un nuevo DISTRIBUIDOR 2. Se aumento DISTRIBUIDOR.intentos en uno. 3. Se creó ROL y fue asociado a DISTRIBUIDOR 4. Se crearon instancias de FUNCIONALIDAD y se asociaron a ROL.

FASE DE DISEÑO

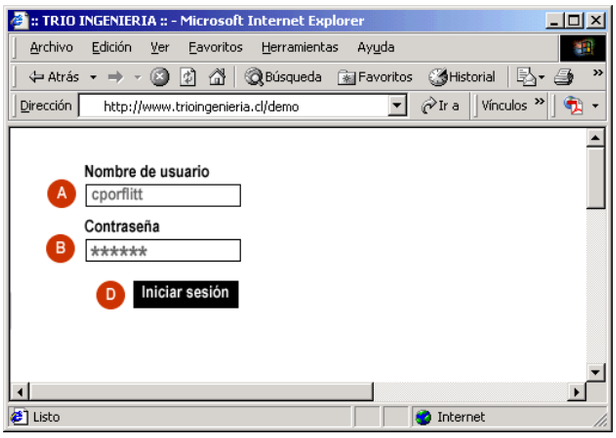


Fig.4 Interfaz Web de inicio de sesión

Caso de uso real	
Caso de uso:	Iniciar sesión
Actores:	Distribuidor de servicios
Propósito:	Iniciar sesión para disponer de las funcionalidades entregadas por el sistema.
Resumen:	Este caso de uso describe cómo un usuario ingresa al sistema con una sesión válida.
Tipo:	Secundario - Real

Curso de eventos típico

Acciones del actor	Respuestas del sistema
1.- Este caso de uso comienza cuando el distribuidor se conecta al SRPV (Sistema Remoto de Procesamiento de Ventas) mediante el URL o dirección HTTP en el cual se hospeda la aplicación Web desde su Terminal conectado a Internet.	2.- Despliega la interfaz Web de inicio de sesión, Fig.4.
3.- El distribuidor ingresa su nombre de usuario en el campo A y contraseña en el campo B y los envía al SRPV haciendo clic en el D.	4.- Valida que el nombre de usuario y contraseña sean correctos realizando una consulta a la base de datos. 5.- Aumenta el número de intentos de inicio de sesión en uno. 6.- Configura la aplicación en base al rol asociado al distribuidor que ingresó al sistema. La configuración se almacena en una variable de sesión en el servidor Web. 7.- Despliega un mensaje de bienvenida con los datos del distribuidor. 8. Registra la fecha y hora en que el distribuidor inició sesión en el sistema, el registro se realiza en la base de datos.
8.- El distribuidor se dispone a iniciar sus actividades.	

Diagrama de secuencia

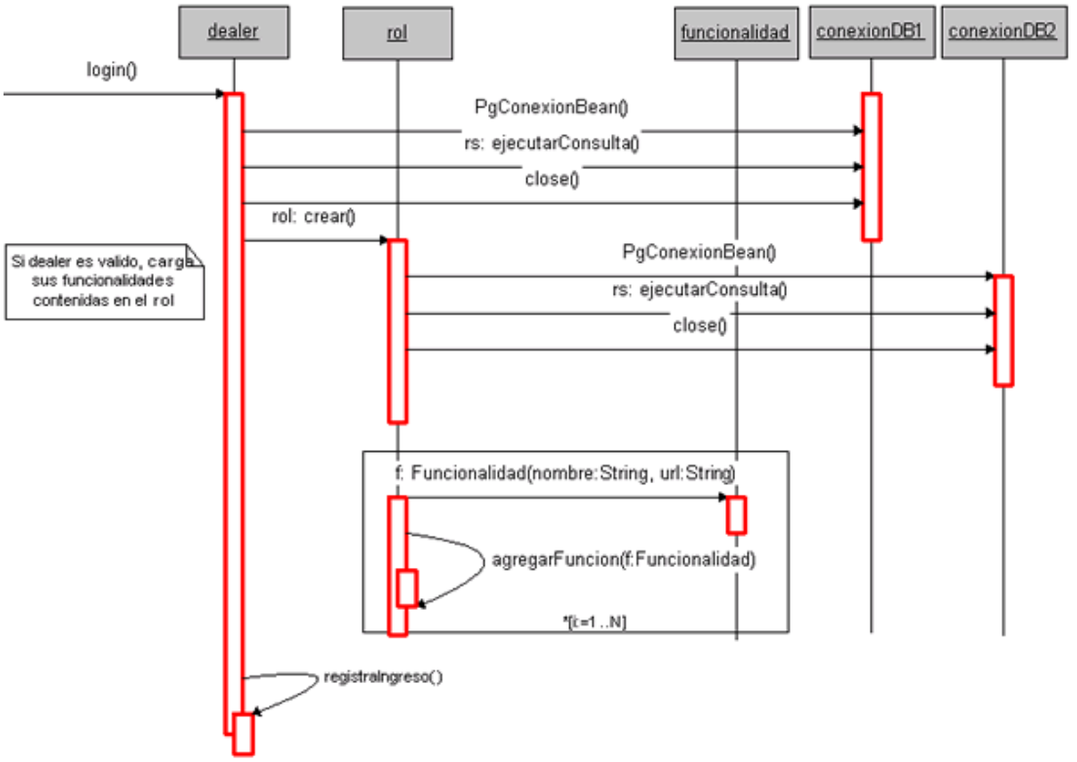


Fig.5 Diagrama de secuencia

Al invocar el método "*login()*" de la instancia *dealer*, se crea una instancia *conexiónDB* para ejecutar una consulta a la base de datos y completar sus atributos. Luego *dealer* crea una instancia *rol*, el cual almacenará todas sus funcionalidades en el sistema. *Rol*, al momento de crearse realiza una consulta a la base de datos, mediante la instancia *conexionDB2*, para cargar sus funcionalidades almacenadas en el medio persistente. Por último la instancia *dealer* registra el ingreso al sistema invocando su propio método "*registraIngreso()*".

Aplicando el *Patrón Creador* para asignación de responsabilidad se obtiene que:

- La instancia *dealer* tiene la responsabilidad de crear el objeto rol ya que cuenta con la información necesaria para crearlo y *dealer* contiene a *rol*.
- La instancia *rol* tiene la responsabilidad de crear instancias de *funcionalidad* ya que *rol* almacena o contiene múltiples instancias de *funcionalidad*.

Visibilidad entre objetos.

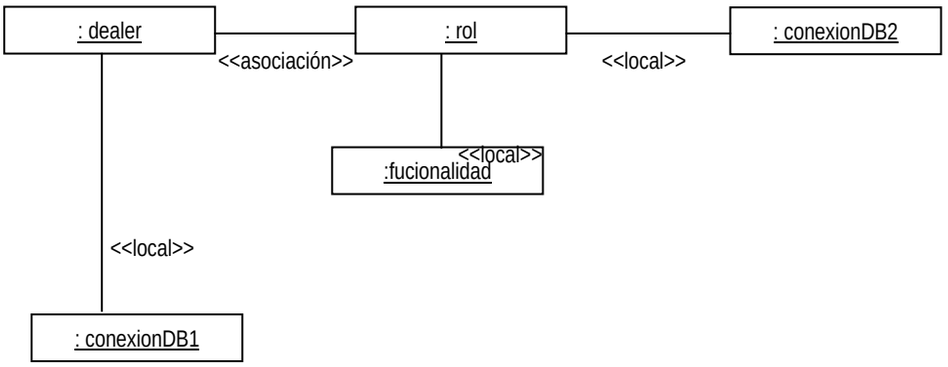


Fig.5 definición de Visibilidad entre Objetos

- rol tiene visibilidad de asociación con dealer ya que es declarado como atributo de dealer.
- funcionalidad tiene visibilidad local con rol ya que es declarado localmente dentro de un método de rol.
- conexionDB2 tiene visibilidad local con rol ya que es declarado localmente dentro de un método de rol.
- conexionDB1 tiene visibilidad local con dealer ya que es declarado localmente dentro de un método de dealer.

Diagrama de clases de diseño

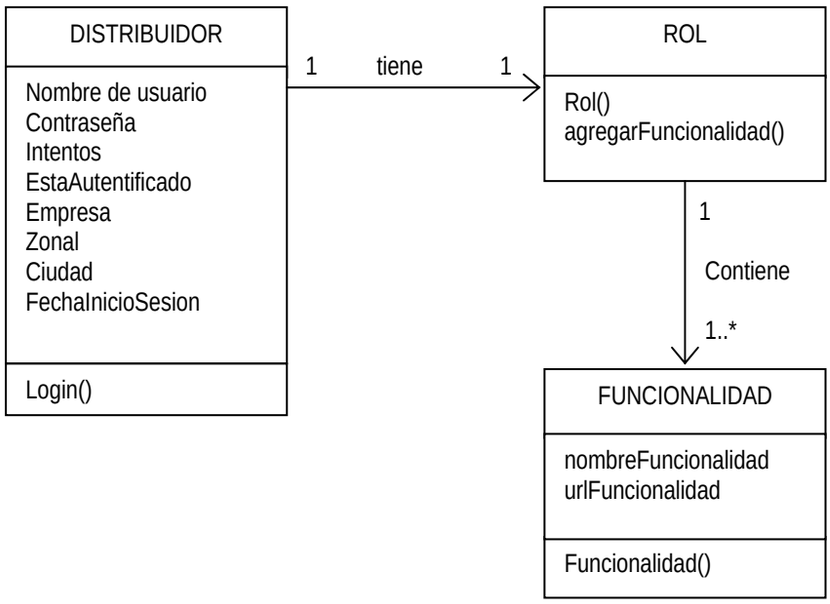


Fig.6 Diagrama de clases del diseño

FASE DE IMPLEMENTACION Y PRUEBAS

Diagramas de clases de implementación

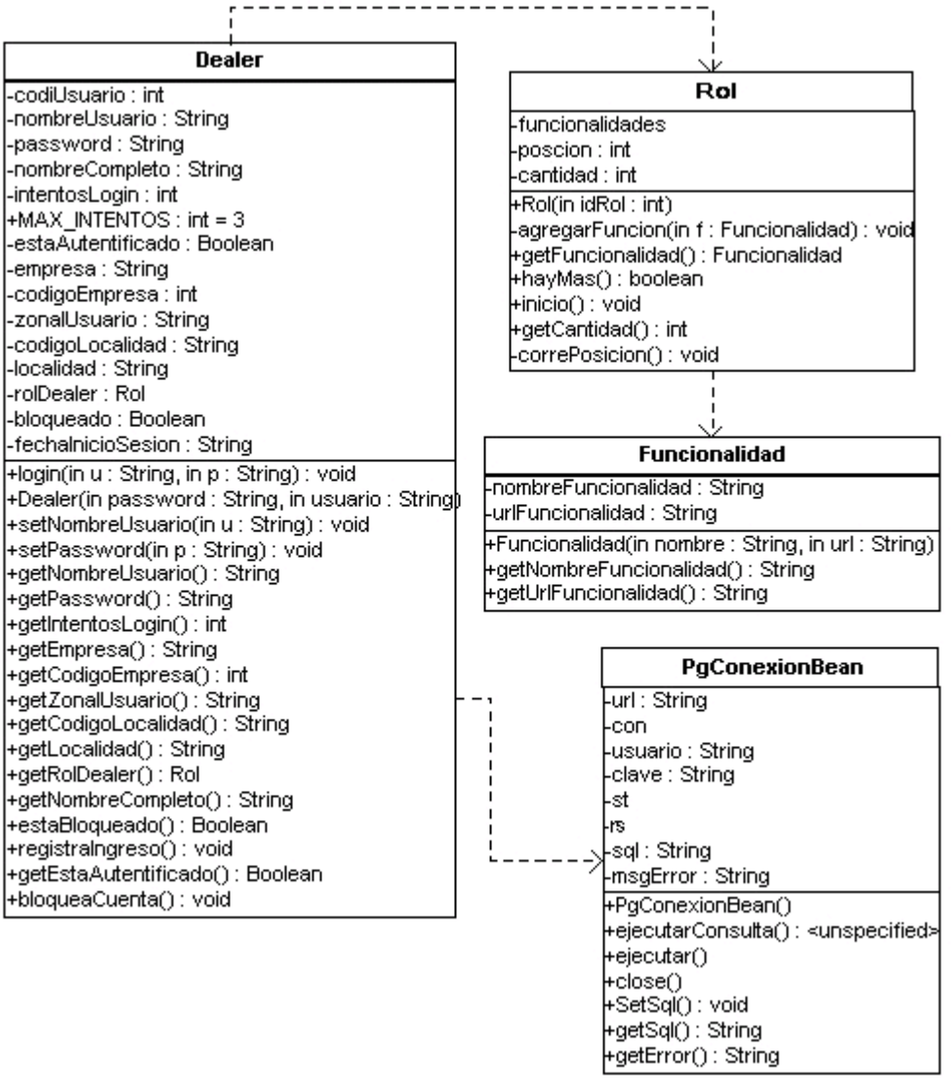


Fig.7 Diagrama de clases de implementación

Casos de prueba con JUnit.

Se implementó un caso de prueba para la clase *dealer.* El caso de prueba verifica la correcta autentificación de un usuario existente en la base de datos.

```
package com.trio.dealer;
import junit.framework.TestCase;

public class DealerTest extends TestCase{
    public DealerTest(String nombre){
     super(nombre);
}
public void testLogin(){
     Dealer dealer = new Dealer("alejandra","muribe");
     dealer.login();
     assertTrue(dealer.estaAutentificado()==true);
}
 public static void main(String args[]) {
       String[] testCaseName = {DealerTest.class.getName()};
        junit.swingui.TestRunner.main(testCaseName);
 }
}
```

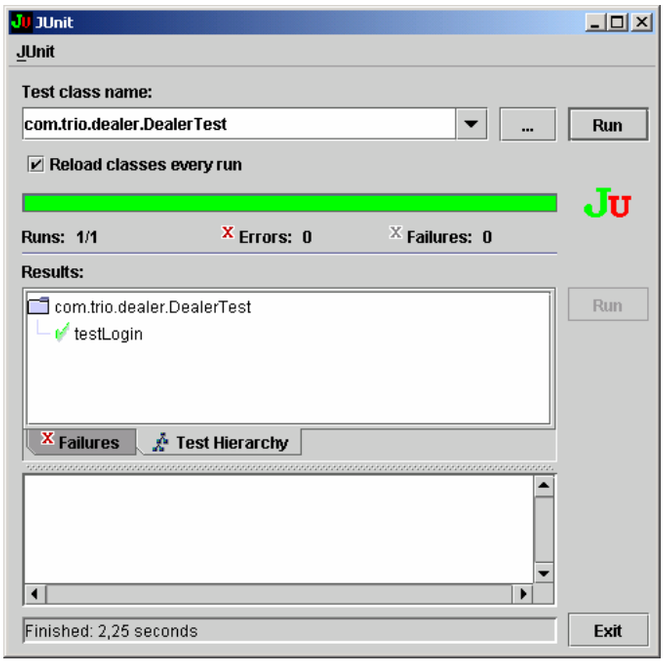


Fig.8 Resultado de ejecución de caso de prueba

Patrón MVC - Struts

Diagrama de configuración Struts

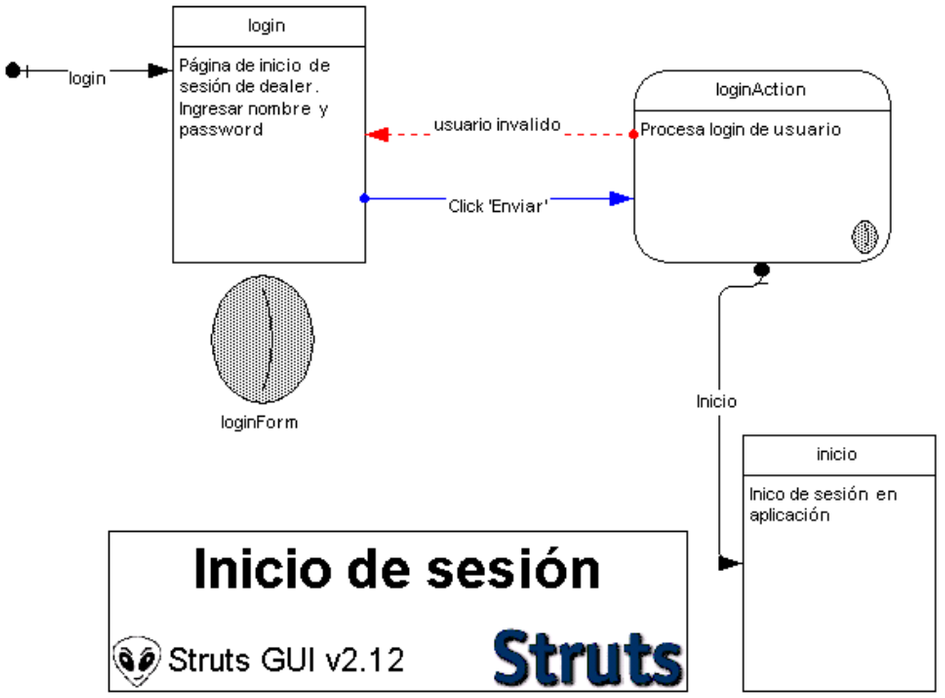


Fig.9 Diagrama configuración de Struts diseñada con StrutsGUI

El diagrama muestra gráficamente la configuración de Struts para el caso de uso Inicio de sesión. La página *login* implementa la interfaz de usuario para el ingreso de la información necesaria para iniciar sesión, nombre de usuario y password. Los datos enviados son capturados automáticamente y traspasados a *loginForm*, el cual se encarga de validar los datos (que no estén en blanco) y encapsularlos para ser procesados por *loginAction*. *loginAction* crea una instancia de *dealer* para hacer uso de la lógica de negocio de autentificación e inicio de sesión en la aplicación, representada por la página de *inicio*.

Descriptor strut-config.xml

El siguiente descriptor fue generado automáticamente a partir del diagrama de configuración Struts, utilizando la herramienta StrutsGUI.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!-- Struts Config XML - Inicio de sesión -->
<!-- ================================== -->
<!-- AutoGenerated from : c:\archivos de programa\visio\solutions\appdealer.vsd -->
<!-- AutoGenerated on   : 03-10-2004 23:07:22 -->
<!-- AutoGenerated by   : Struts GUI v2.13   (c)2002 Alien-Factory -->

<!DOCTYPE struts-config PUBLIC
                          "-//Apache Software Foundation//DTD Struts Configuration 1.0//EN"
                          "http://jakarta.apache.org/struts/dtds/struts-config_1_0.dtd">
<struts-config>

<!-- ==================================== Data Source Configuration -->

<!-- ======================================== Form Bean Definitions -->
<form-beans>
                          <form-bean  name="loginForm"
                                      type="com.trio.dealer.struts.LoginForm">
                                <description>Bean formulario de login</description>
                          </form-bean>
</form-beans>
<!-- ================================= Global Exception Definitions -->
   <global-exceptions>
   </global-exceptions>
<!-- ================================== Global Forward Definitions -->
<global-forwards>
  <forward name="login"   path="/login.jsp"/>
  <forward  name="inicio"  path="/inicio.jsp"/>
</global-forwards>
<!-- ================================== Action Mapping Definitions -->
<action-mappings>
 <action path="/loginAction"
         type="com.trio.dealer.struts.LoginAction"
        name="loginForm"
        scope="request"
        validate="true"
        input="/login.jsp">
        <description>Procesa login de usuario</description>
  </action>
/action-mappings>

</struts-config>
```

Modelo de datos relacional

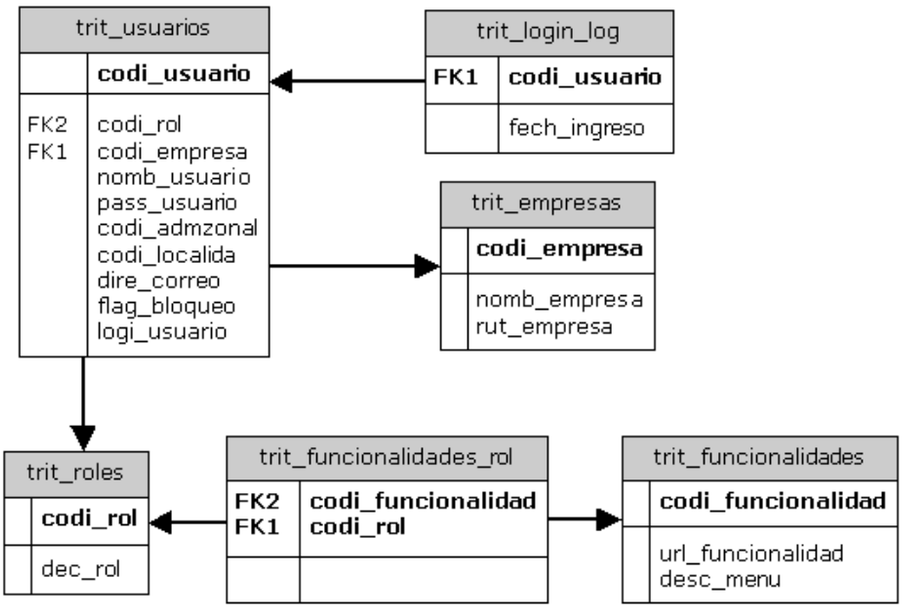


Fig.10 Esquema de Modelo de datos Relacional

Descripción de tablas del modelo de datos

trit_empresas		
Campo	*Tipo*	*Descripción*
codi_empresa	smallint	Identificador único de empresa
rut_empresa	Varchar(12)	RUT de empresa

trit_login_log		
Campo	*Tipo*	*Descripción*
codi_usuario	smallint	Identificador único de usuario
codi_rol	smallint	Identificador único de rol
codi_empresa	smallint	Identificador único de empresa
nomb_usuario	varchar(100)	Nombre completo usuario
pass_usuario	varchar(30)	Password de usuario
codi_admzonal	char(4)	Codigo de zonal
codi_localida	char(4)	Codigo de localidad
dire_correo	varchar()	e-mail
flag_bloqueo	char(1)	Bloqueo cuenta, S: Si – N: No
logi_usuario	varchar(30)	Nombre de usuario de cuenta

trit_usuarios		
Campo	*Tipo*	*Descripción*
codi_usuario	smallint	Identificador único de usuario
Fech_ingreso	date	Fecha de ingreso al sistema.

trit_roles		
Campo	*Tipo*	*Descripción*
codi_rol	smallint	Identificador único de rol
dec_rol	varchar(100)	Descripción del rol.

trit_funcionalidades		
Campo	*Tipo*	*Descripción*
codi_funcionalidad	smallint	Identificador único de funcionalidad
url_funcionalidad	varchar(256)	URL de funcionalidad, link a Página o Servlet.
desc_funcionalidad	varchar(50)	Nombre de opción en menú.

trit_funcionalidades_rol		
Campo	*Tipo*	*Descripción*
codi_funcionalidad	smallint	Identificador único de funcionalidad
codi_rol	smallint	Identificador único de rol

Plataforma tecnológica

El siguiente diagrama ilustra la plataforma componentes tecnológicos utilizados para la implementación de la solución.

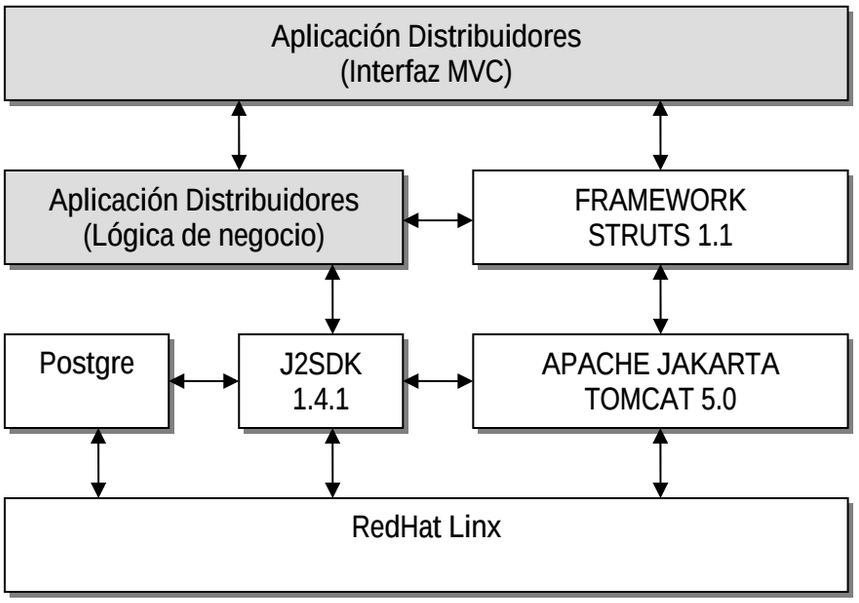


Fig.11 Diagrama de Plataforma Tecnológica

- RedHat Linux. Sistema Operativo
- PostgreSQL. Base de datos
- J2SDK. El paquete J2SDK contiene el entorno de desarrollo de Java de Sun. Sirve para desarrollar programas Java y proporciona el entorno de ejecución necesario para ejecutar dichos programas.
- Apache Jakarta Tomcat. Contenedor de componentes Web.
- Framework Strut. Paquete de clases que ayudan a implementar el patrón arquitectónico MVC sobre tecnología Java.

Código fuente lógica de negocio

Class Funcionalidad

```
java.lang.Object
   |
   +--Funcionalidad
```

```
package com.trio.dealer;
public class Funcionalidad {
                        private String nombreFuncionalidad;
                        private String urlFuncionalidad;

                        // Cosntructor
                        public Funcionalidad (String nombre, String url){
                            this.nombreFuncionalidad = nombre.trim();
                            this.urlFuncionalidad = url.trim();
                        }

                        public String getNombreFuncionalidad(){
                            return this.nombreFuncionalidad;
                        }

                        public String getUrlFuncionalidad(){
                            return this.urlFuncionalidad;
                        }
}
```

Class Rol

```
java.lang.Object
   |
   +--Rol
```

```
package com.trio.dealer;
import java.util.*;
import java.sql.*;
import com.trio.dealer.*;
import com.trio.servios.postgre.*;

public class Rol {
      private List funcionalidades = new ArrayList();
      private int posicion;
      private int cantidad;

      // Constructor
      public Rol(int cr){
         // Conexión a base de datos para extraer funcionalidades de Rol
        PgConexionBean dbConexion = new PgConexionBean();
        String sql = "SELECT * FROM triv_func_rol WHERE codi_rol="'+ cr +"'";
            try {
                ResultSet rs = dbConexion.ejecutarConsulta(sql);
                dbConexion.close();
                while (rs.next()) { // Carga funcionalidades
                    this.agregarFuncion( new
                        Funcionalidad(rs.getString("desc_menu"),rs.getString("url_funcionalidad")));
                };
              // set valores según llenado de funciones
                this.cantidad = funcionalidades.size();
                this.posicion = -1;
            }  catch (SQLException e) {
            }
```

```
}
        private void agregarFuncion(Funcionalidad f){
            funcionalidades.add(f);
        }

        public Funcionalidad getFuncionalidad(){
                return (Funcionalidad)funcionalidades.get(posicion);
        }

        public boolean hayMas() {
                if (this.posicion<this.cantidad-1) {
                    this.posicion++;
                    return true;
                } else {
                            return false;
                            }
        }

        public void inicio(){
                this.posicion = -1;
        }

        public int cantidad(){
                return this.cantidad;
        }

        private void correPosicion(){
                this.posicion++;
        }
    }
```

Class Dealer

java.lang.Object
   |
   +--**Dealer**

```
package com.trio.dealer;

import java.sql.*;
import java.util.*;
import java.text.*;
import com.trio.*;
import com.trio.servios.postgre.*;

public class Dealer {
    private String nombreUsuario;
    private String password;
    private int  intentosLogin;
    public final int MAX_INTENTOS = 3;
    private boolean estaAutentificado;
    private String  nombreCompleto;
    private String empresa;
    private int  codigoEmpresa;
    private String zonalUsuario;
    private String  codigoLocalidad;
    private String localidad;
    private int codigoRol;
    private Rol rolDealer;
    private boolean bloqueado;
    private int codiUsuario;
    private String fechaInicioSesion;
```

```
    // Constructor
     public Dealer(String u, String p){
            this.nombreUsuario = u.trim();
            this.password = p.trim();
            this.intentosLogin = 0;
            this.empresa = null;
            this.codigoEmpresa = -1;
            this.zonalUsuario = null;
            this.codigoLocalidad = null;
            this.localidad = null;
            this.codigoRol = -1;
            this.estaAutentificado = false;
            this.nombreCompleto = null;
            this.bloqueado = false;
            this.codiUsuario = -1;
            this.fechaInicioSesion = null;
    }

// Método principal, autentifica contra base de datos y carga rol
public void login() {
      // Aumenta intentos en uno
         intentosLogin++;
       // Consulta a base de datos
         PgConexionBean dbConexion = new PgConexionBean();
         String sql = "SELECT * FROM  triv_usuarios WHERE ";
                  sql+= "logi_usuario='" + nombreUsuario;
                  sql+="' and pass_usuario='"+password+"'";
           try {
                   ResultSet rs = dbConexion.ejecutarConsulta(sql);
                   dbConexion.close();

                   if (rs.next()) {
                      this.codiUsuario = rs.getInt("codi_usuario");
                      this.empresa = rs.getString("nomb_empresa");
                      this.codigoEmpresa = rs.getInt("codi_empresa");
                      this.zonalUsuario = rs.getString("codi_admzonal");
                      this.codigoLocalidad = rs.getString("codi_localida");
                      this.localidad = "Valdivia";
                      this.nombreCompleto = rs.getString("nomb_usuario");
                      this.rolDealer = new Rol(rs.getInt("codi_rol"));
                      this.estaAutentificado = true;
                      this.registraIngreso();
                      if ((rs.getString("flag_bloqueo")).equals("S")) {
                           this.bloqueado= true;
                      }
                    }

                 } catch (SQLException e) {  }

       }
```

```
// Defición de métodos SET

public void setNombreUsuario(String u){
    this.nombreUsuario = u.trim();
}

public void setPassword(String p){
   this.password = p.trim();
}

// Definición de métodos GET
   public String getNombreUsuario(){
   return this.nombreUsuario;
}

public String getPassword(){
   return this.password;
}

public int getIntentosLogin(){
   return this.intentosLogin;
}

public String getEmpresa(){
   return this.empresa;
}

public int getCodigoEmpresa(){
   return this.codigoEmpresa;
}

public String getZonalUsuario(){
return this.zonalUsuario;
}

public String getCodigoLocalidad(){
return this.codigoLocalidad;
}

public String getLocalidad(){
return this.localidad;
}

public Rol getRolDealer(){
return this.rolDealer;
}

public boolean estaAutentificado(){
return this.estaAutentificado;
}

public String getNombreCompleto(){
return this.nombreCompleto;
}

public boolean estaBloqueado(){
  return this.bloqueado;
}

public void registraIngreso(){
   // Log Registra ingreso a sistema
   PgConexionBean dbConexion = new PgConexionBean();
   String sql = "INSERT INTO trit_login_log VALUES ('"+this.codiUsuario;
   sql+= "', to_date('"+ this.getFechaActual()+"','dd-mm-yyyy'))";
```

```
   try {
          dbConexion.ejecutar(sql);
          dbConexion.close();
    } catch (SQLException e) { }
}


public void bloqueaCuenta() {
   PgConexionBean dbConexion = new PgConexionBean();
  String sql = "UPDATE trit_usuarios SET flag_bloqueo = 'S'";
           sql+= " WHERE logi_usuario = '" + this.nombreUsuario + "'";

  try {
        dbConexion.ejecutar(sql);
        dbConexion.close();
      } catch (SQLException e) { }
  }

public String getFechaActual() {
     // Formato de salida para fecha
    SimpleDateFormat formatoFecha;
    String patron= "dd-MM-yyyy";
    Locale local =  new Locale("es","ES");
    formatoFecha = new SimpleDateFormat(patron, local);

    // Obtencion de fecha actual
    java.util.Date fecha = new java.util.Date();
    return formatoFecha.format(fecha);
 }
}
```

Código fuente MVC

Class LoginForm

```
java.lang.Object
   |
   +--ActionForm
          |
          +--LoginForm
```

```
package com.trio.dealer.struts;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import com.trio.*;
/**
 * LoginActionForm
 *
 * @author Cristian Porflitt
 * @version 1.0
 */
public class LoginForm extends ActionForm {
    // ----------------------------------------------------- Instance Variables
    /** Nombre de usuario */
    private String nombre = null;
    /** Contraseña usuario */
    private String password = null;
    // ----------------------------------------------------------- Constructors

    /**
     * Constructor
     */
    public LoginForm() {
        super();
    }

    // --------------------------------------------------------- Public Methods
    public void reset(ActionMapping mapping, HttpServletRequest request) {
        this.nombre = null;
        this.password = null;
    }

    /**
     * Valida las propiedades de los datos que han sido enviados desde el formulario vía HTTP request,
     */
    public ActionErrors validate(  ActionMapping mapping,
                                   HttpServletRequest request) {
                                   ActionErrors errors = new ActionErrors();

        // Nombre debe ser ingresado
        if ((nombre == null) || (nombre.length() < 1)) {
            errors.add("nombre", new ActionError("login.nombre.requerido"));
        }

        // Password debe ser ingresado
        if ((password == null) || (password.length() < 1)) {
            errors.add("password", new ActionError("login.password.requerido"));
        }
```

```
        return (errors);
    }


// ------------------------------------------------------------------ Properties
    public String getNombre() {
        return nombre;
    }

    public String getPassword() {
        return password;
    }

    public void setNombre(String n) {
        this.nombre = n;
    }

      public void setPassword(String p) {
        this.password = p;
    }
}
```

Class LoginAction

```
java.lang.Object
   |
   +--Action
          |
          +--LoginAction
```

```
package com.trio.dealer.struts;

import java.util.Locale;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServletResponse;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.util.ModuleException;
import org.apache.struts.util.MessageResources;
import org.apache.commons.beanutils.PropertyUtils;
import com.trio.dealer.*;

/**
 * Recupera y procesa los datos encapsulados en LoginForm
* @author Cristian Porflitt
 * @version 1.0
 */
public class LoginAction extends Action {

// ------------------------------------------------------------- Constructors
   /**
    * Constructor para LoginAction.
   */
     public LoginAction() {
       super();
     }

   // ------------------------------------------------------------- Action Methods
   /**
    * Procesa la solicitud y retorna una instancia de ActionForward
    * describiendo donde y cómo el control será traspasado,
    */
   public ActionForward execute(
       ActionMapping mapping,
       ActionForm form,
       HttpServletRequest request,
       HttpServletResponse response)
       throws Exception {
                                 // captura los datos desde formulario, usuario y password
                                 Locale locale = getLocale(request);
                                 String nombreDealer = (String)
                                 PropertyUtils.getSimpleProperty(form, "nombre");
                                 String passwordDealer = (String)
                                 PropertyUtils.getSimpleProperty(form, "password");

// ------------------------------  Recupera variable sesion
                                 HttpSession session = request.getSession();

// ------------------------------ Crear objeto usuario si no existe en session
```

```
                                    if (session.getAttribute("usuario")==null){
                                          session.setAttribute("usuario",new Dealer("x","y"));
                                    };

                                    Dealer dealer = (Dealer)session.getAttribute("usuario");


 if (dealer.getIntentosLogin()>=dealer.MAX_INTENTOS) {  // Valida máximo de intentos
                                                      dealer.bloqueaCuenta();
                                                      ActionErrors errors = new ActionErrors();
                                                      errors.add("maxIntentos", new
                                                      ActionError("login.error.maxintentos"));
                                                      saveErrors(request, errors);
                                                      return (mapping.getInputForward()); // vuelve a formulario origen
                                                } else {
                                                      dealer.setNombreUsuario(nombreDealer);
                                                      dealer.setPassword(passwordDealer);
                                                      dealer.login();
                                                      session.setAttribute("usuario", dealer);

                                                      // usuario no valido
                                                       if (!dealer.estaAutentificado()){
                                                            ActionErrors errors = new ActionErrors();
                                                            errors.add("invalido", new ActionError("login.invalido"));
                                                            saveErrors(request, errors);
                                                            // vuelve a formulario origen
                                                             return (mapping.getInputForward());             };

                                                 // caso usuario autentificado y no tiene cuenta bloqueada
                                                 if (dealer.estaAutentificado()&&(!dealer.estaBloqueado())) {
                                                       return mapping.findForward("inicio");
                                                   } else {
                                                            // usuario autentificado pero bloqueado
                                                             ActionErrors errors = new ActionErrors();
                                                             errors.add("password", new
                                                                                ActionError("login.error.bloqueado"));
                                                            saveErrors(request, errors);
                                                            return (mapping.getInputForward()); // vuelve a formulario origen
                                                 }
                                                }
    }

}
```