PONTIFICIA UNIVERSIDAD CATÓLICA DEL ECUADOR

SEDE AMBATO

ESCUELA DE INGENIERÍA DE SISTEMAS

DISERTACIÓN DE GRADO PREVIA LA OBTENCIÓN DEL TÍTULO DE INGENIERO DE SISTEMAS

"Sistemas Distribuidos Multicapa con Java sobre Linux"

Juan Carlos Freire Naranjo

DIRECTOR DE DISERTACIÓN: Ing. Patricio Medina

AMBATO, 2002

Índice de contenidos

Introducción 1
Capítulo 1 4
Computación cliente/servidor 4
1.1 Antecedentes 4
1.2 Cliente 5
1.3 Servidor 5
1.4 Características de la computación cliente/servidor 6
1.5 Arquitecturas cliente/servidor 6
1.5.1 Arquitectura de red 7
1.5.2 Arquitectura de aplicación 8
1.6 Arquitecturas de aplicaciones cliente/servidor 9
1.6.1 Arquitecturas de dos capas 9
1.6.1.1 Arquitectura de cliente principal (o grueso) 9
1.6.1.2 Arquitectura de servidor principal (o grueso) 10
1.6.1.3 Desventajas de las arquitecturas de dos capas 11
1.6.2 Arquitectura de tres capas 11
1.6.3 Arquitectura de múltiples capas 13
1.6.3.1 Factores clave 13
1.6.3.2 Una arquitectura multicapa basada en servicios 15
1.6.4 Ventajas de una arquitectura multicapa 17
1.7 Componentes de servidor 20
1.7.1 Servidores de aplicaciones 21
1.7.2 Componentes y contenedores 21
Capítulo 2 23
Java Empresarial 23
2.1 La tecnología Java 23
2.1.1 Origen 23
2.1.2 Elementos 24
2.1.2.1 El lenguaje de programación Java 24
2.1.2.2 La plataforma Java 27
2.1.3 ¿Qué puede hacer la tecnología Java? 28
2.2 Java para aplicaciones empresariales 29
2.2.1 Factores clave 30
2.3 Arquitectura de la plataforma 32
2.3.1 Servicios 33
2.4 Elementos de Java Empresarial 36
2.4.1 Invocación de métodos remotos: Java RMI 37
2.4.2 Interoperabilidad con CORBA: Java IDL 38
2.4.2.1 Sobre CORBA 39
2.4.2.2 Sobre Java IDL 39
2.4.3 Acceso a bases de datos: JDBC 40
2.4.4 Servicios de Nombres y Directorios: JNDI 41
2.4.5 Procesamiento de transacciones: JTS 42
2.4.6 Middleware orientado a mensajes: Servicio Java de Mensajes 42
2.4.7 Servicios Web basados en Java: JSP y Java Servlet 43
2.4.7.1 Páginas Java de Servidor (JSP) 43
2.4.7.2 Java servlets 44

2.4.8 Servicios de seguridad: la API de Seguridad Java 45
2.4.9 XML y Java 47
2.4.9.1 APIs Java para XML 48
Capítulo 3 52
La arquitectura de Componentes Empresariales Java 52
3.1 Visión general 52
3.2 Servicios implícitos 53
3.3 Detalles arquitectónicos 53
3.3.1 Objetos transitorios y persistentes 54
3.3.1.1 Componentes de sesión 54
3.3.1.2 Componentes entidad 55
3.3.2 Formas de interacción 55
3.3.2.1 EJBHome 56
3.3.2.2 EJBObject 56
3.3.3 Atributos declarativos 57
3.3.4 Objeto de contexto 57
3.3.5 Protocolos de comunicación 57
3.3.6 Administración del estado 58
3.3.7 Servicios de persistencia 58
3.3.7.1 Persistencia Manejada por el Componente, BMP 59
3.3.7.2 Persistencia Manejada por el Contenedor, CMP 59
3.3.8 Administración de transacciones 59
3.3.9 Seguridad 60
3.4 Roles y ciclo de vida de la aplicación 60
3.4.1 Proveedor de componentes empresariales 61
3.4.2 Ensamblador de aplicaciones 61
3.4.3 Publicador 61
3.4.4 Proveedores de servidor y contenedor EJB 61
3.4.5 Administrador del sistema 62
3.5 La Interfaz de Programación de Aplicaciones EJB 62
3.5.1 Desarrollo de Componentes de Sesión 63
3.5.1.1 La interfaz local 64
3.5.1.2 La interfaz remota 64
3.5.1.3 La clase del componente empresarial 64
3.5.2 Desarrollo de Componentes Entidad 67
3.5.2.1 La interfaz local 68
3.5.2.2 La interfaz remota 70
3.5.2.3 La clase de clave primaria 70
3.5.2.4 La clase del componente empresarial 71
3.5.2.5 Escritura de operaciones de acceso a bases de datos para persistencia manejada por el componente 75
3.5.2.6 Configuración de acceso a bases de datos para persistencia manejada por el contenedor 78
3.5.3 Comportamiento en transacciones 79
3.5.3.1 Administración declarativa de la transacción 79
3.5.3.2 Administración de transacciones distribuidas 82
3.5.4 El Entorno del Componente Empresarial 84
3.5.5 Definición del descriptor de publicación 85
3.5.6 Empacado del componente 88
3.6 Ensamblaje de aplicaciones 88

3.6.1 Ensamblaje de componentes en el servidor 88
3.6.2 Desarrollo de aplicaciones cliente 89
3.7 Infraestructura de seguridad EJB 91
3.7.1 Modelo de seguridad declarativo EJB 1.1 91
3.7.2 Configuración de la seguridad declarativa 91
3.7.2.1 ¿Qué es un rol? 91
3.7.3 Mejora de la seguridad EJB 94
Capítulo 4 97
Entorno Linux para Java Empresarial 97
4.1 Breve historia de Linux 97
4.2 Características de Linux 98
4.3 Linux como plataforma para Java 99
4.3.1 Sobre el desempeño 100
4.4 Elementos del entorno Java Empresarial sobre Linux 100
4.4.1 Paquete de desarrollo Java 100
4.4.1.1 Sun Java 2 Edición Estándar 101
4.4.1.2 Kaffe 102
4.4.2 Soporte para Java servlets 102
4.4.2.1 Apache JServ 103
4.4.3 Soporte para páginas Java de servidor 103
4.4.3.1 Apache Jakarta Tomcat 103
4.4.4 Bases de datos para persistencia 106
4.4.4.1 PostgreSQL 106
4.4.5 Servidores para componentes empresariales Java 108
4.4.5.1 Sun Java 2 Edición Empresarial 108
4.4.5.2 JBoss 109
4.4.5.3 Bullsoft JOnAS EJB 121
4.4.6 Otras herramientas de desarrollo 124
4.4.6.1 NetBeans 124
4.4.6.2 Ant 125
4.4.6.3 Merlot 125
4.4.6.4 Poseidón para UML 126
Capítulo 5 128
Prototipo de Sistema Financiero 128
5.1 Planificación del proyecto 128
5.1.1 Objetivos 128
5.1.2 Funciones 129
5.1.3 Riesgos del proyecto 129
5.1.3.1 Riesgos identificados 129
5.1.4 Recursos del proyecto 129
5.1.4.1 Recursos hardware 129
5.1.4.2 Recursos software 130
5.2 Metodología de desarrollo de software 130
5.3 Resumen del análisis orientado a objetos 131
5.3.1 Casos de uso de alto nivel 132
5.3.2 Casos de uso expandidos 134
5.3.2.1 Caso de uso: Abrir cuenta 134
5.3.2.2 Caso de uso: Depositar dinero en efectivo 135
5.3.2.3 Caso de uso: Retirar dinero en efectivo. 136
5.3.3 Modelo Conceptual 138

5.3.3.1 Clases del módulo Servidor 138
5.3.3.2 Clases del módulo Cliente 140
5.3.3.3 Clases del Módulo Administración 143
5.3.4 Especificación del comportamiento de los módulos 143
5.3.4.1 Caso de uso: Abrir cuenta 143
5.3.4.2 Caso de uso: Depositar dinero y Retirar dinero 145
5.4 Resumen del diseño orientado a objetos 147
5.4.1 Diagramas de interacción 147
5.4.2 Componente de interacción humana 149
5.4.2.1 Jerarquía de ordenes 149
5.4.2.2 Interacción detallada 150
5.5 Resumen de la implementación 156
5.5.1 Distribución de las clases de la implementación en paquetes 156
5.5.2 Resumen de clases por paquete 157
Conclusiones 164
Recomendaciones 166
Bibliografía 168
Anexo A 171
Glosario 183

Listado de tablas

Tabla 2.1: Métodos para asegurar sistemas 46
Tabla 3.1: Efecto de los atributos declarativos para transacciones 80
Tabla 3.2: Nombre de la clase y el protocolo de acceso JDBC a PostgreSQL 107
Tabla 5.1: Menú de ordenes para la aplicación Cliente 150
Tabla 5.2: Menú de ordenes para la aplicación de Administración 150
Tabla 5.3: Los paquetes Java que implementan el Sistema Financiero 156
Tabla 5.4: Clases del paquete edu.pucesa.financiero.admin 157
Tabla 5.5: Clases del paquete edu.pucesa.financiero.admin.gui 158
Tabla 5.6: Clases del paquete edu.pucesa.financiero.admin.utils 158
Tabla 5.7: Clases del paquete edu.pucesa.financiero.beans 159
Tabla 5.8: Clases del paquete edu.pucesa.financiero.cliente 160
Tabla 5.9: Clases del paquete edu.pucesa.financiero.cliente.gui 160
Tabla 5.10: Interfaces del paquete edu.pucesa.financiero.cliente.utils 161
Tabla 5.11: Clases del paquete edu.pucesa.financiero.cliente.utils 161
Tabla 5.12: Clases del paquete edu.pucesa.financiero.data 161
Tabla 5.13: Interfaces del paquete edu.pucesa.financiero.interfaces 162
Tabla 5.14: Interfaces del paquete edu.pucesa.financiero.utils 162

Listado de figuras

Figura 1.1: Arquitectura de cliente principal ... 9
Figura 1.2: Arquitectura de servidor principal ... 10
Figura 1.3: Arquitectura de tres capas ... 12
Figura 1.4: Ejemplo de una arquitectura de cuatro capas ... 15
Figura 2.1: Flujo del código Java hasta su ejecución ... 25
Figura 2.2: Independencia de la plataforma de un programa Java ... 26
Figura 2.3: Colocación de la plataforma Java dentro del entorno de ejecución ... 27
Figura 2.4: Componentes del SDK Java 2, versión 1.3 ... 28
Figura 2.5: Elementos de una arquitectura empresarial Java ... 29
Figura 2.6 Arquitectura de J2EE ... 33
Figura 2.7: Componentes de la plataforma Java Empresarial ... 36
Figura 2.8: Ejemplo de interacción entre cliente y servidor usando RMI ... 37
Figura 2.9: Ejemplo del uso de JDBC para acceder a un Servidor de Base de Datos ... 40
Figura 2.10: Elementos del Servicio de Nombres y Directorios ... 41
Figura 2.11: Ejemplo de uso de servlet ... 45
Figura 3.1: Esquema de la arquitectura EJB ... 54
Figura 3.2: Interacción entre un cliente y un contenedor EJB ... 56
Figura 3.3: Roles de la arquitectura EJB ... 60
Figura 3.4: Interacción de los componentes de la API EJB ... 63
Figura 3.5: Los cuatro casos para disponer del Vigilante de Transacciones ... 83
Figura 5.1: Componentes principales del Sistema Financiero ... 132
Figura 5.2: Diagrama de casos de uso del Sistema Financiero ... 133
Figura 5.3: Diagrama de clases inicial del Módulo Servidor ... 138
Figura 5.4: El componente Socio ... 139
Figura 5.5: El componente Cuenta ... 139
Figura 5.6: El componente Transacción ... 140
Figura 5.7: El componente Semilla ... 140
Figura 5.8: Las clases del módulo Cliente y su interacción con el Servidor ... 142
Figura 5.9: La clase principal del módulo Administración ... 143
Figura 5.10: Diagrama de secuencia del sistema para el caso de uso *Abrir cuenta* ... 143
Figura 5.11: Diagrama de secuencia del sistema para los caso de uso *Depositar dinero* y *Retirar dinero* ... 145
Figura 5.12: Diagrama de colaboración para obtenerNumeroSocio ... 147
Figura 5.13: Diagrama de colaboración para crearSocio ... 148
Figura 5.14: Diagrama de colaboración para crearCuenta ... 148
Figura 5.15: Diagrama de colaboración para depositar ... 149
Figura 5.16: Diagrama de colaboración para registrarTransaccion ... 149
Figura 5.17: Ventana de acceso al módulo cliente ... 150
Figura 5.18: Ventana para ingreso de un nuevo socio ... 151
Figura 5.19: Ventana para la consulta del saldo de una cuenta ... 151
Figura 5.20: Ventana para la lista de las transacciones del Cajero ... 152
Figura 5.21: Ventana para registrar la transacción de depósito ... 152
Figura 5.22: Ventana para el registro de una transacción de retiro ... 153
Figura 5.23: Ventana para especificar los parámetros de conexión del módulo de administración ... 153
Figura 5.24: Ventana para administración de roles y usuarios ... 154
Figura 5.25: Ventana para registro de roles de la aplicación ... 155

Figura 5.26: Ventana para edición de usuarios del sistema... 155
Figura 5.27: Ventana para registro de valores iniciales... 156

Tabla de abreviaturas

AOO	Análisis orientado a objetos
API	Interfaz de programación de aplicaciones
ASP	Página activa de servidor
BMP	Persistencia manejada por el componente
CIH	Componente de Interacción Humana
CMP	Persistencia manejada por el contenedor
DBMS	Sistema manejador de bases de datos
DNS	Sistema de nombres de dominio
DOO	Diseño orientado a objetos
FTP	Protocolo de transferencia de archivos
GNU	GNU's Not Unix
HTML	Lenguaje de marcas de hipertexto
HTTP	Protocolo de transferencia de hipertexto
HTTP-S	Versión segura del HTTP
IDE	Entorno de desarrollo integrado
IDL	Lenguaje de definición de interfaces
IIOP	Protocolo Internet InterORB
IP	Protocolo Internet
JMS	Servicio Java de mensajes

JNDI	Interfaz Java para nombres y directorios
JRMP	Protocolo Java para métodos remotos
JSP	Página Java de servidor
JTM	Administrador de transacciones Java
JTS	Servicio de transacciones Java
LAN	Red de área local
LDAP	Protocolo ligero para acceso a directorios
NDS	Servicio Novell de directorios
NIS	Servicio de información para red
OTS	Servicio de transacciones de objetos
RAM	Memoria de acceso aleatorio
RDBMS	Sistema manejador de bases de datos relacionales
RPC	Llamada de procedimiento remoto
SDK	Paquete de desarrollo de software
SQL	Lenguaje estructurado de consultas
TI	Tecnología de información
WAN	Red de área ancha

Introducción

Las empresas de hoy necesitan extender su alcance, reducir sus costos, y disminuir sus tiempos de respuesta proporcionando servicios de fácil acceso a sus clientes, socios, empleados y proveedores.

Generalmente, las aplicaciones que proporcionan estos servicios deben combinar sistemas de información ya existentes con nuevas funciones del negocio que brindan servicios a un amplio rango de usuarios. Estos servicios tienen que ser:

- *Disponibles de forma constante*, para satisfacer las necesidades del mundo globalizado de hoy.
- *Seguros*, para proteger la privacidad de los usuarios y la integridad de los datos empresariales.
- *Confiables y con capacidad de crecer*, para asegurar que las transacciones del negocio son procesadas apropiada y prontamente.

Aquí surge la cuestión, ¿cómo y con qué herramientas se han de implementar estos servicios? El estado actual de la tecnología fomenta que estos servicios sean creados como aplicaciones distribuidas que consisten de múltiples capas, incluyendo una capa cliente para la presentación, otra capa con los recursos de datos, y una o más capas intermedias entre las dos donde se realiza la mayoría del trabajo de la aplicación. La capa intermedia implementa los servicios que integran los sistemas existentes con las funciones y los datos del nuevo servicio. Las posibilidades son muy variadas dentro del abanico de opciones que existe para elegir las herramientas a utilizarse en la producción y puesta en marcha de éstas aplicaciones distribuidas.

Ésta disertación se propuso para demostrar que era viable el desarrollo y puesta en funcionamiento de aplicaciones de nivel empresarial con herramientas de bajo costo. Para ello se eligió al sistema operativo Linux, reconocido por sí solo como estable y seguro. El lenguaje de programación escogido fue Java, y más concretamente la plataforma Java Empresarial, que proporciona los elementos necesarios para crear las aplicaciones distribuidas con los servicios mencionados.

Específicamente, los objetivos de ésta disertación fueron:

- Conocer la arquitectura de software basada en Java Empresarial para sistemas cliente/servidor multicapa.
- Determinar los elementos a configurar en un entorno empresarial de Java sobre Linux.
- Establecer cómo asegurar el entorno empresarial Java sobre Linux.
- Evaluar la capacidad de Java sobre Linux para desarrollar aplicaciones inherente o susceptiblemente capaces de tener una interfaz intranet/internet.
- Establecer las características de diseño que deben satisfacer los componentes de software del lado servidor.
- Implementar un Prototipo de Sistema Financiero para una Cooperativa de Ahorro y Crédito como ejemplo demostrativo de una aplicación multicapa basada en componentes.
- Utilizar las mejores herramientas de código abierto disponibles para la implementación del Prototipo.

Mucha parte del trabajo para la consecución de los objetivos se ha hecho buscando información en la Internet. La investigación bibliográfica ha servido fundamentalmente para iluminar aspectos relativos a la arquitectura Java Empresarial y la forma en que se desarrollaría el Prototipo.

Los resultados de ésta disertación se plasman dentro de los capítulos siguientes. En el primero, se busca definir las características de la computación cliente/servidor y analizar sus variantes de implementación.

Los capítulos segundo y tercero están concentrados en la tecnología Java. Concretamente, el segundo examina todos los elementos que conforman la plataforma Java Empresarial, realizando una evaluación de sus características y contrastándolas con las necesidades de una organización contemporánea.

El tercer capítulo se enfoca sobre lo que es el elemento más importante de la plataforma Java Empresarial, los Componentes Empresariales Java. Se detalla qué son, para qué sirven, cómo se definen y cómo encajan dentro del esquema empresarial.

El cuarto capítulo está dedicado a Linux y se ocupa principalmente de proporcionar indicaciones sobre cuáles herramientas son las requeridas para formar un entorno empresarial que acoja a las aplicaciones distribuidas creadas con Java. Se brindan guías sobre instalación y configuración de todas esas herramientas.

El quinto capítulo recoge la documentación producida durante el desarrollo del Prototipo de un Sistema Financiero para una Cooperativa de Ahorro y Crédito. Este prototipo se desarrolló para mostrar, sobre un sistema real, cómo se ha de crear una aplicación distribuida usando Java Empresarial. El capítulo recoge los artefactos más importantes producidos durante el ciclo de desarrollo.

Capítulo 1

Computación cliente/servidor

1.1 Antecedentes

A través del tiempo los programas para computadora han migrado a varios modelos arquitectónicos. Las primeras aplicaciones eran completamente independientes. Luego algunas necesitaron manipular datos y para lograrlo tomaron uno de dos caminos; en el primero, enlazaron un conjunto de bibliotecas de base de datos, las cuales les permitían manipular archivos locales de datos, en el segundo, ciertamente eran bases de datos.

Al evolucionar las redes de computadoras, básicamente para compartir impresoras y archivos, las personas comenzaron a pensar en compartir los archivos de bases de datos. Si bien éste enfoque funcionó, el principal problema era que solamente una persona podía acceder en determinado momento al archivo de datos. Para posibilitar el compartir los archivos, las bibliotecas de base de datos cambiaron a programas independientes que arbitraban las solicitudes en la red. Esta forma de desarrollo, caracterizada por un servidor de bases de datos independiente y un cliente que hace todo el procesamiento, fue conocida como **cliente/servidor**.

Inicialmente, los servidores de un ambiente cliente/servidor eran relativamente *puros*, tomaban una solicitud de datos, recuperaban los datos y los enviaban de vuelta al cliente. Conforme aumentaron los programas para estos servidores se les exigía más y más trabajo, hasta el momento en que la lógica del negocio se movió al servidor (por ejemplo, a una base de datos como procedimientos almacenados).

En éste momento ocurrieron dos eventos.

- Emergió un fuerte fundamento teórico en favor del buen diseño del código. Este enfatizó las ventajas prácticas de diseñar código en tres capas diferentes: la interfaz del usuario, una lógica reutilizable del negocio y las fuentes de datos. Particularmente en un mundo orientado a objetos, el separar las tres capas (aún si la aplicación termina

como un solo archivo ejecutable) hace a las aplicaciones más fáciles de escribir y más fáciles de mantener, y

- Apareció una clase de herramientas que posibilitaba a las aplicaciones ser distribuidas a través de una red. Estas herramientas también hacían posible la separación física de la interfaz del usuario de un servidor de lógica del negocio, junto con una base de datos o conjunto de base de datos diferente.

Esencialmente, éste modelo dejó que los desarrolladores crearan servicios personalizados de negocios que se compartían, los cuales se comportaban como la base de datos en el mundo cliente/servidor. Este vino a ser conocido como el modelo de tres capas. Cuando estos servicios de nivel intermedio comenzaron a ponerse uno sobre otro, el modelo de **tres capas** se expandió a una arquitectura de **n capas** (o de **múltiples capas**) con servicios que se comunican con otros servicios.

1.2 Cliente

El cliente es un **proceso** (o *programa*) que envía un mensaje a un proceso servidor, solicitando que el servidor ejecute una tarea (o *servicio*).

Los programas clientes usualmente manejan la porción de interfaz de usuario de la aplicación, validan los datos ingresados por el usuario, despachan solicitudes hacia programas servidores y algunas veces ejecutan la lógica del negocio.

El proceso del lado del cliente es la parte frontal de la aplicación que el usuario ve y con la cual actúa. Además maneja los recursos locales que utiliza el usuario como monitor, teclado, procesador y periféricos.

1.3 Servidor

Un proceso servidor cumple con la petición del cliente ejecutando la tarea solicitada. Los programas servidores generalmente reciben peticiones de programas clientes, ejecutan consultas y actualizaciones de base de datos, manejan la integridad de los datos y despachan respuestas a las solicitudes de clientes.

El proceso del lado del servidor puede ejecutarse en la misma máquina que está el cliente o sobre otra máquina en la red.

El servidor actúa como un motor de software que maneja recursos compartidos tales como bases de datos, impresoras, enlaces de comunicación o procesadores de altas prestaciones.

El proceso servidor desarrolla entre telones las tareas que son comunes para aplicaciones similares.

1.4 Características de la computación cliente/servidor

Las características básicas de la computación cliente/servidor son:

- Combinación de una porción cliente o frontal que actúa con el usuario y una porción servidor o trasera que actúa con el recurso compartido.
- El cliente y el servidor tienen diferentes requerimientos para los recursos como velocidades de procesador, memoria, velocidades y capacidades de disco y dispositivos de entrada/salida.
- El entorno cliente/servidor generalmente es heterogéneo y de múltiples vendedores. La plataforma hardware y el sistema operativo del cliente y del servidor usualmente no son las mismas. Los procesos cliente y servidor se comunican a través de un conjunto bien definido de interfaces de programación de aplicaciones (**APIs**) y llamadas de procedimientos remotos (**RPCs**) estándares.
- Una característica importante de los sistemas cliente/servidor es la **escalabilidad**. Pueden ser escalados horizontal o verticalmente. *Escalado horizontal* significa agregar o remover estaciones de trabajo cliente con tan solo un pequeño impacto en el desempeño. *Escalado vertical* significa la migración a una máquina servidor más grande o más rápida o a varios servidores.

1.5 Arquitecturas cliente/servidor

Una fuente de confusión es que el término “cliente/servidor” se utiliza tanto para describir una arquitectura de diseño de redes (*arquitectura de red*) y una arquitectura de diseño de programas (*arquitectura de aplicación*). Ambas están relacionadas pero a veces se utilizan de forma indistinta.

1.5.1 Arquitectura de red

En el contexto del diseño de redes, el término "arquitectura cliente/servidor" describe un modelo para distribuir el trabajo de ejecutar tareas de cómputo entre las varias computadoras conectadas a una red.

En este sentido, el término "arquitectura cliente/servidor" es utilizado para describir uno de tres modelos básicos de redes:

- La **arquitectura servidor/terminal** describe un tipo de red en el cual una computadora central (el servidor) está conectada a cierto número de estaciones de trabajo (las terminales) y en el cual el servidor maneja todo el procesamiento. Las terminales son utilizadas para ingresar datos al servidor y para revisar reportes, pero son "*tontas*" en el sentido de que ellas no participan en el trabajo de procesamiento. La arquitectura servidor/terminal es apropiada para redes grandes y pequeñas.

  La característica distintiva es que una sola computadora central maneja todo el trabajo de procesamiento.

- La **arquitectura cliente/servidor** describe una red dentro de la cual cada computadora maneja parte del trabajo de procesamiento y que toma el papel de "cliente" o de "servidor" con respecto a cada proceso. Los "servidores" son computadoras centrales compartidas las cuales están dedicadas a manejar tareas específicas para un número de clientes; por ejemplo, los "servidores de archivos" están dedicados al almacenamiento de archivos. Los "clientes" son estaciones de trabajo en las cuales los usuarios ejecutan programas y aplicaciones. Los clientes dependen de los servidores para recursos como archivos, dispositivos y hasta capacidad de procesamiento, pero funcionan independientemente de los servidores. La arquitectura cliente/servidor puede ser usada en redes de cualquier tamaño.

  La característica particular de ésta es que todas las computadoras de la red participan del procesamiento, pero ciertas computadoras están dedicadas a servicios o tareas específicas y no hacen otro trabajo.

- La **arquitectura de igual a igual** describe un tipo de red en el cual un número de estaciones de trabajo están conectadas juntas, pero en el que ninguna computadora está designada como un "servidor". En su lugar, cada estación puede ser (y usualmente es)

tanto "cliente" como "servidor", dependiendo de la tarea involucrada. Por ejemplo, una estación puede estar conectada a una impresora y actuar como un "servidor de impresión" para otras estaciones. Las redes de igual a igual son usualmente pequeñas (menos de 25 estaciones) y no ofrecen desempeño sólido en instalaciones grandes o bajo redes de tráfico pesado.

1.5.2 Arquitectura de aplicación

El término "arquitectura cliente/servidor" es también utilizado para describir un tipo de diseño de aplicaciones, muy relacionado y dependiente de la arquitectura de red cliente/servidor. Dentro del contexto de diseño de aplicaciones, el término "arquitectura cliente/servidor" describe un modelo para distribuir el trabajo de procesamiento de una sola aplicación entre varias computadoras conectadas a una red.

En éste sentido, el término "arquitectura cliente/servidor" es usado para describir uno de tres modelos básicos de diseño de aplicaciones:

- Las **aplicaciones de servidor** son programas y procesos que están diseñados para operar sobre una computadora central. Las terminales son usadas para acceder a los programas o procesos, para ingresar datos y para revisar reportes, pero todas las tareas de procesamiento tienen lugar en una única computadora central. El procesamiento no está distribuido sino centralizado.
- Las **aplicaciones independientes** son programas y procesos que están diseñados para operar sobre una estación de trabajo individual. A ésta categoría pertenecen ciertas aplicaciones que pueden ser instaladas y usadas sobre una red pero en las que el procesamiento es independiente. Por ejemplo, una aplicación de procesamiento de texto instalada en un servidor y ejecutada en una estación conectada a la red. El procesamiento no está distribuido sino que es local y tiene lugar en la estación de trabajo.
- Las **aplicaciones cliente/servidor** son programas o procesos que dividen el procesamiento en tareas ejecutadas sobre diferentes computadoras conectadas a la red. En la implementación más sencilla, dos computadoras (un cliente y un servidor) se reparten la carga del procesamiento.

La característica distintiva es que las funciones de procesamiento están separadas entre cliente y servidor. Ambas computadoras son necesarias para completar el proceso de principio a fin.

1.6 Arquitecturas de aplicaciones cliente/servidor

Las aplicaciones cliente/servidor se construyen con base en diferentes modelos cuya característica es colocar el almacenamiento de los datos en el servidor y la representación de los datos en el cliente. La diferencia entre los modelos radica en cómo es dividida la lógica del negocio entre uno y otro.

1.6.1 Arquitecturas de dos capas

El modelo de dos capas hizo aparición con las aplicaciones desarrolladas para redes de área local en el final de los ochentas e inicios de los noventas y estuvo basada principalmente en técnicas sencillas para compartir archivos implementadas en productos estilo X-base.

1.6.1.1 Arquitectura de cliente principal (o grueso)

Localiza toda la lógica del negocio en el cliente (ver Figura 1.1). Esta arquitectura es comúnmente usada en los casos donde el usuario necesita ejecutar frecuentes análisis de datos que no se actualizan continuamente (o en *tiempo real*).

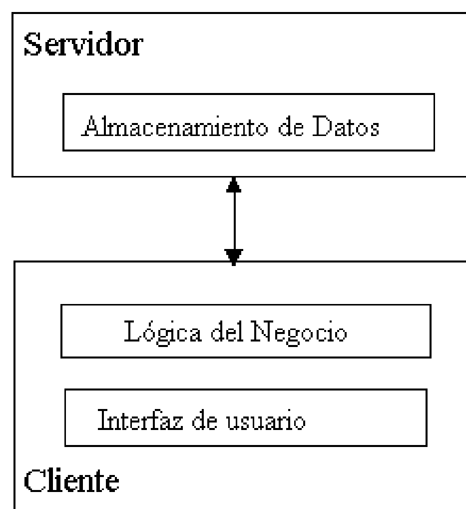

Figura 1.1: Arquitectura de cliente principal

Mientras más compleja es la aplicación, más grueso se hace el cliente y más poderoso debe ser el hardware para apoyarlo. El costo de la tecnología cliente podría incrementarse de manera prohibitiva e impedir el desarrollo de la aplicación. Adicionalmente, los clientes gruesos consumen mucho ancho de banda de la red, haciendo que el número efectivo de clientes que se pueden conectar se reduzca.

El modelo de cliente principal es utilizado en muchos sistemas en los cuales los datos son relativamente estáticos, pero el procesamiento es intenso, como en los sistemas de recursos humanos.

1.6.1.2 Arquitectura de servidor principal (o grueso)

Localiza toda la lógica del negocio en el servidor (ver Figura 1.2) implementándola como procedimientos almacenados, disparadores u otros mecanismos. Es efectivo en el uso del ancho de banda de la red; pues aunque es pesado, sí es más liviano que el enfoque de cliente principal.

La desventaja está en que los procedimientos almacenados enfatizan la dependencia en el motor de base de datos escogido. Adicionalmente, al colocar los procedimientos almacenados dentro de la base de datos, cada base de datos que contenga lógica del negocio debe ser modificada cuando ésta cambia. En bases de datos grandes y distribuidas, esto conducirá a dificultades en el manejo de actualizaciones.

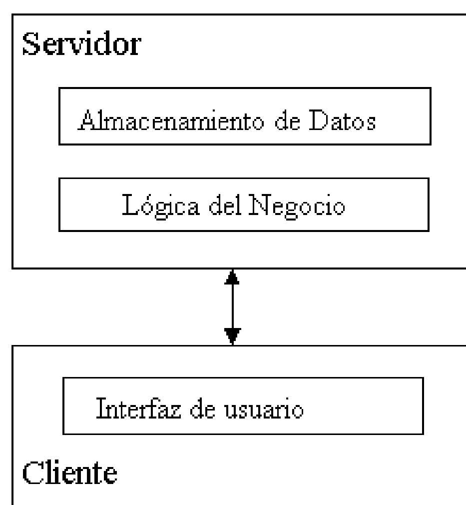

Figura 1.2: Arquitectura de servidor principal

El modelo de servidor principal es utilizado en muchos sistemas de administración de negocios, donde el acceso a los datos es un factor crítico.

1.6.1.3 Desventajas de las arquitecturas de dos capas

Desgraciadamente el modelo de dos capas muestra notables debilidades que hacen el desarrollo y mantenimiento de tales aplicaciones difícil. Las principales desventajas son:

1. La carga en la red se incrementa como resultado del intercambio de datos, especialmente en el modelo cliente principal.
2. Las computadoras personales son consideradas *no confiables* en términos de seguridad; es decir, son relativamente fáciles de violentar. A pesar del riesgo potencial, datos sensitivos se envían al cliente.
3. La lógica del negocio no puede ser reutilizada debido a que está ligada a un programa individual. Si se coloca en el lado cliente debe repetirse en cada uno de ellos, si está del lado del servidor debe replicarse en cada base de datos.
4. Cualquier cambio administrativo, en la política del negocio o en leyes aplicables obliga a actualizar la lógica del negocio replicada en muchos clientes.
5. El modelo de dos capas implica un complicado procedimiento de distribución de software. Cuando la lógica del negocio es ejecutada en el cliente, todos (quizá miles) tienen que actualizarse en el caso de una nueva versión. Además, el nuevo programa debe ser instalado, configurado y probado hasta que se ejecute correctamente. Esto puede ser muy caro, complicado, propenso a errores y necesita tiempo.
6. Los programadores de estas aplicaciones tienen que vérselas más de cerca con las restricciones y particularidades impuestas por el entorno operativo del cliente.

1.6.2 Arquitectura de tres capas

Una arquitectura de tres capas, como la de la Figura 1.3, divide las funciones del software así:

- La capa servidor

  Es responsable por el almacenamiento de los datos.

- La capa media

  Esta capa es nueva y no está presente explícitamente en la arquitectura de dos capas. Los componentes (u objetos) que implementan la lógica del negocio residen aquí. Entre sus tareas está la de acceder a la base de datos en nombre de los usuarios evitando de ésta manera que ellos la utilicen directamente.

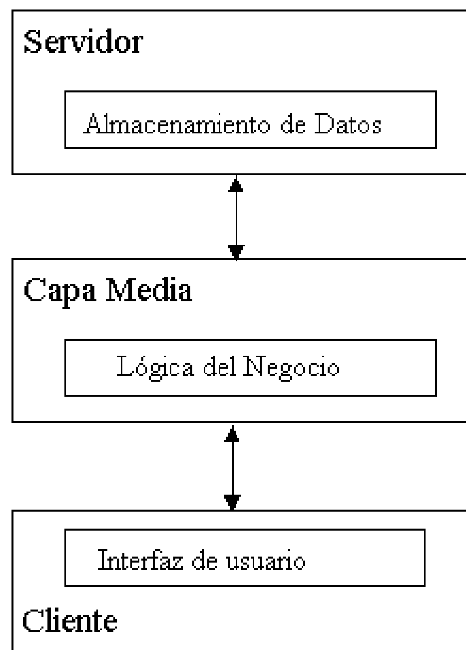

Figura 1.3: Arquitectura de tres capas

- La capa cliente

  Es responsable por la presentación de los datos, la recepción de eventos de usuario y el control de la interfaz de usuario. La lógica real del negocio se ha movido a la capa media.

Este modelo es más difícil de diseñar puesto que la división de la lógica del negocio debe maximizar las capacidades de cómputo en todas las capas para funcionar bien.

Esta arquitectura es típica en casos donde los datos son actualizados frecuentemente y donde el procesamiento es intenso. Se utiliza en muchos sistemas basados en

transacciones, tales como sistemas de administración financiera y sistemas de administración de inventarios.

Es importante notar que los límites entre las capas son lógicos, siendo posible ejecutar las tres capas en la misma máquina física. Lo importante es estructurar cuidadosamente el sistema y definir apropiadamente los límites de software entre las diferentes capas.

1.6.3 Arquitectura de múltiples capas

En la actualidad, la tendencia en la industria es usar una arquitectura de múltiples capas. La mayoría de los nuevos proyectos están siendo escritos como alguna clase de sistemas cliente/servidor de múltiples capas.

Una arquitectura multicapa no excluye el uso de los modelos de dos o de tres capas. Dependiendo de la escala de la aplicación y de los requerimientos de acceso a los datos, el modelo de dos o tres capas puede a menudo ser usado para aplicaciones departamentales.

No tiene sentido obligar a que las necesidades de un cliente pasen por el servidor de aplicaciones cuando no hay que cuidar la integridad de las transacciones. En ésta situación, el cliente debería estar en capacidad de acceder a los datos directamente en el servidor de bases de datos.

Generalmente, cualquier sistema cliente/servidor puede ser implementado con una arquitectura multicapa, donde la lógica de la aplicación es dividida entre varios servidores. Esta división de la aplicación crea una infraestructura integrada la que posibilita acceso global, consistente y seguro a los datos críticos.

Una de las principales metas de ésta arquitectura es proporcionar un entorno que haga posible acceso a escala mundial hacia los datos y aplicaciones corporativas desde cualquier sitio y con seguridad.

1.6.3.1 Factores clave

Una arquitectura multicapa se justifica cuando el escenario informático de la organización requiere un enfoque sofisticado y seguro. Varios factores se consideran antes de optar por ésta alternativa.

1. Visión expandida de los usuarios

El usuario de la aplicación deja de ser tan solo el empleado para convertirse en miembro de un grupo más amplio que incluye contratistas, clientes, asociados empresariales y proveedores, los usuarios ya no pueden ser identificados por la máquina que usan o la dirección IP.

Un usuario se convierte en una entidad independiente, distinguida por un identificador que es único entre empleados, contratistas, socios de negocios y cualquier otro tipo de usuario. Estos identificadores únicos son almacenados en una base de datos corporativa junto con la información necesaria para autentificar cada usuario y los derechos conferidos a él.

2. Ubicuidad

Dicho de manera sencilla, las aplicaciones deben estar disponibles para el usuario autorizado, esté donde esté.

La realidad empresarial de hoy exige que los usuarios se conectan a la aplicación desde cualquier sitio (la oficina, la casa o un avión) utilizando dispositivos diferentes (una computadora de escritorio, una computadora de red, una portátil o un dispositivo con capacidad de navegación Web). Para responder a éste requerimiento las aplicaciones se centran en el servidor.

Las aplicaciones basadas en servidor se descargan y ejecutan cuando y donde se necesiten. No dependen de archivos de configuración locales ni presumen la existencia de determinados recursos locales de red.

3. Seguridad

Dentro de un entorno empresarial seguro, las aplicaciones necesitan autenticar a los usuarios para permitir el acceso a los datos empresariales. Todas las transmisiones de la aplicación que ocurren sobre cualquier clase de medio público, como la Internet, deberían ser privadas y protegidas de mirones.

Asegurar que los servicios del servidor solo son proporcionados a los clientes que pasan, con sus solicitudes, información de autenticación, es una consideración clave de la seguridad. Tal información podría ser un nombre y clave de usuario en el caso más simple o una credencial de sesión cuando otro mecanismo de autenticación sea usado. Si el servidor no hace esto y confía en que el cliente autentifica al usuario, el sistema puede ser engañado. Por lo tanto es importante asegurar el servidor.

La comunicación entre las partes cliente y servidor de una aplicación también necesita ser segura, particularmente cuando la comunicación ocurre sobre un medio público. Una manera de proteger la comunicación es codificándola usando un algoritmo de encriptación.

1.6.3.2 Una arquitectura multicapa basada en servicios

La arquitectura de ejemplo mostrada en la Figura 1.4, está divida en cuatro capas y se basa en servicios reutilizables compartidos entre los clientes. La capa adicional se introdujo para mejorar el tiempo de descarga de los componentes y lograr acceso a los recursos de la red local.

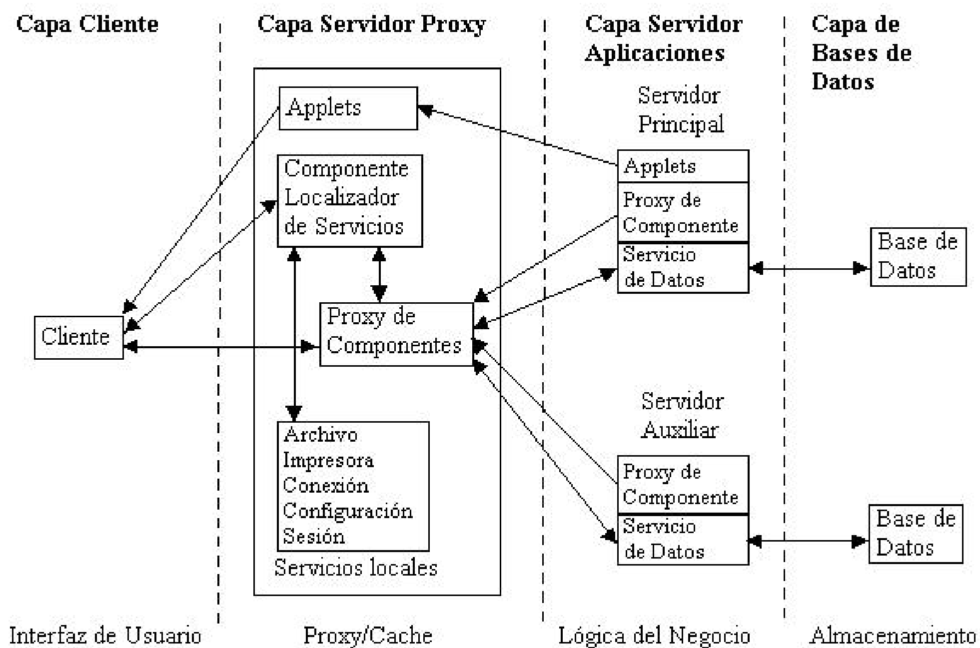

Figura 1.4: Ejemplo de una arquitectura de cuatro capas

1.6.3.2.1 Capa del servidor substituto

La segunda capa se llama servidor substituto y se coloca cerca del cliente. Un servidor substituto (o ***proxy***) proporciona una caché para pequeñas aplicaciones (*applets*) y datos estáticos, brinda acceso a recursos locales tales como archivos de usuario e impresoras. Además funciona como substituto para otros servicios.

El cliente utiliza un componente Localizador de Servicios en el servidor substituto para encontrar el proveedor de servicio apropiado de datos empresariales o recursos de red que necesita. Por ejemplo, si una aplicación cliente necesita imprimir algo, solicita y recibe de vuelta del componente Localizador una "referencia" hacia un servicio de impresión. El cliente luego usa ese servicio de impresión para imprimir sus datos. Si el cliente necesita recuperar o actualizar datos en una base de datos empresarial, el cliente solicita y recibe de vuelta del componente Localizador una "referencia" a la aplicación empresarial de servicio de datos apropiada para los datos en cuestión. El cliente entonces utiliza ese servicio para manipular los datos.

La capa substituto esencialmente empareja applets cliente con uno o más servicios reutilizables según necesite.

1.6.3.2.2 Capa del servidor de aplicaciones

El servidor de aplicaciones proporciona acceso a los datos empresariales y también implementa la lógica del negocio y la validación de los datos. El servidor de aplicaciones es responsable por todo el manejo de las transacciones de base de datos.

El servidor de aplicaciones proporciona un servicio implementando una API. Los clientes llaman a ésta API para recuperar o manipular los datos empresariales gobernados por los servicios. Por ejemplo, un servicio de autenticación valida la conexión del usuario leyendo la base de datos corporativa de autenticación. Un servicio de reporte de gastos proporciona información de la base de datos conteniendo información relacionada a gastos.

Para cada servicio, hay un substituto el cual es dinámicamente descargado desde el servidor de aplicaciones hacia el servidor substituto. Cuando un cliente solicita un servicio, el componente Localizador de Servicios del substituto retorna al cliente una referencia al substituto de componente de aplicación apropiado. El cliente luego envía su petición de

servicio a este substituto de componente dentro del servidor substituto. El substituto de componente despacha las peticiones de servicio del cliente hacia el servidor de aplicaciones.

Este modelo es bastante bueno en términos de seguridad. También es fácil de distribuir dado que todo el código de aplicación para el cliente (los applets) y el servidor substituto (substituto de componentes) es dinámicamente descargado desde el servidor de aplicaciones.

Cualquier servicio puede implementarse usando otros servicios, así que el número de capas para una aplicación real pudiese ser mayor de cuatro.

1.6.4 Ventajas de una arquitectura multicapa

La arquitectura de múltiples capas resuelve cierto número de problemas inherentes a las arquitecturas de dos capas. Naturalmente también causa nuevos problemas, pero estos son compensados por sus ventajas.

1. Separación de componentes

   A través de ésta separación, más clientes están en capacidad de acceder a una amplia variedad de aplicaciones de servidor. Las dos principales ventajas para las aplicaciones cliente son claras: (1) desarrollo más rápido por medio de la reutilización de componentes de lógica del negocio preconstruidos y (2) una fase de pruebas más corta, dado que los componentes de servidor ya han sido probados.

2. Independencia del almacenamiento

   Un cambio en la estrategia de almacenamiento no altera los clientes. En sistemas bien diseñados, el cliente continua usando los datos por medio de una interfaz estable y bien diseñada que encapsula los detalles de almacenamiento.

3. Escalabilidad

   Una arquitectura multicapa incrementa la escalabilidad y desempeño de la aplicación haciendo posible un gran número de conexiones clientes concurrentes.

En un modelo de dos capas, cada cliente mantiene una conexión a la base de datos. En un modelo multicapa, los clientes solamente se conectan al servidor de aplicaciones. El servidor de aplicaciones puede multiplexar solicitudes de clientes por medio de un fondo común de conexiones existentes. Esto reduce la carga en el servidor de bases de datos, mientras la carga en la capa del servidor de aplicaciones puede ser balanceada a través de múltiples servidores.

Esta redistribución incrementa la capacidad de procesamiento de la aplicación y mejora el desempeño global.

4. Variedad de clientes

El enfoque de servidor principal permite que múltiples clientes sean escritos si fuese necesario. Un cliente de capacidad completa para el escritorio puede ser simplificado y proporcionado en una versión más ligera para dispositivos como teléfonos celulares o asistentes personales (PDA).

El modelo multicapa conduce a clientes más delgados ya que la mayoría de la lógica se ejecuta en las capas intermedias. Los clientes por lo tanto requieren menos memoria. Esto asegura que un amplio rango de plataformas clientes puedan ejecutar la aplicación.

5. Eficiencia en la red

Este modelo de múltiples capas incrementa la eficiencia reduciendo el uso de la red en varias formas.

Los datos de aplicación, que son relativamente estáticos, pueden ser colocados en una caché dentro de los servidores de aplicaciones e incluso descargados a caches en la capa del substituto para consultas más rápidas. En este caso, los datos solamente viajan por una red de área amplia una vez en lugar de cada vez por cliente.

Adicionalmente, servicios como impresión y acceso a archivos, que implican recursos de red cercanos al cliente, pueden ser implementados en la capa de substituto, la capa más cercana a ellos. El uso de la red también se reduce porque el cliente ligero es descargado una sola vez para cada grupo de usuarios que utiliza la aplicación.

6. Seguridad más simple

La implementación de aplicaciones seguras se vuelve más fácil con una arquitectura multicapa basada en servicios.

En primer lugar, la autenticación es un servicio auxiliar llamado por cada aplicación, en lugar de implementarse repetidamente en cada aplicación. Usando la tecnología de muros de fuego, el acceso a la base de datos puede ser restringido a solo el servidor de aplicaciones. Esto asegura que la base de datos solo puede ser cambiada en formas permitidas por el servidor de aplicaciones. Finalmente, la capa del substituto puede ser configurada y colocada sobre un muro de fuego, proporcionando un modelo que trabaja tanto para la intranet como para una extranet.

Un importante beneficio de seguridad es la posibilidad de otorgar y revocar privilegios rápidamente para muchos tipos de usuario puesto que la información de autenticación está en un solo lugar y no en cada aplicación de base de datos. Si los usuarios dejan la compañía o cambia su perfil, los privilegios de acceso para todas las aplicaciones son actualizados en una operación. Si la información de conexión está almacenada en cada base de datos de aplicación individual, es improbable que cuando los usuarios se vayan, sus permisos sean bloqueados o removidos para cada aplicación a la que tenían acceso.

7. Administración

Por supuesto que es más fácil y más rápido cambiar un componente en el servidor que equipar numerosos clientes con la nueva versión del programa. Sin embrago, es obligatorio que las interfaces permanezcan estables y que las versiones para clientes antiguos sigan trabajando. Adicionalmente, tales componentes requieren de un alto estándar de control de calidad. Esto es debido a que componentes de mala calidad pueden poner en peligro las funciones de todo un conjunto de aplicaciones cliente.

Clientes de "*Administración Cero*"

Este modelo facilita crear aplicaciones sin ningún conocimiento del cliente. El único requerimiento en el cliente es un navegador que soporte HTTP (o mejor su versión

segura HTTP-S) y SSL. Desde una perspectiva administrativa, la distribución de aplicaciones y el problema de administración es enormemente reducido puesto que solo se administran los servidores. No es necesaria la instalación o configuración de aplicaciones en el cliente.

Servidores de "*Administración Cero*"

Finalmente, puesto que no hay configuración específica de aplicación o administración a ser hecha, los servidores de la capa del substituto son servidores con "administración cero". Las tareas de administración en éstas máquinas se hacen todas al momento de la instalación e incluyen la instalación del sistema operativo, el servidor Web, configuraciones de impresión y otros servicios esenciales. Todo el código de aplicaciones empresariales se descarga bajo petición. Ningún código específico de aplicación es persistente en los servidores de la capa del substituto.

8. Servicios reutilizables

Una ventaja de ésta arquitectura es que al mirar las aplicaciones como colecciones de servicios, aquellos que conforman una aplicación pueden ponerse a disposición de otras aplicaciones que los requieren para sus propios propósitos. Por ejemplo, las aplicaciones pueden compartir un mecanismo de autenticación en lugar de desarrollar y mantener el suyo propio.

Esto reduce el trabajo administrativo y de desarrollo requerido para cada aplicación.

1.7 Componentes de servidor

Los componentes de servidor son componentes de aplicaciones (o *servicios*) que se ejecutan en la capa media, concretamente, en un servidor de aplicaciones. A cada uno de estos componentes se le asigna una parte de la lógica de negocio.

Un servidor de aplicaciones proporciona la infraestructura necesaria para mantener operativos a los componentes que se colocan dentro de él. El modelo permite tomar ventaja del poder de multiproceso y multiprocesamiento de los sistemas actuales.

1.7.1 Servidores de aplicaciones

Un servidor de aplicaciones administra y recicla recursos escasos del sistema tales como procesos, hilos de ejecución, memoria, conexiones de base de datos y sesiones de red en nombre de las aplicaciones. Algunos de los servidores más sofisticados ofrecen servicios de nivelación de carga que pueden distribuir el procesamiento de la aplicación entre muchos sistemas.

Un servidor de aplicaciones también proporciona acceso a servicios de infraestructura, tales como sistemas de nombres, directorios, transacciones, persistencia y seguridad.

El servidor proporciona una interfaz de programación de aplicaciones (**API**) que los clientes usan para llegar a los componentes[1].

Existen diferentes tipos de servidores de aplicaciones de uso común y cada uno proporciona un contenedor para algún tipo de petición basada en servidor. Por ejemplo:

- Un vigilante de transacciones contiene transacciones y administra recursos compartidos en nombre de una transacción. Múltiples transacciones pueden trabajar juntas y confiar en él para coordinar la transacción extendida.
- Un sistema de base de datos contiene peticiones de base de datos. Múltiples clientes de base de datos pueden enviar peticiones a la base de datos concurrentemente y confiar en el sistema para coordinar bloqueos y transacciones.
- Un servidor Web atiende peticiones de páginas Web. Múltiples clientes Web pueden enviar peticiones concurrentes de páginas hacia el servidor Web. El servidor Web proporciona páginas HTML o invoca extensiones de servidor (**servlets**) en respuesta a las peticiones.

1.7.2 Componentes y contenedores

Un **componente** es un bloque de software reutilizable; una pieza preconstruida de código de aplicación encapsulado que puede ser combinado con otros componentes y con nuevo código para producir rápidamente una aplicación personalizada.

[1] Hasta hace poco, cada servidor de aplicaciones utilizaba un conjunto propio de interfaces, ahora con la aparición de Java Empresarial los proveedores pueden acogerse a un marco común.

Los componentes se ejecutan dentro de una construcción llamada un **contenedor**. Un contenedor proporciona un contexto de aplicación para uno o más componentes y brinda servicios de administración y control para ellos. En términos prácticos, un contenedor proporciona un proceso o hilo del sistema operativo en el cual ejecutar el componente.

Los componentes de servidor no son visuales y se ejecutan dentro de un contenedor que es proporcionado por el servidor de aplicaciones. El contenedor maneja todos los recursos en nombre del componente y administra todas las interacciones entre los componentes y el sistema externo.

Un **modelo de componentes** define la arquitectura básica de un componente, especificando la estructura de sus interfaces y los mecanismos por los cuales actúa con su contenedor y con otros componentes. El modelo de componentes proporciona guías para crear e implementar componentes que puedan trabajar juntos para formar una aplicación mayor.

Un desarrollador de aplicaciones debería estar en capacidad de hacer uso completo del componente sin requerir acceso a su código fuente. Los componentes pueden ser personalizados para satisfacer los requerimientos específicos de una aplicación a través de un conjunto de propiedades externas. Por ejemplo, un componente que implementa una función de administración de cuentas podría permitir a un usuario añadir un proceso especial de aprobación para préstamos sobre una cierta cantidad de dólares. Una propiedad sería usada para indicar que esas funciones especiales de aprobación están habilitadas, una segunda propiedad identificaría las condiciones que requieren aprobación especial y una tercera propiedad indicaría el nombre del componente de proceso de aprobación que debería ser llamado cuando las condiciones existan.

Con el propósito de alcanzar los máximos beneficios de la arquitectura multicapa, los componentes de servidor deberían ser implementados como **servidores compartidos**. Los servidores compartidos de alto crecimiento necesitan dar soporte a usuarios concurrentes y necesitan compartir eficientemente recursos escasos del sistema como hilos, procesos, memoria, conexiones de base de datos y conexiones de red. Para operaciones de negocios, los servidores compartidos deben participar en transacciones. En la mayoría de los casos, un servidor compartido necesita reforzar políticas de seguridad.

Capítulo 2

Java Empresarial

2.1 La tecnología Java

2.1.1 Origen

Java se desarrolló en Sun Microsystems para resolver simultáneamente todos los problemas que se le plantean a los desarrolladores de software por la proliferación de arquitecturas incompatibles, tanto entre las diferentes máquinas como entre los diversos sistemas operativos y sistemas de ventanas que funcionaban sobre una misma máquina, añadiendo la dificultad de crear aplicaciones distribuidas en una red como Internet.

Hace algunos años, Sun Microsystems decidió intentar introducirse en el mercado de la electrónica de consumo y desarrollar programas para pequeños dispositivos electrónicos. Sun decidió crear una filial, denominada FirstPerson Inc., para dar margen de maniobra al equipo responsable del proyecto.

El mercado inicialmente previsto para los programas de FirstPerson eran los equipos domésticos: microondas, tostadoras y, fundamentalmente, televisión interactiva. Este mercado, dada la falta de pericia de los usuarios para el manejo de estos dispositivos, requería unas interfaces mucho más cómodas e intuitivas que los sistemas de ventanas que proliferaban en el momento.

Otros requisitos importantes a tener en cuenta eran la fiabilidad del código y la facilidad de desarrollo. James Gosling, el miembro del equipo con más experiencia en lenguajes de programación, decidió que las ventajas aportadas por C++ no compensaban el gran costo de pruebas y depuración. Gosling había estado trabajando en su tiempo libre en un lenguaje de programación que él había llamado **Oak**, el cual, aún partiendo de la sintaxis de C++, intentaba remediar las deficiencias que iba observando.

El primer proyecto en que se aplicó este lenguaje recibió el nombre de Proyecto Green y consistía en un sistema de control completo de los aparatos electrónicos y el entorno de un

hogar. Posteriormente se aplicó a otro proyecto denominado VOD (Video On Demand) en el que se empleaba como interfaz para la televisión interactiva. Ninguno de estos proyectos se convirtió nunca en un sistema comercial, pero fueron desarrollados enteramente en un Java primitivo.

FirstPerson cerró en la primavera de 1994 una vez que en Sun se dieron cuenta de que a corto plazo la televisión interactiva no iba a ser un gran éxito.

Pese a lo que parecía ya un olvido definitivo, Bill Joy, cofundador de Sun y uno de los desarrolladores principales del Unix de Berkeley, juzgó que Internet podría llegar a ser el campo adecuado para disputar a Microsoft su primacía casi absoluta en el terreno del software y vio en Oak el instrumento idóneo para llevar a cabo estos planes. Tras un cambio de nombre y modificaciones de diseño, el lenguaje Java fue presentado en agosto de 1995.

2.1.2 Elementos

La tecnología Java es tanto un lenguaje de programación como una plataforma.

2.1.2.1 El lenguaje de programación Java

El lenguaje de programación Java es un lenguaje de alto nivel cuyo propósito es mejorar la productividad y por este motivo está diseñado para ayudar más e interferir menos, para ser práctico. Las decisiones de diseño del lenguaje están basadas en proporcionar el máximo beneficio al programador.

Las características principales que ofrece Java respecto de cualquier otro lenguaje de programación son:

- Es simple

  Java ofrece toda la funcionalidad de un lenguaje potente, sin las características menos usadas y más confusas de otros. C y C++ son los lenguajes más difundidos, por ello Java se diseñó para ser parecido a C++ y así facilitar un rápido y fácil aprendizaje.

  Java reduce en un 50% los errores más comunes de programación con lenguajes como C y C++ al eliminar muchas de las características de éstos.

- Es orientado a objetos

  Java proporciona una implementación completa del paradigma orientado a objetos al satisfacer sus tres requisitos: encapsulamiento, herencia y polimorfismo. Aunque no es igual, se puede implementar herencia múltiple por medio de las interfaces.

- Posee un esquema de manejo de errores

  El manejo de errores en C es un problema notable y a menudo es ignorado. Cuando se construye un sistema grande y complejo, no hay nada peor que tener un error oculto en algún lugar sin pista de dónde proviene. El manejo de excepciones de Java es una manera de garantizar que un error es identificado y que algo ocurre como resultado.

- Es de arquitectura neutral

  Con la mayoría de lenguajes de programación, bien se puede compilar o interpretar un programa para ejecutarlo en una computadora. El lenguaje de programación Java es inusual porque es tanto compilado como interpretado. Con el compilador, se traduce el código fuente en un lenguaje intermedio llamado *bytecodes Java*. El interprete analiza y ejecuta en la computadora cada instrucción bytecode Java. La compilación se realiza una sola vez; la interpretación ocurre cada vez que el programa es ejecutado. La siguiente figura ilustra como trabaja esto.

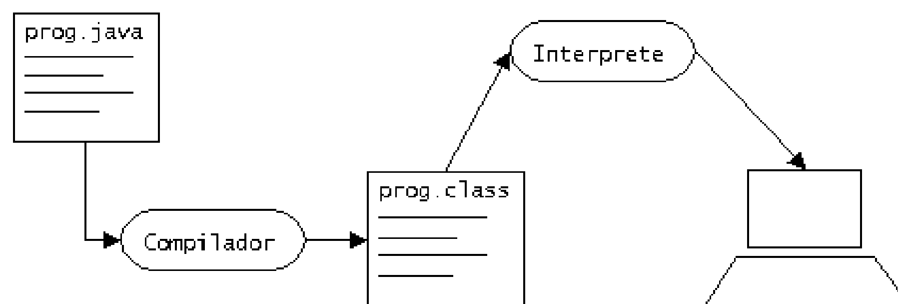

Figura 2.1: Flujo del código Java hasta su ejecución

  Se puede compilar un programa a bytecodes en cualquier plataforma que tenga un compilador Java. Los bytecodes pueden ser ejecutados sobre cualquier implementación de la Máquina Virtual Java. Esto significa que mientras una computadora tenga una Máquina Virtual Java, el mismo programa se puede ejecutar donde sea (ver Figura 2.2).

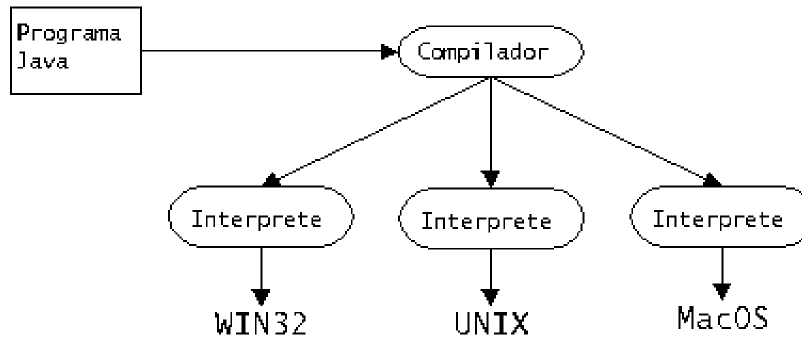

Figura 2.2: Independencia de la plataforma de un programa Java

- Es seguro

La seguridad en Java tiene dos facetas. En el lenguaje, características como los punteros o la conversión implícita se eliminan para prevenir el acceso ilegal a la memoria.

El código Java pasa muchas pruebas antes de ejecutarse en una máquina. El código se pasa a través de un verificador de bytecodes que comprueba el formato de los fragmentos de código y aplica un probador de teoremas para detectar fragmentos de código ilegal.

El *Cargador de Clases* también ayuda a Java a mantener su seguridad, separando el espacio de nombres del sistema de archivos local, del de los recursos procedentes de la red. Esto limita cualquier aplicación del tipo *Caballo de Troya* ya que las clases se buscan primero entre las locales y luego entre las procedentes del exterior. Lo que imposibilita a una clase suplantar a una predefinida.

Las aplicaciones Java resultan extremadamente seguras ya que no acceden a zonas delicadas de memoria o del sistema, con lo cual evitan la interacción de ciertos virus. Java no posee una semántica específica para modificar la pila de programa, la memoria libre o utilizar objetos y métodos de un programa sin los privilegios del kernel del sistema operativo. Además, para evitar modificaciones por parte de los crackers de la red, implementa un método ultraseguro de autenticación por clave pública. El Cargador de Clases puede verificar una firma digital antes de crear una instancia de un objeto. Por tanto, ningún objeto se crea y almacena en memoria, sin que se validen los privilegios de acceso.

- Es concurrente

  Al ser de múltiples hilos, Java permite muchas actividades simultáneas en un programa. Al estar los hilos construidos en el lenguaje, son más fáciles de usar y más robustos que sus homólogos en C o C++.

  El beneficio de ser concurrente consiste en mejor rendimiento y mejor comportamiento en tiempo real. Aunque el comportamiento en tiempo real está limitado a las capacidades del sistema operativo subyacente, aún supera a los entornos de flujo único de programa (single-threaded) tanto en facilidad de desarrollo como en rendimiento.

2.1.2.2 La plataforma Java

Una *plataforma* es el entorno hardware o software dentro del cual un programa se ejecuta. La mayoría de las plataformas pueden ser descritas como una combinación del sistema operativo y el hardware. La plataforma Java difiere de la mayoría en que es una plataforma solamente de software que se ejecuta sobre otras plataformas basadas en hardware.

La plataforma Java tiene dos componentes:

- La Máquina Virtual Java (JVM)
- La Interfaz de Programación de Aplicaciones Java (API Java)

Ya se ha mencionado a la JVM. Es la base de la plataforma Java y se ha llevado a varias plataformas basadas en hardware.

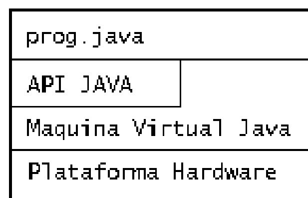

Figura 2.3: Colocación de la plataforma Java dentro del entorno de ejecución

La Figura 2.3 presenta un programa que se ejecuta sobre la plataforma Java y muestra cómo la API Java y la JVM aíslan al programa del hardware. La API Java es una gran colección de componentes de software ya listo que proporciona muchas capacidades útiles,

como por ejemplo utilidades de la interfaz de usuario. Ella está agrupada en bibliotecas de clases e interfaces relacionadas. Estas bibliotecas son conocidas como *paquetes*.

2.1.3 ¿Qué puede hacer la tecnología Java?

El lenguaje de programación Java de alto nivel y propósito general es una potente plataforma software. Utilizando la API se pueden escribir muchos tipos de programas. Se pueden escribir applets para ejecutarse dentro de navegadores y aplicaciones que funcionan de forma independiente. Una clase especial de aplicación conocida como *servidor* sirve y apoya a los clientes de una red. (Ejemplos de servidores son servidores Web, servidores substitutos y servidores de correo.) Otro tipo especial de programa es un *servlet*. Los Servlets Java son la elección más popular para crear aplicaciones Web interactivas.

¿Cómo puede la API Java soportar todas estas clases de programas? Lo hace con paquetes de componentes de software que proporcionan un amplio rango de funciones. La Figura 2.4 muestra lo que una implementación completa de la plataforma Java proporciona.

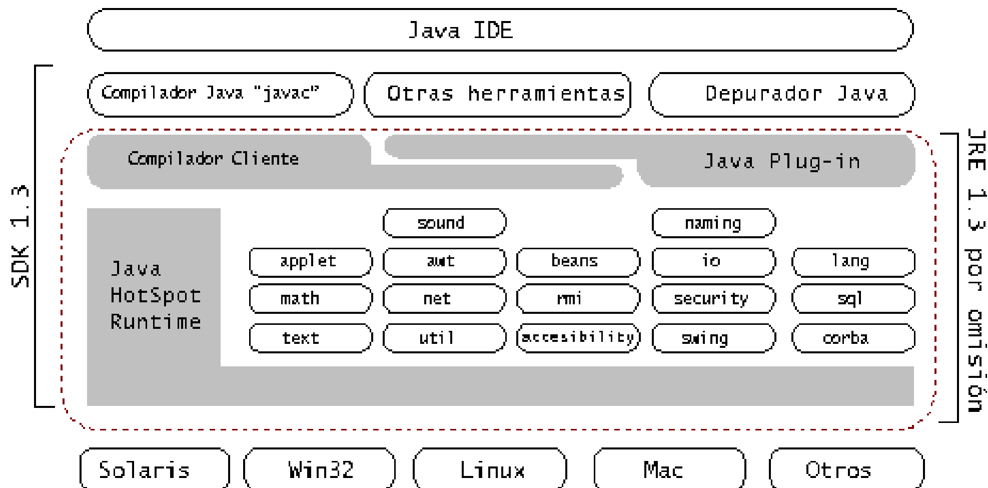

Figura 2.4: Componentes del SDK Java 2, versión 1.3

El Entorno de Ejecución Java 2 (JRE) consiste de la máquina virtual, las clases del núcleo de la plataforma Java y archivos de soporte. El Paquete de Desarrollo de Software (SDK) Java 2 incluye el JRE y herramientas de desarrollo como compiladores y depuradores.

2.2 Java para aplicaciones empresariales

La plataforma Java es inherentemente adecuada para la computación multicapa. En el cliente, Java proporciona mejoras a la lógica de presentación a través de los applets Java y los JavaBeans. Su independencia de plataforma junto con la seguridad y la capacidad de concurrencia establecen a Java como la plataforma a elegir para los clientes ligeros. Es evidente el apoyo de Java en todos los mejores navegadores Web del mercado.

Del lado del servidor, la plataforma Java está formada por un conjunto de APIs formalmente conocidas como **Java 2 Enterprise Edition** (**J2EE**) que proporcionan servicios de clase empresarial para aplicaciones multicapa escritas en Java[2]. La Figura 2.5 demuestra la interacción de los elementos de la plataforma empresarial Java.

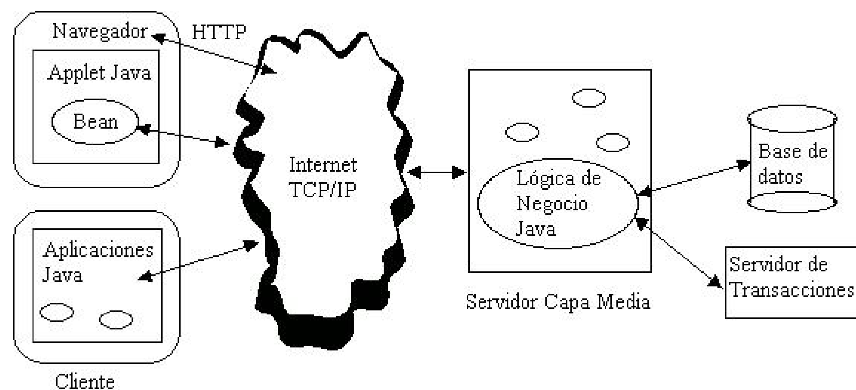

Figura 2.5: Elementos de una arquitectura empresarial Java

La plataforma Java Empresarial ofrece al desarrollador muchas de las mismas ventajas en el servidor que Java ofrece en el cliente, como portabilidad y seguridad. Adicionalmente, Java proporciona el medio necesario para integrar y extender la variedad de sistemas de toda la empresa. Usando la plataforma Java Empresarial, una organización puede rápidamente desarrollar aplicaciones de plataforma independiente e integrar a los ya existentes monitores de procesamiento de transacciones, sistemas de acceso de datos, servicios CORBA y otros sistemas de infraestructura. En otras palabras, una organización puede usar Java Empresarial para juntar e integrar hardware y software existente en nuevas maneras.

[2] De aquí en adelante, el término "Java Empresarial" es usado para referirse a éste conjunto de APIs del lado servidor.

La plataforma Java posibilita auténtica computación multicapa al permitir a programas ejecutándose en un servidor ser movidos a un servidor diferente sin volver a escribir el código. Más importante, estos programas Java pueden interoperar con los servicios de infraestructura subyacentes en diferentes plataformas sin reescribir código.

2.2.1 Factores clave

Java actualmente está en boca de todos. Pero, surge la pregunta de si ésta es una buena tecnología para desarrollar aplicaciones de nivel empresarial. ¿Es Java una buena tecnología allí donde la red es el punto crítico? Vamos a intentar responder comparando las capacidades de Java contra la lista de necesidades de la red corporativa.

1. Aplicaciones efectivas y eficientes

Las aplicaciones que se crean en grandes empresas deben ser más efectivas que eficientes; es decir, conseguir que el programa funcione y el trabajo salga adelante es más importante que el que lo haga eficientemente. Esto no es una crítica, es una realidad.

Al ser un lenguaje más simple que cualquiera de los que ahora están disponibles, Java permite concentrarse en la mecánica de la aplicación, en vez de pasarse horas y horas incorporando APIs para el control de las ventanas, controlando minuciosamente la memoria, sincronizando los archivos de cabecera y corrigiendo los agónicos mensajes del enlazador.

Java tiene su propio conjunto de interfaces, maneja por sí mismo la memoria que utiliza la aplicación y solamente usa enlace dinámico. Muchas de las implementaciones de Java ofrecen compiladores nativos Just-In-Time (JIT). Si la JVM dispone de un compilador instalado, las clases del byte-code de la aplicación se compilarán hacia la arquitectura nativa de la computadora del usuario. Los programas Java en ese momento rivalizarán con el rendimiento de programas en C++. La compilación JIT tiene lugar en el sistema del usuario como una parte (opcional) del entorno local de Java.

2. Programador y programa portátiles

En una empresa de relativo tamaño hay una multitud de computadoras posiblemente muy diferentes entre sí. Desarrollar aplicaciones corporativas para un grupo así es excesivamente complejo y caro. Hasta ahora era complicado unificar la arquitectura utilizando una API común para reducir el costo de las aplicaciones.

Con una JVM portada a cada una de las plataformas presentes en la empresa y una buena biblioteca de clases, los programadores pueden entenderse. Esta posibilidad fomenta el uso de Java, justo donde otros intentos anteriores han fracasado.

Una vez que los programas estén escritos en Java otro lado interesante del asunto es que los programadores también son “portátiles”. El grupo de programadores de la empresa puede ahora enfrentarse a un desarrollo para cualquiera de las plataformas. La parte del cliente y del servidor de una aplicación estarán ahora escritas en el mismo lenguaje.

3. Menores costes de desarrollo

En contraste con el alto costo de los desarrollos realizados sobre estaciones de trabajo, el coste de creación de una aplicación Java es similar a desarrollar sobre una computadora personal.

Desarrollar utilizando un software caro para una estación de trabajo es un problema en muchas empresas. La eficiencia del hardware y el poco costo de mantenimiento de una estación de trabajo Sun, por ejemplo, resulta muy atractivo para las empresas; pero el coste adicional del entorno de desarrollo con C++ es prohibitivo para la gran mayoría de ellas. La llegada de Java e intranet reducen considerablemente estos costes.

Las herramientas Java no están en el rango de precios de decenas de miles de dólares, sino a los niveles confortables. Y con el crecimiento cada día mayor de la comunidad de desarrolladores de software **GNU**, los programadores corporativos tienen un amplio campo donde moverse y muchas oportunidades de aprender y muchos recursos a su disposición.

4. Mejor mantenimiento y soporte

Un problema bien conocido que ocurre con el software corporativo es la demanda de cuidados y actualización. Java no es, ciertamente, la cura definitiva, pero tiene varias características que hacen la vida más fácil.

Uno de los componentes del SDK es ***javadoc***, que puede fácilmente generar páginas HTML con el contenido de comentarios formateados y que pueden visualizarse en cualquier navegador. Esto hace que el trabajo de documentar el código de nuevas clases Java sea trivial.

Java reduce drásticamente las dependencias en proyectos complejos. No hay archivos de cabecera. Java necesita que todo el código fuente de una clase se encuentre en un solo archivo.

5. Fácil aprendizaje

Si la empresa está llena de programadores de C++ con alguna experiencia en el manejo de bibliotecas gráficas, aprenderán rápidamente lo esencial de Java. Si el equipo de ingenieros no conoce C++, pero maneja cualquier otro lenguaje de programación orientada a objetos, les llevará pocas semanas dominar la base de Java.

Si los ingenieros de la empresa no conocen ningún lenguaje orientado a objetos, sí que tienen que aprender los fundamentos de ésta tecnología antes de nada y luego aplicarlos a la programación con Java. Tanto el análisis como el diseño orientado a objetos deben ser comprendidos antes de intentar nada con Java. Los programadores de Java sin un fondo de conocimientos de AOO/DOO producirán código pobre.

2.3 Arquitectura de la plataforma

Como se muestra en la Figura 2.6, la arquitectura de Java Empresarial es bastante simple. Está basada en la metáfora de “capas” y componentes reutilizables. La premisa básica es que puede ser construida sobre servicios de infraestructura y sistemas antiguos existentes tales como servicios de procesamiento de transacciones, nombre y directorios, acceso a bases de datos y CORBA.

Además de un conjunto de interfaces para los diferentes servicios de infraestructura, la plataforma Java Empresarial proporciona un modelo de componentes para encapsulamiento y reutilización. El modelo incluye JavaBeans, el modelo de componentes Java para el cliente y los componentes empresariales Java (**Enterprise JavaBeans, EJB**).

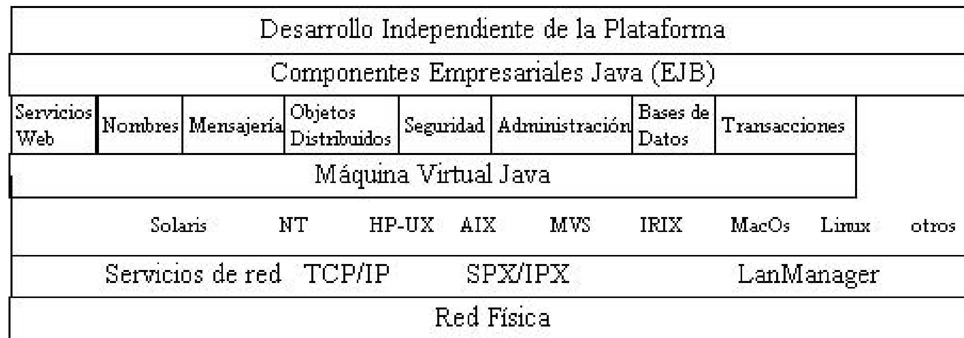

Figura 2.6 Arquitectura de J2EE

Este modelo hace posible producir rápidamente una aplicación personalizada basada en componentes más pequeños preconstruidos.

Para comprender el alcance completo de ésta arquitectura, es necesario hacer un acercamiento a la infraestructura empresarial para la cual fue diseñada.

2.3.1 Servicios

Las redes corporativas modernas (intranets) están construidas alrededor de una infraestructura empresarial la cual, como conjunto, proporciona un amplio rango de servicios a los administradores de sistemas y los desarrolladores de aplicaciones críticas. Los servicios más importantes que la infraestructura debe proveer son:

1. Servicios de nombres y directorios

Los servicios de nombres y directorios cumplen un papel vital en la empresa permitiendo compartir en toda la red una variedad sobre usuarios, máquinas, redes, servicios y aplicaciones. Usando Java Empresarial, las aplicaciones pueden usar transparentemente información de diferentes servicios como LDAP, NDS, DNS, NIS y NIS+ a través de una interfaz común. Esto permite a las aplicaciones Java coexistir con aplicaciones y sistemas antiguos.

2. Servicios de acceso a bases de datos

La mayoría de las aplicaciones empresariales requieren algún tipo de acceso y conexión a base de datos. Las aplicaciones deben ser capaces de leer los datos en el amplio rango de bases de datos existentes. La plataforma proporciona una interfaz uniforme para enviar instrucciones SQL y procesar los resultados para cualquier base de datos que disponga de un controlador.

3. Servicios de objetos distribuidos

En un ambiente distribuido, la lógica de negocio y los servicios de red en la forma de componentes podrían residir en uno o más servidores dispersos en la red. Uno de los enfoques más corrientes para interconectar estos componentes diferentes es por medio de CORBA. Java Empresarial proporciona una interfaz uniforme para usar directamente e integrarse con servicios de objetos remotos CORBA. De ahí que, una gran parte de la infraestructura "antigua" existente pueda ser reutilizada sin tener que volver a codificar nada en Java.

4. Servicios de administración empresarial

La administración empresarial es un requisito clave en cualquier ambiente de computación distribuida. Proporciona operaciones remotas de administración para salvaguardar contra acceso no autorizado y para administrar recursos de red. La plataforma proporciona una interfaz independiente del protocolo que trabaja con diferentes protocolos, incluyendo SNMP. Esta interfaz puede ser usada en una combinación particular de ambientes involucrando numerosos sistemas operativos, arquitecturas y protocolos de red.

5. Servicios de procesamiento de transacciones

Java Empresarial proporciona un conjunto de interfaces estándares que trabajan con monitores de transacciones existentes como Object Transaction Service (OTS) de OMG. Está basada en el modelo de procesamiento distribuido de transacciones (DTP) de X/Open y proporciona un mecanismo para conectar recursos de múltiples sistemas y bases de datos en una sola transacción.

6. Servicios de mensajería

Las organizaciones generalmente tienen una o más aplicaciones antiguas que requieren comunicarse y operar entre ellas. Los mensajes y las colas de mensajes proporcionan éste mecanismo, el cual incluye solicitud / respuesta, publicación / suscripción y envío. La plataforma Java Empresarial proporciona servicios para manejar colas de mensajes y hace estas colas confiables, garantizadas y en algunos casos basadas en transacciones.

El mecanismo es independiente del vendedor y provee interfaces para hacer el código de aplicación portátil entre diferentes servidores de mensajería.

7. Servicios de seguridad

La seguridad del servidor requiere soluciones en cuatro áreas generales: autenticación, autorización, integridad y privacidad.

Autenticación es el medio por el cual el sistema reconoce al usuario remoto que ingresa una transacción o lee datos. Autorización es el medio por el cual el sistema determina que datos podría leer un usuario particular y bajo qué condiciones él podría continuar leyéndolos. Integridad es el medio que usa el sistema para garantizar que los datos no son corrompidos o modificados maliciosamente mientras están en tránsito sobre la red. Y finalmente, privacidad es el medio por el cual el sistema previene la revelación de datos sensitivos durante la transmisión y almacenaje.

La plataforma proporciona un conjunto de APIs de seguridad las cuales utilizan y extienden servicios de seguridad en cada una de estas cuatro áreas. Juntas, estas interfaces aseguran aplicaciones Java seguras sobre LANs y WANs.

8. Servicios Web

Para lograr el potencial completo de la computación de cliente ligero, la mayoría de las organizaciones confían en el servidor HTTP para entregar páginas Web estáticas o para aceptar entrada basada en formas y generar dinámicamente páginas HTML. Java Empresarial proporciona un conjunto de extensiones Web independientes de plataforma y clases que pueden ser usadas para construir servidores de red como servidores Web, servidores substitutos, servidores de correo, servidores de impresión,

etc. Estas trabajan independientemente del hardware, sistema operativo y tipo de servidor Web. Además es el reemplazo perfecto para los guiones CGI.

2.4 Elementos de Java Empresarial

La plataforma Java Empresarial proporciona un modelo de computación y una arquitectura de plataforma abierta que apoya un entorno compacto para desarrollar, implementar y administrar aplicaciones empresariales multicapa. Este modelo está enfocado en tres temas importantes:

- Facilidad de uso y desarrollo en todas las capas, particularmente en la capa media
- Interoperabilidad extendida
- Portabilidad y seguridad

La Figura 2.7 presenta los elementos de la plataforma Empresarial Java y su relación dentro del contexto de varias capas.

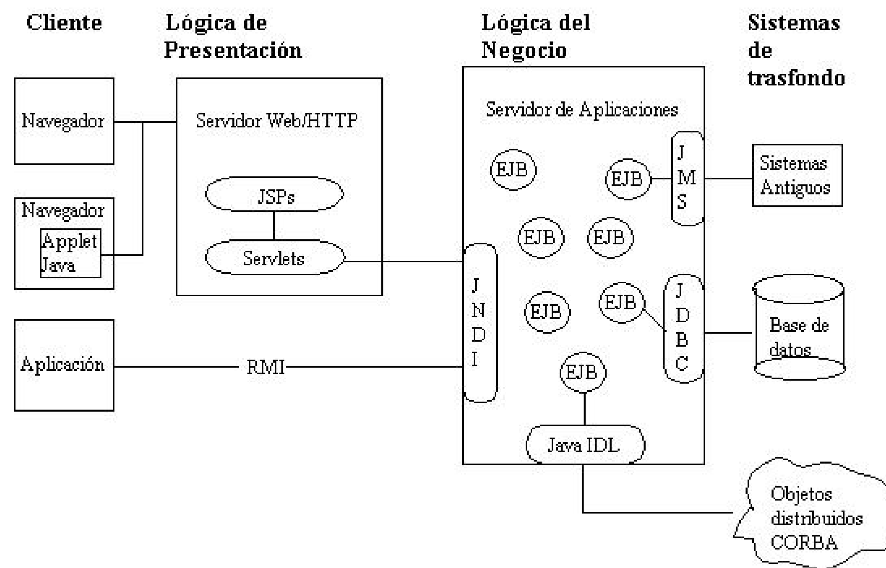

Figura 2.7: Componentes de la plataforma Java Empresarial

JavaBeans y Enterprise JavaBeans forman la arquitectura basada en componentes que promueve facilidad de uso y facilidad de desarrollo en todas las capas. Las APIs de la plataforma hacen posible interoperabilidad extendida con fuentes heterogéneas de información en la empresa incluyendo sistemas antiguos, fuentes de datos y servicios existentes. Portabilidad y seguridad son inherentes a la plataforma; el lenguaje Java es el

lenguaje más portátil en la historia de las computadoras y el modelo de seguridad Java es flexible y confiable.

Enterprise JavaBeans es una extensión a la arquitectura JavaBeans que facilita el desarrollo e implementación de aplicaciones distribuidas basadas en componentes.

Las APIs de la plataforma permiten utilizar la infraestructura de TI nueva y existente. Proporcionan acceso a bases de datos vía JDBC, servicios de nombres y directorios con Java Naming and Directory Interface, administración de sistemas y redes con la API Java Management, computación distribuida con Java IDL y RMI, demarcación de transacciones con Java Transaction Service, e interoperabilidad con sistemas de mensajería con Java Message Service.

A continuación examinamos con más detalle cada una de las APIs que forma parte de ésta plataforma empresarial.

2.4.1 Invocación de métodos remotos: Java RMI

Los desarrolladores pueden crear aplicaciones distribuidas para red con Java usando la Invocación de Métodos Remotos Java (Java Remote Method Invocation, Java **RMI**) para llamar objetos Java o CORBA desde otras máquinas virtuales Java, posiblemente en máquinas diferentes (ver Figura 2.8).

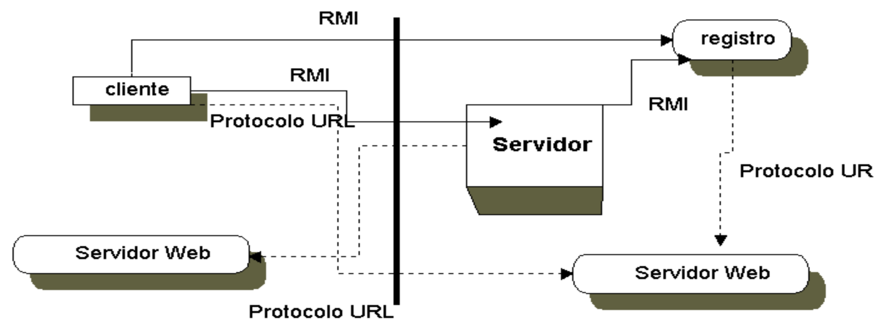

Figura 2.8: Ejemplo de interacción entre cliente y servidor usando RMI

Java RMI fue diseñada para permitir la utilización de diferentes protocolos subyacentes. Actualmente soporta dos protocolos para acceso remoto:

1. **JRMP**: El Protocolo Java de Métodos Remotos apoya todas las funciones de RMI, incluyendo el paso de objetos.
2. **IIOP**: El Protocolo Internet InterORB apoya casi todas las operaciones Java tal como JRMP, excepto el paso de objetos. Este protocolo fue diseñado para usarse con CORBA, el estándar industrial para computación heterogénea. IIOP permite a Java actuar como objetos CORBA, proporcionando fácil integración entre objetos CORBA y Java. Los desarrolladores no necesitan tratar con IDL porque está embebido dentro del marco.

En un entorno puramente Java, JRMP es preferido debido a que es poderoso y permite verdadero paso de objetos. En el caso de ambientes mixtos como objetos Java a CORBA, IIOP es preferido para portabilidad e interoperabilidad entre plataformas y lenguajes. En cualquier caso, Java RMI es la API para desarrollar programas para usar diferentes componentes y protocolos.

Un programa Java puede efectuar una llamada sobre un objeto remoto una vez que obtiene una referencia a ese objeto remoto. Esto puede hacerse buscando el objeto remoto en el servicio de nombres proporcionado por RMI o recibiendo la referencia como un argumento o un valor retornado. Un cliente puede llamar un objeto remoto en un servidor y ese servidor puede ser a su vez cliente de otros objetos remotos. RMI utiliza ***Serialización de Objetos*** para emitir y receptar parámetros y no altera los tipos de datos, apoyando verdadero polimorfismo. IIOP apoya llamadas CORBA enviadas o recibidas por objetos Java.

2.4.2 Interoperabilidad con CORBA: Java IDL

Java **IDL** añade capacidad CORBA a la plataforma Java, proporcionando interoperabilidad y conectividad basadas en estándares. Java IDL permite que aplicaciones Java distribuidas invoquen transparentemente operaciones sobre servicios de red remotos usando el estándar industrial IDL e IIOP definidos por el Object Management Group (OMG). Los componentes para ejecución incluyen un ORB Java completo para computación distribuida usando comunicación IIOP.

2.4.2.1 Sobre CORBA

CORBA es la arquitectura estándar de objetos distribuidos desarrollada por el consorcio OMG. Desde 1989 la misión de OMG ha sido la especificación de una arquitectura para un bus de software abierto, u Object Request Broker (ORB), en el que objetos escritos por diferentes vendedores pueden interoperar sobre redes y sistemas operativos. Este estándar permite que objetos CORBA invoquen a otros sin tener que saber donde residen los objetos que usan o en que lenguaje están implementados los objetos que solicitan. El Lenguaje de Definición de Interfaz (IDL) especificado por OMG se utiliza para definir las interfaces a objetos CORBA.

Los objetos CORBA difieren de los típicos objetos en que:

- Los objetos CORBA pueden ser colocados en cualquier parte de la red.
- Los objetos CORBA pueden interoperar con objetos en otras plataformas.
- Los objetos CORBA pueden ser escritos en cualquier lenguaje de programación para el cual exista una correspondencia entre OMG IDL y ese lenguaje.

2.4.2.2 Sobre Java IDL

Java IDL es un ORB proporcionado con el SDK 1.2. Junto con el compilador ***idltojava***, puede ser usado para definir, implementar y utilizar objetos CORBA desde el lenguaje Java. Java IDL satisface la especificación CORBA/IIOP 2.0.

El ORB Java IDL apoya objetos CORBA transitorios; es decir objetos cuya vida está limitada por el tiempo de vida su proceso servidor. Java IDL proporciona también un servidor de nombres transitorio para organizar los objetos en una estructura de árbol de directorios. El servidor de nombres satisface la especificación de Servicio de Nombres.

Un cliente solo puede invocar los métodos especificados en la interfaz del objeto CORBA. La interfaz del objeto CORBA se define usando IDL. Una interfaz define el tipo de objeto y especifica un conjunto de métodos y parámetros junto con los tipos de excepciones que estos métodos pudiesen devolver. Un compilador IDL como ***idltojava*** traduce las definiciones de objeto CORBA en un lenguaje de programación específico de acuerdo con la correspondencia indicada por el OMG. Así, el compilador ***idltojava*** traduce las

definiciones IDL en construcciones Java de acuerdo a la correspondencia IDL - lenguaje Java.

2.4.3 Acceso a bases de datos: JDBC

La API **JDBC** proporciona a los programadores Java una interfaz uniforme hacia un amplio rango de bases de datos relacionales. Es una interfaz basada en SQL que proporciona independencia de la base de datos a aplicaciones Java (ver Figura 2.9). Está implementada sobre la API de Manejador SQL Java, la cual a su vez es apoyada por una variedad de controladores de bases de datos.

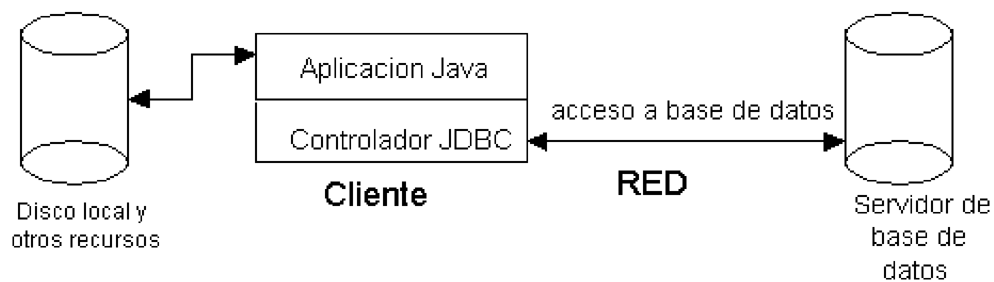

Figura 2.9: Ejemplo del uso de JDBC para acceder a un Servidor de Base de Datos

JDBC define clases Java para representar conexiones a bases de datos, instrucciones SQL, conjuntos de resultados, metadatos de base de datos, RPC, etc. Permite a un programador Java enviar instrucciones SQL y procesar los resultados. Es implementada a través de un administrador que puede soportar varios controladores conectados a diferentes bases de datos. Los controladores JDBC pueden estar completamente escritos en Java para poder descargarlos como parte de un applet o pueden ser implementados utilizando métodos nativos como puente hacia bibliotecas existentes para acceso a bases de datos.

JDBC proporciona una base común sobre la que se pueden construir herramientas e interfaces de nivel superior. Dado que JDBC es una API independiente del DBMS y que el SDK incluye un Administrador de Controlador JDBC, las aplicaciones JDBC pueden usar cualquier base de datos para la cual esté disponible un controlador JDBC. Esto incluye bases de datos relacionales y no relacionales y combinadas con objetos.

JDBC es una parte estándar de la plataforma Java desde la versión 1.1. SDK incluye un controlador especial llamado Puente JDBC ODBC, que permite usar JDBC con la mayoría

de controladores ODBC. Este controlador especial proporciona a las aplicaciones Java acceso inmediato a cualquier base de datos que satisface ODBC.

2.4.4 Servicios de Nombres y Directorios: JNDI

La Interfaz Java para Nombres y Directorios (**JNDI**) es una API que proporciona una interfaz unificada a múltiples servicios de nombres y directorios en la empresa. Usando JNDI, las aplicaciones Java pueden guardar y recuperar objetos Java de cualquier tipo. Además, JNDI proporciona métodos para ejecutar operaciones estándar de directorios de nombres, tales como asociar atributos con objetos y buscar objetos usando sus atributos y asignar correspondencia entre los nombres de máquinas y las direcciones de red.

La Figura 2.10 muestra cómo la Interfaz Java para Nombres y Directorios incluye una Interfaz de Proveedor de Servicio (SPI) que capacita a los proveedores de servicio de nombres y directorios a desarrollar, conectar y permitir operaciones cooperativas entre sus implementaciones a fin de que los servicios correspondientes estén disponibles desde aplicaciones que utilicen la API JNDI.

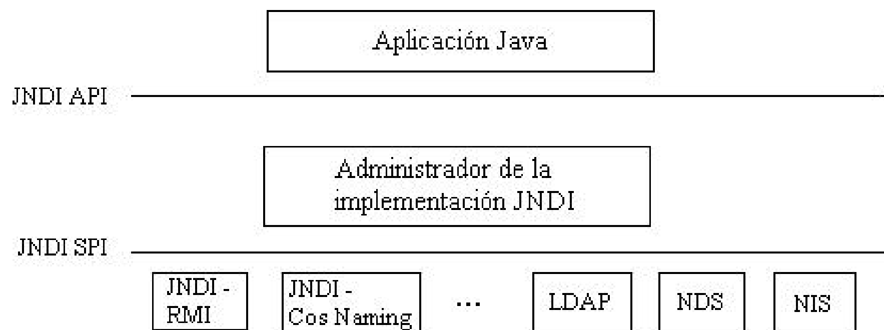

Figura 2.10: Elementos del Servicio de Nombres y Directorios

JNDI está definida independientemente de cualquier implementación específica de servicio de nombres y directorios. Permite que aplicaciones Java usen diferentes, quizá múltiples, servicios de nombres y directorios utilizando una API común. Diferentes proveedores de servicios de nombres y directorios pueden ser conectados de forma transparente detrás de ésta API. Esta independencia permite a las aplicaciones Java tomar ventaja de información en una variedad de servicios existentes de nombres y directorios como LDAP, NDS, DNS y NIS, además permite a las aplicaciones Java coexistir con aplicaciones y sistemas antiguos.

Los desarrolladores que empleen JNDI pueden producir consultas que usen LDAP u otros protocolos de acceso para recuperar resultados; sin embargo, no están limitados a LDAP ni tienen que desarrollar sus aplicaciones apoyados en LDAP.

JNDI proporciona la interfaz que especifica un contexto de nombres. También define operaciones básicas como añadir un enlace nombre-objeto, buscar el objeto enlazado a un nombre especificado, listar los enlaces, remover un enlace nombre-objeto, crear y destruir subcontextos del mismo tipo, etcétera.

2.4.5 Procesamiento de transacciones: JTS

El Servicio de Transacciones Java (**JTS**) es la API Java de enlace con OTS (Object Transaction Service). OTS es la API estándar de CORBA para administración de transacciones distribuidas. Igual que OTS, JTS está basada en el modelo de procesamiento distribuido de transacciones (DTP) de X/Open. JTS proporciona un mecanismo para conectar recursos XA en diferentes máquinas para una transacción y hace que el código sea portátil entre administradores de transacciones OTS.

Como se mencionó antes, los programadores EJB no necesitan hacer llamadas directas a JTS. JTS es una API de bajo nivel ideada para administradores de recursos y programadores de monitores de transacciones. El Servicio de Transacciones Java reemplaza las interfaces procedimentales XA y TX con un nuevo conjunto de interfaces CORBA IDL. Al mantener compatibilidad completa con aplicaciones antiguas basadas en DTP, JTS es capaz de importar y exportar transacciones desde y hacia administradores de recursos que cumplen XA (bases de datos) y administradores de transacciones que cumplen con TX.

Los desarrolladores que deseen programar directamente con JTS u OTS estarán en capacidad de utilizar servicios de transacciones basados en CORBA de varios vendedores.

2.4.6 Middleware orientado a mensajes: Servicio Java de Mensajes

La API de Servicio de Mensajes Java (**JMS**) es utilizada para comunicarse con middleware orientado a mensajes. Soporta la implementación de mensajería punto a punto (PTP) que es similar al correo electrónico directo con otra persona. También soporta la mensajería publicar/suscribir, que es similar a un grupo de noticias.

Proporciona apoyo a entrega garantizada de mensaje, entrega de mensaje basada en transacciones, mensajes persistentes y suscriptores duraderos. JMS proporciona otro medio para integrar los nuevos sistemas con los antiguos.

JMS es una forma asincrónica de despachar y recibir bloques de datos ("*mensajes*") desde el proveedor middleware con entrega garantizada. Su potencial se muestra en un mundo de componentes empresariales que confían en flujos de datos y notificación entre componentes, sin la sobrecarga de CORBA y el bloqueo en una respuesta que no se necesita inmediatamente.

Por lo que va en el mensaje real, JMS y el middleware orientado a mensajes (MOM) en general, está diseñado principalmente para mensajes entre componentes reales en lugar de mensajes "humanizados", lo que no significa que lo último no sea una aplicación válida.

El verdadero poder de JMS está en enviar flujos de bytes, pares nombre - valor y cualquier objeto Java encapsulado entre componentes. El proveedor MOM real que se elija maneja el enrutamiento, la entrega y la transferencia en red de los mensajes.

Existen algunos escenarios en la empresa donde hay sobrecarga por solicitudes de objetos y operaciones intensivas de recursos, las cuales podrían usar JMS para notificar a otros procesos inhibirse de actualizar.

Las arquitecturas que requieren que datos críticos sean propagados a diferentes departamentos y servidores DBMS, ciertamente se podrían beneficiar de un modelo basado en JMS donde los administradores reciben los mensajes y determinan qué entradas necesitan aprobación.

2.4.7 Servicios Web basados en Java: JSP y Java Servlet

2.4.7.1 Páginas Java de Servidor (JSP)

JSP es el equivalente de plataforma independiente de las Páginas Activas de Servidor (ASP) de Microsoft. Fueron diseñadas para ayudar a los desarrolladores de contenido Web a crear páginas dinámicas con relativamente poco código. Una página Java de servidor consiste de código HTML entremezclado con código Java. El servidor procesa el código

Java cuando la página es solicitada por el cliente, retornando la página HTML generada hacia el navegador.

Se trata esencialmente de un documento HTML, con marcas especiales JSP, que es compilado en un servlet. Si se usan las marcas JSP para llamar a componentes empresariales, la lógica del negocio es separada de HTML.

Java es más poderoso que el lenguaje de guiones ASP y tiene la ventaja de apoyar muchas plataformas. Los componentes Java son más fáciles de desarrollar que los objetos COM y pueden ser distribuidos a través de varios servidores diferentes sin la necesidad de recompilar. ASP ofrece persistencia de datos limitada entre páginas, mientras las páginas JSP pueden compartir datos usando el mismo componente Java. Finalmente, aunque ambas pueden estar mezcladas con código en una página Web, solo JSP permite remover el código de la página usando marcas.

Si se desea mantener la lógica de presentación separada de la lógica del negocio y no se desea las desventajas de los applets, ASP, plantillas, CGI o servlets puros, la opción viable es JSP usado con EJB y HTML/XML.

2.4.7.2 Java servlets

Un servlet proporciona mucha de la funcionalidad de una JSP, aunque toma un enfoque diferente. Mientras las páginas Java de servidor consisten generalmente de código HTML mayoritario entremezclado con pequeñas cantidades de código Java, los servlets, por otro lado, están escritos totalmente en Java y producen código HTML.

Un servlet es un pequeño programa Java que extiende la funcionalidad de un servidor Web. Por ejemplo, la Figura 2.11 muestra un servlet responsable por tomar los datos de una forma HTML y aplicar la lógica del negocio usada para actualizar la base de datos del inventario de una compañía.

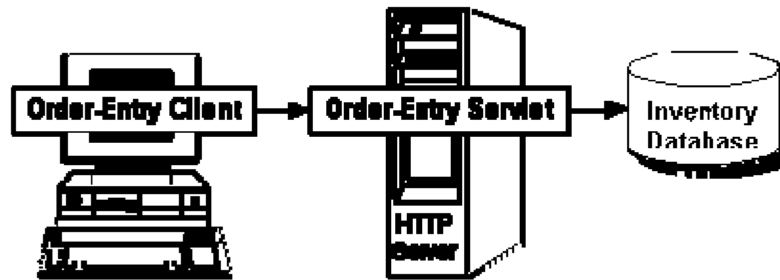

Figura 2.11: Ejemplo de uso de servlet

Un servlet es una aplicación del lado del servidor ejecutada dinámicamente bajo solicitud, parecido a los guiones Perl CGI en servidores Web más tradicionales. Una de las mayores diferencias entre los guiones CGI y los servlets es que los guiones CGI requieren todo un nuevo proceso para ser iniciados (incurriendo en sobrecarga adicional) mientras que los servlets son ejecutados como un hilo separado dentro del motor servlet. Por lo mismo, los servlets tienden a ofrecer escalabilidad mejorada.

Los servlets avanzan un paso por delante de los programas CGI, pero no separan la presentación de la lógica del negocio. El HTML en los servlets es a menudo rudo y difícil de mantener y requiere un cambio de código cada vez que hay una actualización. Se puede usar servlets para tareas no HTML, como extender la funcionalidad de un servidor Web o generar imágenes dinámicas u otros datos binarios.

2.4.8 Servicios de seguridad: la API de Seguridad Java

Como se mencionó anteriormente cuando se trató sobre los servicios de un infraestructura empresarial, la seguridad de la red requiere soluciones en cuatro áreas generales: autenticación, autorización, integridad y privacidad.

Autenticación es el medio con que el sistema reconoce al usuario ingresando una transacción o leyendo datos. Esta área es considerada el perímetro del sistema o muro de fuego, dado que es la primera línea de defensa contra entradas ilegales.

Autorización es el medio con que el sistema determina que datos podría usar un usuario en particular y por cuanto tiempo y bajo que condiciones podría continuar usándolos. La autorización es usualmente realizada con listas de control de acceso a archivos u objetos o mecanismos al nivel de aplicación que son sensibles a la semántica de los datos usados.

Integridad es el medio que usa el sistema para garantizar que los datos no son corrompidos o modificados maliciosamente en su almacén o durante su tránsito sobre la red. Esto incluye protección usando sumas de verificación o mensajes firmados.

Privacidad es el medio por el cual el sistema previene la revelación de datos sensitivos como números de tarjeta de crédito y previene la exposición de las relaciones entre usuarios y sistemas. (La privacidad se logra usualmente con autorización o encriptación.)

Una solución segura puede incluir una o más de las siguientes:

- Perímetro de defensa (muro de fuego),
- Encriptación en la capa de red como los protocolos Simple Key Management for Internet (SKIP) los que aseguran la red al nivel de paquete IP, agregando encriptación a aplicaciones de red existentes,
- Protecciones en la capa de sesión como SSL (Secure Socket Layer), e
- Implementaciones en la capa de aplicación.

La Tabla 2.1 delinea varios métodos que pueden ser empleados para construir sistemas seguros.

Tabla 2.1: Métodos para asegurar sistemas

Para proporcionar	**Se utiliza**
Autenticación de usuario	La API Java Card, la API Java de Firma Digital o administración de certificados X.509
Autorización	Las APIs Java de Seguridad, ACL
Integridad	La API de Criptografía Java, la API Java de Firma Digital, encriptación en la capa de red, encriptación en la capa de aplicación con SSL o encriptación en la capa de sesión
Privacidad	Encriptación en la capa de red, encriptación en la capa de aplicación o encriptación en la capa de sesión usando las APIs de Criptografía Java y de Firma Digital

La seguridad se puede implementar en más de una capa; por ejemplo, un muro de fuego que asegura la intranet, junto con seguridad en la capa de sesión para proteger las transacciones individuales que llegan de clientes o socios de negocios. Un sistema completo aunque manejable de autenticación de usuarios otorga a los usuarios internos y los socios externos acceso a los datos en red.

Para reforzar los aspectos de integridad o privacidad, a menudo la mejor solución es la seguridad en la capa de aplicación. En la capa de aplicación, las APIs de criptografía Java que implementan DES son probablemente el camino más confiable hacia la encriptación. La API de criptografía permite a los programadores proteger datos individuales con alto grado de seguridad y confianza.

Las listas de control de acceso (ACL) son efectivas para el control de seguridad declarativa. Una aplicación basada en EJB puede especificar una ACL para cada componente empresarial. Una ACL especifica los roles del usuario (tareas funcionales, por ejemplo) que indican a quién se permite usar el componente.

2.4.9 XML y Java

Con los primeros sistemas de cómputo pocos desarrolladores se preocupaban por el intercambio de datos. Como consecuencia, las interfaces que creaban (si las creaban) eran a menudo *ad hoc* y usualmente diferían entre sistemas.

Todos los métodos para intercambiar información entre dos sistemas (incluidos RMI, RPC, CORBA y COM) tienen mucho en común. En esencia, ellos pasan información que ha sido reducida a un bloque de datos que, generalmente, no se describe a sí mismo. La clave sobre lo que significa el bloque (cuáles son los campos y cuál es el formato) está incorporada al código. El código es responsable por analizar el bloque de datos y validar la información en él contenida. Este enfoque presenta serios inconvenientes para expandir el modelo de datos.

XML comenzó su vida como un HTML *nuevo y mejorado* pero, con el tiempo ha encontrado espacio en dominios diferentes al de la publicación Web. XML y Java coinciden en algunos de esos otros dominios y pueden juntarse para el desarrollo de aplicaciones de nivel empresarial especialmente en el intercambio de datos.

2.4.9.1 APIs Java para XML

La utilización de XML para intercambio de datos plantea dos retos. Primero, aunque la generación de XML es un proceso sencillo, la operación inversa, utilizar datos XML desde dentro de un programa, no lo es. Segundo, las tecnologías XML actuales pueden usarse de forma inapropiada, lo que puede llevar al programador a crear un sistema lento y ávido de memoria.

Para que una aplicación Java pueda usar XML debe disponer de una API que le permita hacer lo siguiente:

1. Realizar transformaciones de objetos Java a documentos XML y
2. Convertir documentos XML en objetos Java.

Algunas herramientas actuales son mejores que otras para trabajar con XML. Vamos a examinar brevemente algunas APIs.

2.4.9.1.1 La API DOM

La API DOM es una API basada en un modelo de objetos. Los procesadores[3] XML que implementan DOM crean un modelo objeto genérico en memoria que representa el contenido del documento XML. Una vez que el procesador XML ha completado el reconocimiento, la memoria contiene un árbol de objetos DOM que proporcionan información tanto de la estructura como de los contenidos del documento XML.

El modelo DOM se generó en el mundo HTML, donde un modelo objeto documento común representa el documento HTML cargado en el navegador. Este DOM HTML se pone a disposición de lenguajes como JavaScript. DOM HTML ha tenido mucho éxito en este campo.

A primera vista, la API DOM se muestra con más características y por tanto mejor, que la API SAX que veremos a continuación. Sin embargo, DOM tiene serios problemas de eficiencia que pueden malograr aplicaciones donde es importante el desempeño. Los procesadores DOM implementan el modelo objeto en memoria creando muchos pequeños

[3] Un procesador XML verifica que los elementos del documento formen construcciones XML legales. Si se dispone de un Documento de Definición de Tipos (**DTD**) el procesador la puede usar para determinar si el documento está bien formado.

objetos que representan nodos DOM que contienen texto u otros nodos DOM. Esto parece normal, pero tiene implicaciones negativas en desempeño. Una de las operaciones más caras en Java es el operador ***new***. Eventualmente, cada llamada a ***new*** provocará una llamada al Recolector de Basura cuando se considera que el objeto ya no es útil. La API DOM tiende a colapsar la memoria con sus múltiples objetos pequeños.

Otra desventaja de DOM es el hecho de que carga el documento XML completo en memoria. Para documentos grandes esto es un problema.

Otro problema sutil es que se debe reconocer el documento dos veces. La primera pasada crea la estructura DOM en memoria, la segunda localiza todos los datos XML en los que está interesado el programa. En contraste, el estilo de codificación SAX refuerza la localización y recolección de datos XML en una sola pasada.

Algunos de estos inconvenientes podrían ser evitados con un mejor diseño de la estructura de datos subyacente para representar internamente el modelo objeto DOM.

2.4.9.1.2 La API SAX

La API SAX tiene importantes características que le proporcionan un buen desempeño. Con ella se puede unir Java y XML con un mínimo de memoria, incluso en las estructuras XML más complejas.

La API SAX está basada en eventos. Los procesadores XML que implementan la API SAX generan eventos que se corresponden con las diferentes características encontradas en el documento XML analizado. Respondiendo a éste flujo de eventos SAX en el código Java, se pueden escribir programas Java conducidos por datos XML.

Comparada con la API DOM, la API SAX es un enfoque atractivo. SAX no tiene un modelo objeto genérico, por tanto no tiene los problemas de memoria y desempeño asociados con abusar del operador ***new***. Con SAX se puede utilizar el modelo objeto que mejor se ajuste al domino de aplicación. Además, dado que SAX procesa el documento XML en una sola pasada, requiere mucho menos tiempo de procesamiento.

SAX posee algunas desventajas, pero están más relacionadas al programador y no al desempeño en tiempo de ejecución de la API.

La primera desventaja es conceptual. Los programadores están acostumbrados a navegar hasta encontrar los datos. Con SAX es a la inversa. Esto es, se establece un código que escucha la lista de cada pieza de datos XML disponible. Ese código se activa solo cuando datos XML interesantes son escuchados.

La segunda desventaja es más peligrosa. El código SAX debe ser diseñado considerando que el procesador SAX navega exhaustivamente la estructura XML mientras proporciona simultáneamente los datos almacenados en el documento XML. La mayoría de la gente se enfoca en los datos y dejan el aspecto de navegación a una lado, lo que provoca sutiles interacciones.

2.4.9.1.3 La API Jato

Jato es una API Java y lenguaje XML de código abierto para transformar documentos XML en un conjunto de objetos Java y viceversa. Los guiones Jato describen las operaciones a ejecutarse y dejan los algoritmos de implementación a un interprete. Un guión Jato expresa las relaciones entre elementos XML y objetos Java, liberando al programador de escribir lazos de repetición, rutinas recursivas, código de verificación de errores y muchas otras tareas monótonas y propensas a errores del reconocimiento XML.

Jato tiene muchas ventajas sobre APIs Java para XML como JDOM, SAX o DOM. Algunas incluyen:

- Jato alienta que los diseños XML y Java sean optimizados para sus tareas específicas. Los sistemas bien diseñados tienen un bajo grado de cohesión, permitiendo hacer cambios independientes a porciones del sistema sin romperlo. Por lo tanto, es buena idea que la DTD XML y el diseño orientado a objetos de Java se desarrollen e implementen de forma independiente.
- Con Jato, los desarrolladores simplemente expresan los elementos XML que corresponden con o desde clases Java específicas. El interprete Jato luego implementa los algoritmos necesarios de reconocimiento y generación para cumplir las acciones deseadas. Como tal, se evita código de reconocimiento y generación monótono, monolítico y difícil de mantener.
- Usar XML para describir las transformaciones desde o hacia XML en aplicaciones Java parece natural.

Las características de Jato y su diseño son el resultado directo de la implementación de proyectos del mundo real. Entre sus características más importantes están:

1. El código fuente de Jato está completamente disponible para su uso, modificación y mejora.
2. Los guiones Jato se leen en tiempo de ejecución y se emplea reflexión y JDOM para la transformación entre Java y XML. Otras APIs que realizan tareas similares necesitan que el programador cree un esquema, genere un conjunto de clases a partir del esquema, compile las clases y distribuya las clases compiladas.
3. La mayoría de aplicaciones del mundo real necesitan clases que usan constructores y métodos con parámetros y también métodos estáticos. Jato los puede invocar a todos.
4. Las transformaciones pueden condicionarse al tipo de un elemento XML, la presencia o valor de un atributo y el estado de un objeto.
5. Los desarrolladores pueden añadir fácilmente nuevas funciones al interprete Jato a través de marcas personalizadas. Haciendo posible, por ejemplo, hacer uso de bases de datos o crear formatos personales.

Capítulo 3

La arquitectura de Componentes Empresariales Java

3.1 Visión general

La arquitectura de Componentes Empresariales Java (**Enterprise JavaBeans, EJB**) es el núcleo de la plataforma Java Empresarial y permite que los desarrolladores escriban lógica del negocio reutilizable y portátil.

Los Componentes Empresariales Java no son un producto sino una especificación[4] que define su arquitectura y las interfaces entre los servidores EJB y los componentes. La especificación es una iniciativa liderada por Sun Microsystems Inc. y apoyada por los principales vendedores de la industria.

El modelo de componentes EJB extiende el modelo de componentes JavaBeans para apoyar **componentes de servidor**. Los componentes de servidor son piezas preconstruidas de funcionalidad reutilizable diseñadas para ejecutarse en un servidor de aplicaciones. Estos se pueden combinar con otros componentes para crear sistemas personalizados. Los componentes EJB no pueden ser manipulados por un IDE visual de Java en la forma en que lo son los JavaBeans. En lugar de eso, pueden ser ensamblados y personalizados en tiempo de publicación[5] usando herramientas proporcionadas por el servidor de aplicaciones compatible con EJB.

La arquitectura de componentes empresariales Java proporciona un marco integrado que simplifica dramáticamente el proceso de desarrollo de sistemas empresariales. Un servidor EJB administra automáticamente cierto número de servicios de capa media en nombre de los componentes. La aplicación se desarrolla más rápido porque los desarrolladores están concentrados en la lógica del negocio y no en los detalles de la compleja capa media.

[4] La Especificación EJB versión 2.0 es la última versión oficial aunque ya se ésta desarrollando una nueva que incorpora ciertas mejoras.
[5] Tiempo de publicación es el momento en que se indica al servidor de aplicaciones las características del componente para que éste lo ponga a disposición de sus usuarios.

3.2 Servicios implícitos

El modelo EJB apoya un número de servicios implícitos incluyendo ciclo de vida, administración de estado y persistencia.

- Ciclo de vida

  Los componentes empresariales no tienen necesidad de manejar explícitamente la asignación de procesos, administración de hilos concurrentes, activación de objetos o destrucción de objetos. El contenedor EJB administra automáticamente el ciclo de vida del objeto en nombre del componente empresarial.

- Administración de estado

  Los componentes empresariales no necesitan guardar o restaurar explícitamente el estado entre llamadas del objeto. El contenedor EJB administra automáticamente el estado del objeto en nombre del componente empresarial.

- Seguridad

  Los componentes empresariales no necesitan explícitamente autenticar usuarios o revisar los niveles de autorización. El contenedor EJB ejecuta automáticamente todas las verificaciones de seguridad en nombre del componente.

- Transacciones

  Los componentes empresariales no necesitan explícitamente especificar código de demarcación de transacciones para participar en transacciones distribuidas. El contenedor EJB puede administrar automáticamente el inicio, finalización, aceptación y rechazo de transacciones en nombre del componente empresarial.

- Persistencia

  Los componentes empresariales no necesitan recuperar o guardar explícitamente los datos persistentes del objeto desde una base de datos. El contenedor EJB puede administrar automáticamente los datos persistentes en nombre del componente.

3.3 Detalles arquitectónicos

Un servidor de aplicaciones compatible con Enterprise JavaBeans, llamado un **servidor EJB**, proporciona un ambiente que apoya la ejecución de aplicaciones desarrolladas

usando la tecnología EJB. Él administra y coordina la asignación de recursos a las aplicaciones. (ver Figura 3.1)

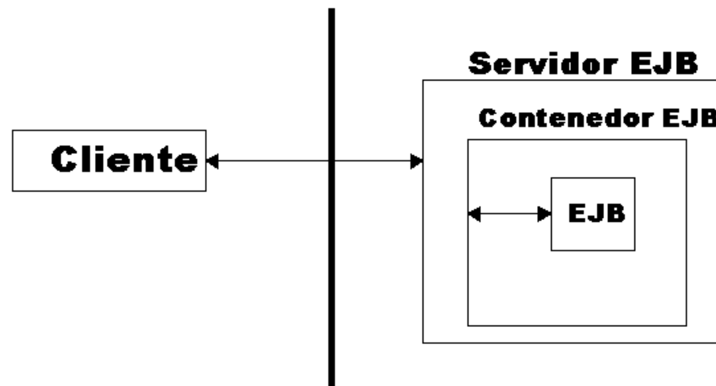

Figura 3.1: Esquema de la arquitectura EJB

El servidor EJB debe proporcionar uno o más **contenedores EJB**. Un contenedor EJB implementa servicios de administración y control para una o más clases EJB alojadas dentro de él. Para cada componente empresarial, el contenedor es responsable por registrar el objeto, proporcionar una interfaz remota para el objeto, crear y destruir instancias del objeto, verificar la seguridad del objeto, administrar el estado activo del objeto y coordinar transacciones distribuidas. Opcionalmente, el contenedor también puede manejar todos los datos persistentes dentro del objeto.

3.3.1 Objetos transitorios y persistentes

La tecnología de componentes empresariales Java soporta tanto objetos transitorios como persistentes. Un objeto transitorio es llamado un componente de ***sesión*** y un objeto persistente es llamado un componente ***entidad***.

3.3.1.1 Componentes de sesión

Un componente de sesión es creado por un cliente y en la mayoría de los casos existe únicamente durante una sola sesión cliente/servidor. Un componente de sesión ejecuta operaciones en nombre del cliente, tales como acceso a bases de datos o desarrollo de cálculos. Los componentes de sesión pueden participar en transacciones, pero (normalmente) no son recuperables luego de caer el sistema. Estos pueden no recordar su

estado o pueden mantenerlo a lo largo de métodos y transacciones. El contenedor administra el estado de un componente de sesión si éste necesita ser desalojado de memoria. Un componente de sesión debe manejar sus propios datos persistentes.

3.3.1.2 Componentes entidad

Un componente entidad es un objeto que representa datos persistentes mantenidos en un almacenamiento permanente como una base de datos. Una clave primaria identifica a cada instancia de un componente entidad. Estos pueden ser creados insertando datos directamente en la base de datos o creando un objeto (por medio de un método de construcción). Los componentes entidad participan en transacciones y se pueden recuperar luego de una caída del sistema.

Los componentes entidad pueden administrar su propia persistencia o pueden delegar los servicios de persistencia a su contenedor. Si el componente delega la persistencia al contenedor, entonces el contenedor ejecuta automáticamente todas las operaciones de recuperación y almacenamiento de datos en nombre del componente.

3.3.2 Formas de interacción

Un contenedor EJB maneja los componentes empresariales que son publicados dentro de él. Las aplicaciones clientes no usan directamente un componente empresarial. En su lugar, la aplicación cliente utiliza el componente empresarial a través de dos interfaces que son generadas por el contenedor: la interfaz ***EJBHome*** y la interfaz ***EJBObject*** (ver Figura 3.2). Cuando el cliente invoca operaciones usando las interfaces, el contenedor intercepta cada llamada a método e inserta los servicios de administración.

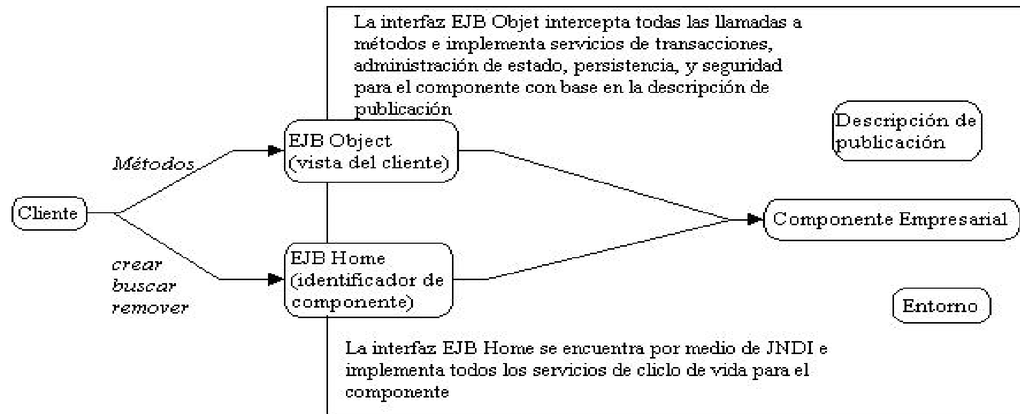

Figura 3.2: Interacción entre un cliente y un contenedor EJB

3.3.2.1 EJBHome

La interfaz EJBHome (**local**) proporciona acceso a los servicios de ciclo de vida del componente. Los clientes pueden utilizar la interfaz local para crear y destruir instancias del componente. Para componentes entidad, la interfaz local también proporciona uno o más métodos de búsqueda que permiten a un cliente encontrar una instancia existente del componente y recuperarla desde su almacenamiento persistente.

Para cada clase instalada en el contenedor, éste automáticamente registra la interfaz EJBHome en un directorio usando la API JNDI. Empleando JNDI, cualquier cliente puede localizar la interfaz EJBHome para crear una nueva instancia del componente o para buscar una instancia de componente entidad existente. Cuando un cliente crea o encuentra un componente, el contenedor regresa una interfaz EJBObject.

3.3.2.2 EJBObject

La interfaz EJBObject (**remota**) proporciona acceso a los métodos de la lógica de negocio dentro del componente empresarial. Un EJBObject representa la perspectiva de un cliente del componente empresarial. El EJBObject expone todas las interfaces del objeto relacionadas a la aplicación, pero no las interfaces que permiten al contenedor EJB administrar y controlar el objeto. La envoltura EJBObject permite al contenedor EJB interceptar todas las operaciones hechas sobre el componente empresarial. Cada vez que un cliente invoca un método sobre el EJBObject, la solicitud pasa a través del contenedor EJB

antes de ser delegada al componente empresarial. El contenedor EJB implementa transparentemente los servicios de administración de estado, control de transacciones, seguridad y persistencia tanto para el cliente como para el componente empresarial.

3.3.3 Atributos declarativos

Las reglas asociadas con la manipulación de ciclo de vida, transacciones, seguridad y persistencia del componente empresarial están definidas en un objeto asociado de *Descripción de Publicación*. Estas reglas son declaradas al momento de publicación en lugar de programarse dentro del código. Al momento de ejecutar, el contenedor EJB automáticamente implementa los servicios de acuerdo a los valores especificados en el objeto de descripción de publicación asociado con el componente empresarial.

3.3.4 Objeto de contexto

Para cada instancia activa del componente empresarial, el contenedor EJB genera un objeto de contexto de la instancia para mantener información sobre las reglas de administración y el estado actual de la instancia. Un componente de sesión utiliza un objeto ***SessionContext*** y un componente entidad emplea un objeto ***EntityContext***. El objeto de contexto es usado por ambos, el contenedor EJB y el componente, para coordinar transacciones, seguridad, persistencia y otros servicios del sistema. También asociada con cada componente empresarial está una tabla de propiedades llamada el objeto de *Entorno*. El objeto de Entorno contiene los valores de las propiedades personalizadas fijadas durante el proceso de ensamblaje de la aplicación o en el proceso de publicación del componente empresarial.

3.3.5 Protocolos de comunicación

La tecnología Enterprise JavaBeans utiliza la API RMI para proporcionar acceso a los componentes empresariales. Sin embargo, la especificación no establece un protocolo específico y RMI puede ser usado sobre múltiples protocolos.

El protocolo nativo para RMI es el Java Remote Method Protocol pero se puede usar el protocolo CORBA estándar para comunicaciones Internet InterORB Protocol (**IIOP**). Usando IIOP, los componentes pueden unirse a clientes y servidores en lenguaje nativo. IIOP permite una fácil integración entre sistemas CORBA y sistemas EJB. Los componentes empresariales pueden usar servidores CORBA y los clientes CORBA pueden

usar componentes empresariales. Utilizando el servicio COM/CORBA Internetworking, los clientes ActiveX pueden usar los componentes empresariales y los componentes empresariales pueden usar servidores COM.

3.3.6 Administración del estado

Un componente de sesión representa el trabajo que está haciendo un cliente individual. En algunos casos, el trabajo se puede realizar completamente dentro de una sola llamada a método. En otros casos, el trabajo pudiese esparcirse entre varias llamadas. De ser éste el caso, el objeto debe mantener el estado entre llamadas. Por ejemplo, en una aplicación de comercio electrónico, el método VerificaAutorizaciónDeCrédito ejecuta un trabajo y no requiere mantener el estado. El objeto CocheDeCompra, por otro lado, debe mantener el registro de todos los elementos seleccionados mientras el comprador está navegando y hasta que esté listo para comprar.

Las opciones de administración de estado para un componente de sesión se definen en la descripción de publicación. Todos los componentes entidad deben conservar el estado.

Si un objeto no tiene estado, el contenedor automáticamente elimina el estado dentro de la instancia después de cada llamada a método. Si el objeto conserva el estado, el contenedor mantiene automáticamente el estado del objeto entre llamadas hasta que el objeto de sesión es destruido, aun si el objeto es removido temporalmente de la memoria. La arquitectura EJB proporciona un modelo sencillo de programación para permitir a los programadores especificar funciones a invocar cuando los objetos son cargados o removidos de la memoria.

3.3.7 Servicios de persistencia

La tecnología EJB proporciona un modelo simple para administrar la persistencia de objetos. Las funciones de persistencia deben ser ejecutadas siempre que los objetos se crean o destruyen o cuando los objetos son cargados o removidos de la memoria.

Un objeto entidad puede manejar su propia persistencia o puede delegarla a su contenedor. Las opciones de persistencia para un componente entidad son definidas en la descripción de publicación.

3.3.7.1 Persistencia Manejada por el Componente, BMP

Si el objeto entidad maneja su persistencia, entonces el desarrollador del componente debe implementar las operaciones de persistencia directamente en los métodos de la clase del componente empresarial.

3.3.7.2 Persistencia Manejada por el Contenedor, CMP

Si el objeto entidad delega los servicios de persistencia, el contenedor EJB administra transparente e implícitamente la persistencia. El desarrollador del componente empresarial no necesita codificar ninguna función de acceso a bases de datos dentro de los métodos de la clase del componente.

3.3.8 Administración de transacciones

Aunque el modelo EJB puede ser utilizado en sistemas sin transacciones, la arquitectura fue diseñada para apoyarlas. La tecnología EJB requiere el uso de un sistema de administración de transacciones distribuidas que apoye protocolos de aceptación en dos fases para transacciones llanas.

La especificación EJB sugiere, pero no requiere, transacciones basadas en la API Java Transaction Service (**JTS**). JTS es la respuesta Java a OTS de CORBA. JTS apoya transacciones distribuidas que pueden involucrar múltiples bases de datos sobre múltiples sistemas coordinadas por múltiples administradores de transacciones.

Las aplicaciones EJB se comunican con servicios de transacciones usando la Java Transaction API (**JTA**). JTA proporciona una interfaz de programación para iniciar transacciones, unirse a transacciones existentes, aceptar transacciones y deshacer transacciones.

Los componentes empresariales no necesitan colocar ninguna instrucción de demarcación de transacciones en su código. La semántica de transacciones se declara y no se programa.

3.3.9 Seguridad

El modelo EJB utiliza los servicios de seguridad Java disponibles en Java 2. La seguridad en Java permite servicios de autenticación y autorización para acceso restringido a objetos y métodos seguros.

La tecnología EJB automatiza el empleo de la seguridad Java a fin de que los componentes no necesiten rutinas explícitas con código de seguridad. Las reglas de seguridad para cada componente empresarial son declaradas en la descripción de publicación.

3.4 Roles y ciclo de vida de la aplicación

La arquitectura EJB define seis roles distintos para el ciclo de vida de desarrollo y publicación de la aplicación. Cada **Rol EJB** puede ser desempeñado por una entidad diferente. La arquitectura EJB especifica los contratos que aseguran que la producción de cada rol sea compatible con la producción de los otros.

Dentro de algunos escenarios, una sola entidad podría cumplir varios roles de la arquitectura. Por ejemplo, el Proveedor del Contenedor y el Proveedor del Servidor EJB pudiesen ser el mismo. O un solo programador podría desempeñar los roles de Proveedor de Componentes y Ensamblador de Aplicaciones.

En la Figura 3.3 se muestra la relación que existe entre los diferentes roles.

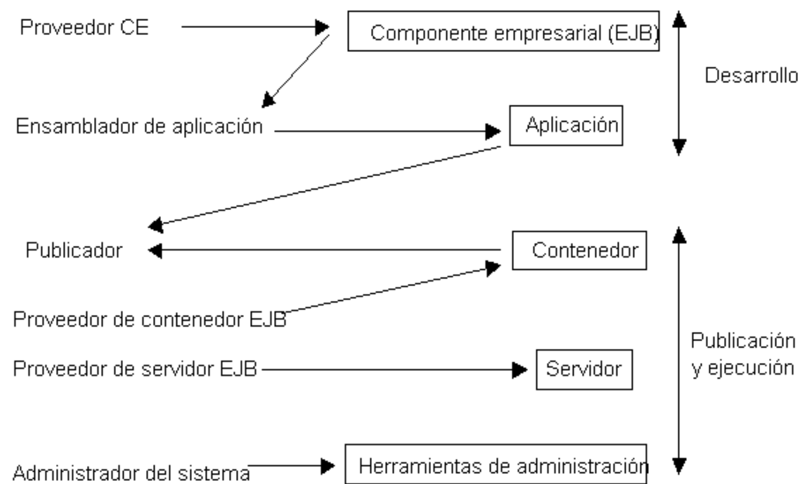

Figura 3.3: Roles de la arquitectura EJB

En pocas palabras, el *Proveedor de Componentes Empresariales* desarrolla los componentes empresariales, el *Ensamblador de Aplicaciones* escribe la parte cliente de la aplicación, el *Publicador* instala los componentes en la plataforma EJB, mientras que el *Administrador del Sistema* hace su trabajo habitual empleando las herramientas de vigilancia y administración de la plataforma EJB. El *Proveedor del Servidor EJB* y el *Proveedor del Contenedor EJB* son generalmente el mismo.

3.4.1 Proveedor de componentes empresariales

El proveedor de componentes empresariales es generalmente el programador que desarrollará los componentes de software. El resultado de su trabajo es un archivo que contiene los componentes. Él es responsable por implementar el comportamiento del componente de software y el contrato con el contenedor. Esto último es un conjunto de reglas y métodos a implementar a fin de que el componente sea ejecutable dentro de la plataforma EJB. Detalles de las responsabilidades del Proveedor de componentes se tratan en la sección *La Interfaz de Programación de Aplicaciones EJB*.

3.4.2 Ensamblador de aplicaciones

El ensamblador de aplicaciones utiliza componentes empresariales existentes para componer aplicaciones. Él trabaja con las interfaces remota y local del componente para desarrollar "aplicaciones cliente" (desde el punto de vista del servidor EJB). Tales aplicaciones cliente pueden ser nuevos componentes empresariales, servlets, applets, etc. Más detalles se tratan en la sección *Ensamblaje de aplicaciones*.

3.4.3 Publicador

El publicador instala los componentes empresariales en la plataforma EJB. Él hace uso de diferentes herramientas proporcionadas por aquella a fin de alcanzar éste propósito. La instalación del componente empresarial pudiese consistir en modificar sus propiedades de entorno a fin de adaptarlo al ambiente de ejecución específico (seguridad, base de datos, soporte…). La implementación es un proceso dependiente del servidor EJB.

3.4.4 Proveedores de servidor y contenedor EJB

El servidor y contenedor EJB son conceptos dedicados a los proveedores de la plataforma EJB. Por lo tanto no hay secciones que cubran esos tópicos en éste documento.

Los proveedores de servidor EJB son aquellos que proporciona los servicios básicos de la plataforma tales como apoyo para transacciones distribuidas y persistencia. La interfaz correspondiente es usada por los contenedores.

Los proveedores de contenedor son personas que proporcionan el medio para mezclar el comportamiento del componente con programación de nivel de sistema (administración de transacciones y persistencia…).

Generalmente, los proveedores de contenedor proporcionan herramientas de generación que permitirán que un componente sea ejecutable sobre un servidor EJB.

3.4.5 Administrador del sistema

El administrador del sistema es responsable por el comportamiento del entorno de ejecución. Él podría afinar el servidor EJB a través de herramientas de vigilancia y administración. Las herramientas proporcionadas dependen igualmente del servidor EJB específico que se esté utilizando.

3.5 La Interfaz de Programación de Aplicaciones EJB

Un componente empresarial está compuesto de las siguientes partes, que serán desarrolladas por el proveedor de componentes empresariales:

- La **interfaz local** contiene todos los métodos de ciclo de vida del componente (creación, supresión) y aquellos de búsqueda de instancia (encontrar uno o más objetos) usados por la aplicación cliente.
- La **interfaz remota** es la vista cliente del componente. Contiene toda la "lógica del negocio".
- La **clase clave primaria** (solo para componentes entidad) contiene un subconjunto de las propiedades del componente que identifican una instancia particular del componente entidad. Esta clase es opcional dado que el programador del componente puede elegir una clase estándar (por ejemplo `java.lang.String`) en su lugar.
- La **clase de implementación del componente**, la cual implementa la lógica del negocio y todos los métodos (descritos en la especificación EJB) que permiten al componente ser manejado por el contenedor.

- El **descriptor de publicación** que contiene las propiedades del componente que pueden ser editadas al momento de ensamblaje o publicación.

En la siguiente figura se puede apreciar la interacción entre los diferentes elementos de la interfaz de programación de aplicaciones EJB.

Figura 3.4: Interacción de los componentes de la API EJB

3.5.1 Desarrollo de Componentes de Sesión

Un objeto de componente de sesión es un objeto de vida corta que se ejecuta en nombre de un solo cliente. Hay componentes de sesión **con y sin estado**. Los componentes *sin estado* no mantienen su estado entre las llamadas a métodos. Cualquier instancia de componentes sin estado puede ser usada por cualquier cliente en cualquier momento. Los componentes de sesión *con estado* mantienen su estado dentro y entre transacciones. Cada objeto componente de sesión con estado está asociado con un cliente específico.

Un componente de sesión con estado que usa demarcación de transacciones del contenedor puede opcionalmente implementar la interfaz **SessionSynchronization**. En éste caso, los objetos componentes serán informados de los límites de la transacción. Una vuelta hacia atrás podría resultar en el estado de un objeto componente de sesión inconsistente; así, implementando la interfaz **SessionSynchronization** pudiese permitir al objeto componente actualizar su estado de acuerdo al estado de la transacción.

3.5.1.1 La interfaz local

La interfaz local debe extender la interfaz **javax.ejb.EJBHome**. Esta interfaz define uno o más métodos `create(...)`. Cada método `create` debe ser llamado así y debe emparejar uno de los métodos `ejbCreate` definidos en la clase componente empresarial. El tipo de retorno de un método de creación debe ser el tipo de la interfaz remota del componente empresarial.

Todas las excepciones definidas en la cláusula ***throws*** de un método `ejbCreate` deben estar definidas en la cláusula ***throws*** del método `create` que empareja en la interfaz local.

En los siguientes ejemplos usaremos un componente de sesión llamado Op.

Ejemplo:

```
public interface OpHome extends EJBHome {
      Op create(String user) throws CreateException,
                                    RemoteException;
   }
```

3.5.1.2 La interfaz remota

La interfaz remota es la perspectiva del cliente de una instancia del componente de sesión. Esta interfaz contiene la lógica del negocio del componente empresarial. Debe extender la interfaz **javax.ejb.EJBObject**. Los métodos definidos en ésta interfaz deben seguir las reglas de Java RMI (o sea, sus argumentos y el tipo a retornar deben ser tipos válidos para Java RMI y su cláusula ***throws*** debe incluir **java.rmi.RemoteException**). Por cada método definido en la interfaz remota, debe existir un método correspondiente en la clase del componente empresarial (mismo nombre, mismo tipo y número de argumentos, mismo valor de retorno y lista de excepciones, menos **RemoteException**).

Ejemplo:

```
public interface Op extends EJBObject {
     public void buy(int shares) throws RemoteException;
     public int read() throws RemoteException;
}
```

3.5.1.3 La clase del componente empresarial

Esta clase implementa la lógica del negocio de la interfaz remota y los métodos de la interfaz **javax.ejb.SessionBean** que están dedicados al entorno EJB. La clase debe ser

definida como pública y no debería ser abstracta. Los métodos de la interfaz **SessionBean** que el proveedor del componente debe desarrollar son los siguientes:

- public void **setSessionContext**(SessionContext ic);

  Usado por el contenedor para pasar una referencia al contexto de sesión de la instancia componente. El contenedor invoca éste método sobre una instancia luego que ella ha sido creada. Generalmente almacena la referencia en una variable de la instancia.

- public void **ejbRemove**();

  Este método es invocado por el contenedor cuando la instancia está en el proceso de ser removida del contenedor. Puesto que la mayoría de componentes de sesión no tienen ningún recurso que retirar, generalmente se deja vacío.

- public void **ejbPassivate**();

  Este método es invocado por el contenedor cuando desea poner a "hibernar" la instancia. Luego de completado, la instancia debe estar en un estado que permita al contenedor usar el protocolo de Serialización Java para almacenar el estado de la instancia.

- public void **ejbActivate**();

  Este método es invocado por el contenedor justo después de haber reactivado la instancia. La instancia debería adquirir cualquier recurso que liberó previamente en el método *ejbPassivate()*.

Un componente de sesión con estado y persistencia manejada por el contenedor puede implementar opcionalmente la interfaz **javax.ejb.SessionSynchronization**. Esta interfaz puede proporcionar al componente notificación sobre sincronización de transacciones. Los métodos de la interfaz **SessionSynchronization** que el proveedor EJB debe desarrollar son los siguientes:

- public void **afterBegin**();

  Este método notifica a una instancia de componente de sesión que ha comenzado una nueva transacción. En éste punto la instancia ya está en la transacción y podría hacer cualquier trabajo requerido dentro del alcance de la transacción.

- public void **afterCompletion**(boolean committed);

  Este método notifica a una instancia de componente de sesión que una transacción ha acabado y le dice a la instancia si la transacción culminó o dio vuelta atrás.

- public void **beforeCompletion**();

  Este método notifica a una instancia de componente de sesión que una transacción está por finalizar.

Ejemplo:

```
package sb;
import java.rmi.RemoteException;
import javax.ejb.EJBException;
import javax.ejb.EJBObject;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.SessionSynchronization;
import javax.naming.InitialContext;
import javax.naming.NamingException;

// En este ejemplo un componente de sesión,
// con estado y sincronizado.

public class OpBean implements SessionBean, SessionSynchronization {
   protected int total = 0;   // estado real del componente
   protected int newTotal = 0;   // valor durante Tx,
                                 // aún no finalizada
   protected String clientUser = null;
   protected SessionContext sessionContext = null;

   public void ejbCreate(String user) {
      total = 0;
      clientUser = user;
   }

   public void ejbActivate() {
      // No utilizado
   }

   public void ejbPassivate() {
      // No utilizado
   }

   public void ejbRemove() {
      // No utilizado
   }

   public void setSessionContext(SessionContext sessionContext) {
      this.sessionContext = sessionContext;
   }
```

```
        public void afterBegin() {
           newTotal = total;
        }

        public void beforeCompletion() {
           // Por ahora...nada
           // Podemos acceder al entorno del
           // componente en cualquier parte
           // por ejemplo aquí!
           try {
              InitialContext ictx = new InitialContext();
              String value = (String)ictx.lookup("java:comp/env/prop1");
              // value debería ser aquel definido en ejb-jar.xml
           } catch (NamingException e) {
             throw new EJBException(e);
           }
        }

        public void afterCompletion(boolean committed) {
           if (committed) {
              total = newTotal;
           }
        }

        public void buy(int s) {
           newTotal = newTotal + s;
        }

        public int read() {
           return newTotal;
        }
     }
```

3.5.2 Desarrollo de Componentes Entidad

Un componente entidad representa datos persistentes. Es la visión objeto de una entidad almacenada en una base de datos relacional. La persistencia de un componente entidad puede ser manejada de dos maneras:

- **Persistencia manejada por el contenedor**: la persistencia es manejada implícitamente por el contenedor, ningún código de acceso a datos va a ser proporcionado por el proveedor del componente. El estado del componente será almacenado en una base de datos relacional de acuerdo a la descripción de correspondencias entregada dentro del descriptor de publicación.
- **Persistencia manejada por el componente**: el proveedor del componente escribe las operaciones de acceso a la base de datos (código JDBC) en los métodos del componente empresarial que son especificados para operaciones de creación, carga, almacenamiento, recuperación y remoción de datos (`ejbCreate`, `ejbLoad`, `ejbStore`, `ejbFindXXX`, `ejbRemove`).

Actualmente, la plataforma maneja la persistencia en sistemas de almacenamiento relacional a través de la interfaz JDBC. Tanto para persistencia manejada por el contenedor como manejada por el componente, las conexiones JDBC son obtenidas de un objeto *DataSource* proporcionado en el nivel de servidor EJB. La interfaz **DataSource** es definida en las extensiones estándares JDBC 2.0. Un objeto *DataSource* identifica una base de datos y una manera de accederla vía JDBC (un controlador JDBC). Un servidor EJB pudiese tener acceso a varias bases de datos y entonces proporciona varios objetos *DataSource*.

3.5.2.1 La interfaz local

La interfaz local es usada por cualquier aplicación cliente para crear y recuperar instancias del componente entidad. El proveedor del componente solo necesita proporcionar la interfaz deseada, el contenedor proveerá automáticamente la implementación. Debe extender la interfaz **javax.ejb.EJBHome**. Los diferentes métodos de ésta interfaz deben seguir las reglas para Java RMI. Las firmas de los métodos `create` y `findXXX` deberían emparejar a las firmas de los métodos `ejbCreate` y `ejbFindXXX` que serán proporcionados luego en la clase de implementación del componente empresarial (igual número y tipo de argumentos, pero diferente tipo de retorno).

3.5.2.1.1 Métodos de creación

- El tipo de retorno es la interfaz remota del componente empresarial
- Las excepciones definidas en la cláusula ***throws*** deben incluir las excepciones definidas por los métodos `ejbCreate` y `ejbPostCreate` y deben incluir **javax.ejb.CreateException** y **java.rmi.RemoteException**.

3.5.2.1.2 Métodos de búsqueda

Estos métodos son utilizados para buscar un objeto EJB o una colección de objetos EJB. Los argumentos del método son usados por la implementación del componente empresarial para localizar los objetos entidad solicitados.

En el caso de persistencia manejada por el componente, el proveedor del componente es responsable de desarrollar los correspondientes métodos `ejbFindXXX` en la implementación de componente. En el caso de persistencia manejada por el contenedor, el

proveedor del componente no escribe estos métodos, ellos son generados al momento de publicación por las herramientas de la plataforma; la descripción del método es proporcionada en el descriptor de publicación, como se define en la sección *Configurando acceso a bases de datos para persistencia manejada por el contenedor*.

En la interfaz local, los métodos de búsqueda deben seguir las siguientes reglas:

- Deben ser llamados "`find<método>`" (Ej. `findTotalCuenta`)
- El tipo de retorno debe ser la interfaz remota del componente empresarial o una colección de ésta
- Las excepciones definidas en la cláusula ***throws*** deben incluir las excepciones definidas por los correspondientes métodos `ejbFindXXX` y deben incluir **javax.ejb.FinderException** y **java.rmi.RemoteException**.

Por lo menos uno de estos métodos es obligatorio, *findByPrimaryKey*, el cual toma como argumento un valor de clave primaria y retorna el correspondiente objeto EJB.

Ejemplo

El componente de ejemplo Cuenta, será usado para ilustrar estos conceptos. El estado de una instancia componente entidad es almacenado dentro de una base de datos relacional donde la siguiente tabla debería existir:

CREATE TABLE CUENTA (CUENTNO INTEGER PRIMARY KEY, CLIENTE VARCHAR(30), BALANCE NUMBER(15, 4));

```
public interface CuentaHome extends EJBHome {
   public Cuenta create(int cuentno, String cliente, double balance)
      throws RemoteException, CreateException;

   public Cuenta findByPrimaryKey(CuentaBeanPK pk)
      throws RemoteException, FinderException;

   public Cuenta findPorNumero (int cuentno)
      throws RemoteException, FinderException;

   public Enumeration findCuentasMayores(double val)
      throws RemoteException, FinderException;
}
```

3.5.2.2 La interfaz remota

La interfaz remota es la visión del cliente de una instancia de un componente entidad. Es lo que retorna al cliente la interfaz local después de crear o encontrar una instancia de componente entidad. Esta interfaz contiene la lógica del negocio del componente empresarial. La interfaz debe extender la interfaz **javax.ejb.EJBObject**.

Los métodos de ésta interfaz deben seguir las reglas de Java RMI. Por cada método definido en ésta interfaz remota, debe haber un método correspondiente en la clase de implementación del componente empresarial (mismo número y tipo de argumentos, mismo tipo de retorno, mismas excepciones menos **RemoteException**).

Ejemplo

```
public interface Cuenta extends EJBObject {
   public double getBalance() throws RemoteException;
   public void setBalance(double d) throws RemoteException;
   public String getCliente () throws RemoteException;
   public void setCliente (String c) throws RemoteException;
   public int getNumero () throws RemoteException;
}
```

3.5.2.3 La clase de clave primaria

La manera más simple de definir una clave primaria es usar una clase estándar Java como **java.lang.String**. Este debe ser el tipo de un campo en la clase componente. No es posible definirla como un campo primitivo. La otra forma es definiendo una clase propia, como se indica más adelante.

La clase clave primaria es necesaria solamente para componentes entidad. Encapsula los campos que representan la clave primaria de un componente entidad en un solo objeto. La clase debe ser serializable y debe proporcionar una implementación apropiada para los métodos `hashcode()` y `equals(Object)`.

Para *persistencia manejada por el contenedor*, las siguientes reglas deben cumplirse:

- Los campos de la clase clave primaria deben ser declarados como públicos
- La clase clave primaria debe tener un constructor por omisión público
- Los nombres de los campos en la clase clave primaria deben ser un subconjunto de los nombres de los campos manejados por el contenedor del componente empresarial.

Ejemplo

```
public class CuentaBeanPK implements java.io.Serializable {
   public int cuentno;

   public CuentaBeanPK() {}
   public CuentaBeanPK(int cuentno) {this.cuentno = cuentno; }
   public int hashcode() {return cuentno; }
   public boolean equals(Object other) {
      // La implementación adecuada
   }
}
```

3.5.2.4 La clase del componente empresarial

La clase de implementación EJB implementa la lógica del negocio del componente indicada por la interfaz remota y los métodos requeridos para el entorno EJB, la interfaz de los cuales se define explícitamente en la especificación EJB. La clase debe implementar la interfaz **javax.ejb.EntityBean**, debe ser definida como pública y no debería ser abstracta.

El primer conjunto de métodos son aquellos que corresponden a los métodos de creación y búsqueda en la interfaz local:

- public PrimaryKeyClass **ejbCreate**(...)

  Este método es invocado por el contenedor cuando un cliente invoca la correspondiente operación de creación sobre la interfaz local del componente empresarial. El método debería iniciar las variables de la instancia a partir de los argumentos de entrada. El objeto retornado debería ser la clave primaria de la instancia creada. En el caso de persistencia manejada por el componente, el proveedor del componente debería desarrollar aquí el código JDBC para crear los datos correspondientes en la base de datos. En el caso de persistencia manejada por el contenedor, el contenedor ejecutará la inserción en la base de datos **luego** que el método *ejbCreate* termina y el valor retornado debería ser una clave primaria nula.

- public void **ejbPostCreate**(...);

  Existe un método *ejbPostCreate* (con el mismo número de parámetros) que corresponde a cada método *ejbCreate*. El contenedor invoca éste método luego de la ejecución del método *ejbCreate* correspondiente. Durante el método *ejbPostCreate*, la identidad del objeto está disponible.

- public <Clase Clave Primaria o Colección> **ejbFindXXX**(...);

  En persistencia manejada por el componente, el contenedor invoca éste método sobre una instancia de componente que no está asociada con ninguna identidad objeto en particular (algún tipo de método de clase) cuando el cliente invoca el método correspondiente en la interfaz local.

  La implementación utiliza los argumentos para localizar el o los objetos requeridos en la base de datos y retorna una clave primaria (o una colección de ellos). Al momento, las colecciones se representarán con objetos **java**.**util**.**Enumeration**.

  El método obligatorio *ejbFindByPrimaryKey* toma como argumento un valor de tipo clave primaria y retorna un objeto clave primaria (verifica que la correspondiente entidad exista en la base de datos).

  En el caso de *persistencia manejada por el contenedor*, el proveedor del contenedor no tiene que escribir estos métodos de búsqueda, ellos son generados al momento de publicación por las herramientas de la plataforma EJB. La información necesitada por la plataforma EJB para la generación automática de estos métodos de búsqueda debería ser proporcionada por el programador del componente; la especificación EJB no indica el formato de ésta descripción de métodos de búsqueda.

Luego, deben ser implementados los métodos de la interfaz **javax.ejb.EntityBean**:

- public void **setEntityContext**(EntityContext ic);

  Usado por el contenedor para pasar una referencia del contexto hacia la instancia del componente. El contenedor invoca éste método sobre una instancia después de que ella ha sido creada. Generalmente éste método es usado para almacenar la referencia en una variable de la instancia.

- public void **unsetEntityContext**();

  Elimina el contexto asociado. El contenedor llama éste método antes de remover la instancia. Este es el último método que el contenedor invoca sobre la instancia.

- public void **ejbActivate**();

  El contenedor invoca éste método cuando la instancia es retirada del fondo común de instancias disponibles para ser asociada con un objeto EJB específico. Este método coloca a la instancia en el estado de listo.

- public void **ejbPassivate**();

  El contenedor invoca éste método sobre una instancia antes de que ella se desasocie de un objeto EJB específico. Luego de concluido éste método, el contenedor pondrá la instancia dentro del fondo común de instancias disponibles.

- public void **ejbRemove**();

  Este método es invocado por el contenedor cuando un cliente invoca una operación de remoción sobre el componente empresarial. Para componentes entidad con *persistencia manejada por el componente*, éste método debería contener el código JDBC para remover los datos correspondientes en la base de datos. En el caso de *persistencia manejada por el contenedor*, éste método es llamado **antes** de que el contenedor remueva la entidad de la base de datos.

- public void **ejbLoad**();

  El contenedor invoca éste método para instruir a la instancia sincronizar su estado cargándolo desde la base de datos subyacente. En el caso de *persistencia manejada por el componente*, el proveedor EJB debería codificar en éste sitio las instrucciones JDBC para leer los datos desde la base de datos. Para *persistencia manejada por el contenedor*, la carga de datos desde la base de datos se hará automáticamente por el contenedor justo antes de llamar a *ejbLoad* y éste método solo debería contener "*instrucciones de cálculo posteriores a la carga*".

- public void **ejbStore**();

  El contenedor invoca éste método para instruir a la instancia sincronizar su estado almacenándolo en la base de datos subyacente. En el caso de *persistencia manejada por el componente*, el proveedor EJB debería codificar en éste sitio las instrucciones JDBC para escribir los datos en la base de datos. Para *persistencia manejada por el contenedor*, éste método debería contener únicamente algunas "*instrucciones de prealmacenaje*", puesto que el contenedor extraerá los campos manejados por el

componente y los escribirá en la base de datos justo **después** de la llamada a éste método.

Ejemplo

Este es un ejemplo para persistencia manejada por el contenedor.

```
package eb;

import java.rmi.RemoteException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.ObjectNotFoundException;
import javax.ejb.RemoveException;
import javax.ejb.EJBExeption;


public class CuentaImplBean implements EntityBean {
   // Para mantener la referencia al contexto
   protected EntityContext entityContext;

   // Estado del objeto
   public int cuentno;
   public String cliente;
   public double balance;

   public CuentaBeanPK ejbCreate(int valCuentno, String valCliente,
double valBalance) {
      // estado inicial
      cuentno = valCuentno;
      cliente = valCliente;
      balance = valBalance;

      return null;
   }

   public void ejbPostCreate(int valCuentno, String valCliente,
double valBalance) {
      // Nada en este ejemplo sencillo
   }

   public void ejbActivate() {
      // Nada en este ejemplo sencillo
   }

   public void ejbPassivate() {
      // Nada en este ejemplo sencillo
   }

   public void ejbLoad() {
      // Nada en este ejemplo de persistencia implícita
   }

   public void ejRemove() {
      // Nada en este ejemplo de persistencia implícita
```

```
    }

    public void ejbStore() {
       // Nada en este ejemplo persistencia implícita
    }

    public void setEntityContext(EntityContext ctx) {
       // Almacenar el contexto
       entityContext = ctx;
    }

    public void unSetEntityContext() {
       entityContext = null;
    }


    public double getBalance() {
       return balance;
    }

    public void setBalance(double d) {
       balance = balance + d;
    }

    public String getCliente() {
       return cliente;
    }

    public void setCliente (String c) {
       cliente = c;
    }

    public int getNumero() {
       return cuentno;
    }
}
```

3.5.2.5 Escritura de operaciones de acceso a bases de datos para persistencia manejada por el componente

En éste las operaciones de acceso a los datos son desarrolladas por el proveedor del componente usando la interfaz JDBC. Sin embargo, la gestión de conexiones a bases de datos debería hacerse a través de la interfaz **javax.sql.DataSource** proporcionada por la plataforma EJB. Esto es obligatorio puesto que la plataforma EJB es responsable de administrar el fondo común de conexiones y las transacciones. Así que, a fin de conseguir una conexión JDBC, en cada método que ejecute operaciones con base de datos, el proveedor del componente debería:

- Llamar al método *getConnection*(...) de la clase **DataSource**, con el fin de obtener una conexión para ejecutar operaciones JDBC en el contexto de la transacción actual (si hay alguna) y,
- Llamar al método *close*() sobre ésta conexión después de las operaciones con base de datos, con el propósito de devolver la conexión al fondo común (y desligarla de la transacción actual).

Por ejemplo:

```
public void haceAlgoEnBD(...) {
   conn = dataSource.getConnection();
   //... las operaciones con la base de datos…
   conn.close();
}
```

Un objeto *DataSource* asocia un controlador JDBC con una base de datos. Generalmente es registrado en JNDI por el servidor EJB al momento de arranque.

Un objeto *DataSource* es una fábrica de conexiones que gerencia recursos para objetos **java.sql.Connection** e implementa conexiones a un sistema manejador de base de datos. El código del componente empresarial refiere a fábricas de recursos usando nombres lógicos llamados "*Referencias a fábrica de conexiones que gerencia recursos*". Las referencias a fábrica de conexiones que gerencia recursos son entradas especiales en el entorno del componente empresarial. El proveedor del componente debería usar referencias a fábrica de conexiones que gerencia recursos para obtener objetos fuente de datos como sigue:

- Declarar la referencia al recurso en el descriptor estándar de publicación usando un elemento `resource-ref`.
- Buscar la fuente de datos en el entorno del componente empresarial usando la interfaz JNDI.

El publicador lía las referencias a fábrica de conexiones que gerencia recursos a las fábricas de recursos reales que están configuradas en el servidor. Esta parte de la configuración no se indica en la Especificación EJB, así que cada proveedor de servidor EJB implementa un mecanismo diferente.

Ejemplo

La declaración de la referencia a recurso en el descriptor estándar luce así:

```
<resource-ref>
   <res-ref-name>jdbc/CuentaExplDs</res-ref-name>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>Container</res-auth>
</resource-ref>
```

El método *ejbStore* del mismo ejemplo **Cuenta** con persistencia manejada por el componente se muestra debajo. Ejecuta operaciones JDBC para actualizar los registros de la base de datos que representan el estado de la instancia del componente entidad. La conexión JDBC es obtenida de la fuente de datos asociada con el componente. Esta fuente de datos ha sido creada por el servidor EJB y está disponible al componente a través de su nombre de referencia a recurso, el cual está definido en el descriptor estándar de publicación.

En alguna parte del componente, una referencia a un objeto fuente de datos del servidor EJB es iniciada:

```
it = new InitialContext();
ds = (DataSource)it.lookup("java:comp/env/jdbc/CuentaExplDs");
```

Luego, éste objeto fuente de datos es usado en la implementación de los métodos que realizan operaciones JDBC, como *ejbStore*, que se muestra aquí:

```
public void ejbStore() {
   try {  // consiga una conexión
      conn = ds.getConnection();
      // almacena el estado del objeto
      PreparedStatement st = conn.prepareStatement(
       "update Cuenta set cliente=?,balance=? where cuentno=?");
      st.setString(1, cliente);
      st.setDouble(2, balance);
      CuentaBeanPK pk = (CuentaBeanPK)entityContext.getPrimaryKey();
      st.setInt(3, pk.cuentno);
      st.executeUpdate();
      // cerrar
      st.close();
      conn.close();
   } catch (SQLException sqle) {
      throw new javax.ejb.EJBException(
       "Falla al almacenar en la base de datos", sqle);
   }
}
```

Nótese lo importante que sería la instrucción *close* si el servidor es usado intensamente por muchos clientes que utilizan los componentes. Puesto que *st* solo tiene alcance de método, será eliminada al final de éste (y se cerrará implícitamente), sin embargo, puede pasar algún tiempo hasta que el Recolector de Basura de Java elimine el objeto, por lo tanto, si el número de clientes realizando acceso al componente empresarial es importante, el DBMS podría lanzar una excepción “demasiados cursores abiertos” (una instrucción JDBC corresponde a un cursor DBMS). Dado que el fondo común de conexiones es mantenido por la plataforma, cerrar la conexión no afecta a la conexión física y por lo mismo los cursores abiertos no serán cerrados. Así que es preferible cerrar explícitamente la instrucción en el método.

Pudiese ser una buena regla de programación colocar las operaciones de cierre de la instrucción JDBC y la conexión JDBC en el bloque ***finally*** de una sentencia ***try***.

3.5.2.6 Configuración de acceso a bases de datos para persistencia manejada por el contenedor

Primero que todo, la manera estándar de indicar a una plataforma EJB que un componente entidad tiene persistencia manejada por el contenedor es dentro del descriptor de publicación. Con la marca `persistence-type` puesta a *Container* indicamos que es un componente CMP y luego se usa la marca `cmp-field` para indicar cuales son los campos que debe manejar el contenedor.

Con persistencia manejada por el contenedor, el programador no tiene que desarrollar el código para acceder a los datos en la base de datos relacional; éste código está incluido en el propio contenedor (generado por las herramientas de la plataforma). Sin embargo, a fin de que la plataforma EJB sepa como acceder a la base de datos y que datos leer y escribir en ésta, dos tipos de información deben ser entregados con el componente:

- Primero el contenedor debería saber a que base de datos acceder y como accederla. Para lograr tal propósito, la única información requerida es el nombre de la fuente de datos que será usada para obtener las conexiones JDBC. Para persistencia manejada por el contenedor, debería ser usada una y solo una fuente de datos por componente.
- Luego, es necesario conocer la correspondencia entre los campos del componente y la base de datos subyacente (qué tabla con qué columna).

La especificación EJB no indica cómo está información debería ser proporcionada a la plataforma EJB por el Proveedor y por el Publicador de componentes. Por lo tanto, cada Proveedor de Servidor EJB puede implementar un mecanismo propio (Generalmente basado en una descripción que emplea XML).

El Proveedor del Componente es responsable por definir la correspondencia entre los campos del componente y las columnas de las tablas en la base de datos. El nombre de la fuente de datos puede ser fijado al momento de publicación ya que depende de la configuración de la plataforma EJB.

En el caso de persistencia manejada por el componente, la correspondencia con la base de datos no existe y la propiedad del nombre de la fuente de datos debería ser parte del archivo de propiedades del componente.

3.5.3 Comportamiento en transacciones

3.5.3.1 Administración declarativa de la transacción

En el caso de que la administración de transacciones sea realizada por el contenedor, el comportamiento de un componente empresarial en la transacción está definido en tiempo de configuración y es parte del elemento `assembly-descriptor` del descriptor de publicación estándar. Es posible definir un comportamiento común para todos los métodos del componente o definirlo individualmente. Esto se logra especificando un atributo de transacciones, el cual puede ser uno de los siguientes:

- **NotSupported**: Si el método es llamado dentro de una transacción, ésta transacción es suspendida durante el tiempo de ejecución del método.
- **Required**: Si el método es llamado dentro de una transacción, el método es ejecutado dentro del alcance de ésta transacción, de lo contrario, una nueva transacción es iniciada para la ejecución del método y terminada antes de que el resultado del método sea enviado de vuelta a quien llama.
- **RequiresNew**: El método siempre será ejecutado dentro del alcance de una nueva transacción. La nueva transacción es iniciada desde la ejecución del método y terminada antes de enviar el resultado a quien llama. Si el método es llamado dentro

del alcance de una transacción, ésta es suspendida antes de que la nueva comience y es reactivada cuando la nueva se ha completado.

- **Mandatory**: El método siempre debería ser llamado dentro del alcance de una transacción, de lo contrario el contenedor lanzará la excepción **TransactionRequired**.
- **Supports**: El método es invocado dentro del alcance de la transacción de quien llama, si quien llama no tiene una transacción asociada, el método es invocado sin alcance de transacción.
- **Never**: El cliente está obligado a llamar al componente sin un contexto de transacción; si ese no es el caso el contenedor lanzará una excepción **java.rmi.RemoteException**.

La Tabla 3.1 muestra el efecto que cada atributo produce sobre las transacciones.

Tabla 3.1: Efecto de los atributos declarativos para transacciones

Atributo de Transacción	**Transacción del Cliente**	**Transacción asociada con el método del componente**
NotSupported	- T1	- -
Required	- T1	T2 T1
RequiresNew	- T1	T2 T2
Mandatory	- T1	Error T1
Supports	- T1	- T1
Never	-	-

Atributo de Transacción	Transacción del Cliente	Transacción asociada con el método del componente
	T1	Error

Ejemplo

En el descriptor de publicación, la especificación de los atributos de transacción aparece así:

```
<assembly-descriptor>
   <container-transaction>
      <method>
         <ejb-name>CuentaImpl</ejb-name>
         <method-name>*</method-name>
      </method>
      <trans-attribute>Supports</trans-attribute>
   </container-transaction>
   <container-transaction>
      <method>
         <ejb-name>CuentaImpl</ejb-name>
         <method-name>getBalance</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
   </container-transaction>
   <container-transaction>
      <method>
         <ejb-name>CuentaImpl</ejb-name>
         <method-name>setBalance</method-name>
      </method>
      <trans-attribute>Mandatory</trans-attribute>
   </container-transaction>
</assembly-descriptor>
```

En este caso, el atributo de transacción por omisión es **Supports** (definido al nivel de componente) para todos los métodos para los cuales no se ha especificado explícitamente una marca `container-transaction`, y para los métodos *getBalance* y *setBalance* sus respectivos atributos de transacción son **Required** y **Mandatory** (definidos al nivel de método).

Un componente de sesión que administre él mismo sus transacciones debe fijar en su descriptor de publicación estándar `transaction-type` a `Bean`. Para demarcar los límites de la transacción en un componente de sesión con transacciones manejadas por el componente, el programador del componente debería usar la interfaz **javax.transaction.UserTransaction**, definida sobre un objeto servidor EJB que puede ser

obtenido usando el método *EJBContext.getUserTransaction()* (el objeto **SessionContext** o el objeto **EntityContext** dependiendo si el método está definido en un componente de sesión o uno entidad).

El ejemplo que sigue muestra un método de componente de sesión *doTxJob* demarcando los límites de la transacción; el objeto **UserTransaction** es obtenido del objeto **SessionContext**, que debería haber sido iniciado en el método *setSessionContext* (véase el ejemplo de componente de sesión).

```
public void doTxJob() throws RemoteException {
   UserTransaction ut = sessionContext.getUserTransaction();
   ut.begin();
   //...las operaciones de la transacción...
   ut.commit();
}
```

3.5.3.2 Administración de transacciones distribuidas

Como se explicó en la sección previa, el comportamiento en transacciones de una aplicación puede ser definido en una forma declarativa o codificado dentro del propio componente y / o cliente (demarcación de límites de transacción). En cualquier caso, los aspectos distribuidos de las transacciones son completamente transparentes al proveedor del componente y al ensamblador de la aplicación. Esto significa que la transacción podría involucrar a componentes localizados en varios servidores EJB y que la plataforma tomará a cargo la administración de la transacción global. Realizará el protocolo de conclusión de dos fases entre los diferentes servidores y el programador del componente no tendrá nada que hacer para éste fin.

Una vez que los componentes han sido desarrollados y la aplicación ha sido ensamblada, es posible para el Publicador y el Administrador configurar la distribución de los diferentes componentes en una o varias máquinas y dentro de uno o varios servidores EJB. Esto se puede hacer sin perjudicar ni el código de los componentes ni los descriptores de publicación. La configuración distribuida es especificada en tiempo de arranque: dentro de las propiedades del entorno del servidor EJB, se podría especificar:

- cuales componentes empresariales manejará el servidor EJB y
- si un Vigilante de Transacciones Java (JTM) será colocado en la misma JVM o no.

El Vigilante de Transacciones Java podría ejecutarse fuera de cualquier servidor EJB; o sea, podría ser lanzado de manera independiente.

La Figura 3.5 ilustra cuatro casos de configuración de distribución para tres componentes.

- **Caso 1**: Los tres componentes B1, B2 y B3 están localizados en el mismo servidor EJB, el cual incluye un Vigilante de Transacciones Java.
- **Caso 2**: Los tres componentes están colocados en diferentes servidores EJB, uno de los cuales ejecuta el JTM, quien maneja la transacción global.
- **Caso 3**: Los tres componentes están colocados en diferentes servidores EJB, el JTM está ejecutándose fuera de cualquier servidor EJB.
- **Caso 4**: Los tres componentes están colocados en diferentes servidores EJB. Cada servidor EJB está ejecutando un JTM. Uno de los JTM actúa como vigilante maestro, mientras los otros dos están subordinados.

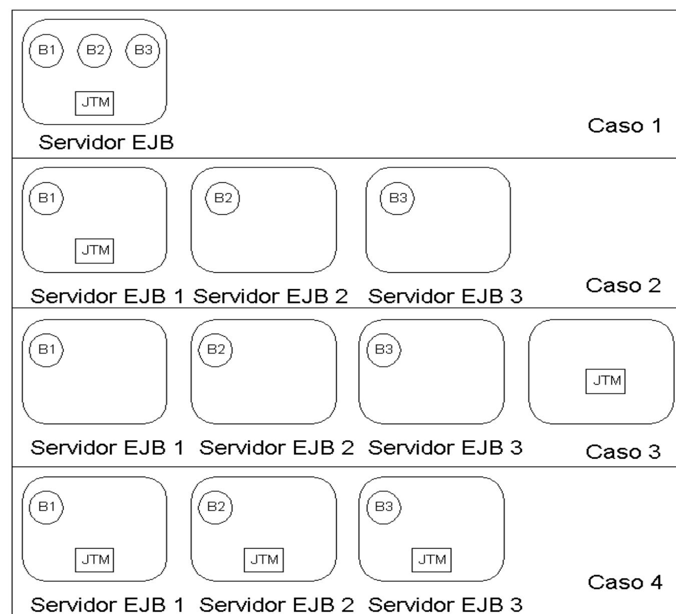

Figura 3.5: Los cuatro casos para disponer del Vigilante de Transacciones

Estas diferentes configuraciones pudiesen obtenerse arrancando los servidores EJB y eventualmente el JTM (Caso 3) con las propiedades adecuadas. La razón para optar por uno u otro caso es la disposición de los recursos y el balance de la carga. Sin embargo, los siguientes consejos debiesen notarse:

- Si los componentes debieran ejecutarse en la misma máquina, con la misma configuración de servidor EJB, el Caso 1 es ciertamente el más apropiado.
- Si los componentes debieran ejecutarse sobre máquinas diferentes, el Caso 4 es el más apropiado, puesto que favorece el tratamiento local de transacciones.
- Si los componentes debieran ejecutarse en la misma máquina, pero necesitan diferentes configuraciones para el servidor EJB, el Caso 2 es un enfoque apropiado.

3.5.4 El Entorno del Componente Empresarial

El entorno del componente empresarial es un mecanismo para permitir la personalización de la lógica del negocio del componente empresarial durante el ensamblaje o la publicación. El entorno del componente empresarial le permite a éste ser ajustado sin la necesidad de acceder o cambiar su código fuente.

El entorno del componente empresarial es proporcionado por el contenedor al componente a través de la interfaz JNDI como un contexto JNDI. El código del componente llega al entorno empleando JNDI con un nombre que comienzan con **java:comp/env/**.

El Proveedor del Componente declara en el descriptor de publicación todo el entorno del componente a través de la marca `env-entry`. El Publicador puede establecer o modificar los valores de las entradas del entorno.

Ejemplo

```
InitialContext ictx = new InitialContext();
Context myEnv = ictx.lookup("java:comp/env");
Integer min = (Integer) myEnv.lookup("minvalue");
Integer max = (Integer) myEnv.lookup("maxvalue");
```

En el descriptor de publicación estándar, la declaración de estás variables es:

```
<env-entry>
   <env-entry-name>minvalue</env-entry-name>
   <env-entry-type>java.lang.Integer</env-entry-type>
   <env-entry-value>12</env-entry-value>
</env-entry>
<env-entry>
   <env-entry-name>maxvalue</env-entry-name>
   <env-entry-type>java.lang.Integer</env-entry-type>
   <env-entry-value>120</env-entry-value>
</env-entry>
```

3.5.5 Definición del descriptor de publicación

El Proveedor del Componente es también responsable de proporcionar el descriptor de publicación asociado a los componentes empresariales desarrollados. Mientras se va recorriendo los diferentes pasos del desarrollo de la aplicación y el ciclo de vida de implementación de los componentes empresariales, el descriptor se va completando.

La responsabilidad del Proveedor de Componentes y del Ensamblador de Componentes es proporcionar descriptores de publicación XML que honren la DTD XML definida para la especificación EJB.

Para posibilitar la publicación del componente empresarial Java en el servidor EJB, información no definida en el descriptor estándar XML de publicación es necesaria, como la correspondencia entre el componente y la base de datos subyacente para componentes entidad con persistencia manejada por el contenedor, por ejemplo. Estos datos son especificados, en el paso de publicación, en otro descriptor XML de publicación el cual es específico del servidor EJB.

El descriptor de publicación estándar debería contener la siguiente información estructural para cada componente empresarial:

- el nombre del componente empresarial,
- la clase del componente empresarial,
- la interfaz local del componente empresarial,
- la interfaz remota del componente empresarial,
- el tipo del componente empresarial,
- indicaciones de reentrada para componente entidad,
- el tipo de administración de estado del componente de sesión,
- el tipo de demarcación de transacciones del componente de sesión,
- administración de persistencia del componente entidad,
- clase clave primaria del componente entidad,
- campos manejados por el contenedor,
- entradas del entorno,
- referencias a fábrica de conexiones que administra recursos,
- atributos de transacción.

Ejemplo de descriptor de sesión

```
<!DOCTYPE ejb-jar SYSTEM "......./xml/ejb-jar_1_1.dtd">
<ejb-jar>
   <description>Aquí va la descripción del componente</description>
   <enterprise-beans>
      <session>
         <description>Componente de Ejemplo UNO</description>
         <display-name>Componente ejemplo uno</display-name>
         <ejb-name>ExampleOne</ejb-name>
         <home>tests.Ex1Home</home>
         <remote>tests.Ex1</remote>
         <ejb-class>tests.Ex1Bean</ejb-class>
         <session-type>Stateful</session-type>
         <transaction-type>Container</transaction-type>
         <env-entry>
            <env-entry-name>name1</env-entry-name>
            <env-entry-type>java.lang.String</env-entry-type>
            <env-entry-value>value1</env-entry-value>
         </env-entry>
         <resource-ref>
            <res-ref-name>jdbc/mydb</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Application</res-auth>
         </resource-ref>
      </session>
   </enterprise-beans>
   <assembly-descriptor>
      <container-transaction>
         <method>
            <ejb-name>ExampleOne</ejb-name>
            <method-name>*</method-name>
         </method>
         <trans-attribute>Required</trans-attribute>
      </container-transaction>
      <container-transaction>
         <method>
            <ejb-name>ExampleOne</ejb-name>
            <method-intf>Home</method-intf>
            <method-name>*</method-name>
         </method>
         <trans-attribute>Supports</trans-attribute>
      </container-transaction>
      <container-transaction>
         <method>
            <ejb-name>ExampleOne</ejb-name>
            <method-name>methodOne</method-name>
         </method>
         <trans-attribute>NotSupported</trans-attribute>
      </container-transaction>
      <container-transaction>
         <method>
            <ejb-name>ExampleOne</ejb-name>
            <method-name>methodTwo</method-name>
            <method-params>
               <method-param>int</method-param>
            </method-params>
         </method>
         <trans-attribute>Mandatory</trans-attribute>
```

```
        </container-transaction>
        <container-transaction>
           <method>
              <ejb-name>ExampleOne</ejb-name>
              <method-name>methodTwo</method-name>
              <method-params>
                 <method-param>java.lang.String</method-param>
              </method-params>
           </method>
           <trans-attribute>NotSupported</trans-attribute>
        </container-transaction>
     </assembly-descriptor>
  </ejb-jar>
```

Ejemplo de descriptor para entidad con persistencia manejada por el contenedor

```
<!DOCTYPE ejb-jar SYSTEM ".......//xml/ejb-jar_1_1.dtd">
<ejb-jar>
   <description>Aqui va la descripcion del componente</description>
   <enterprise-beans>
      <entity>
         <description>Componente de ejemplo</description>
         <display-name>Bean example two</display-name>
         <ejb-name>ExampleTwo</ejb-name>
         <home>tests.Ex2Home</home>
         <remote>tests.Ex2</remote>
         <ejb-class>tests.Ex2Bean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>tests.Ex2PK</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-field>
            <field-name>field1</field-name>
         </cmp-field>
         <cmp-field>
            <field-name>field2</field-name>
         </cmp-field>
         <cmp-field>
            <field-name>field3</field-name>
         </cmp-field>
         <primkey-field>field3</primkey-field>
         <env-entry>
            <env-entry-name>name1</env-entry-name>
            <env-entry-type>java.lang.String</env-entry-type>
            <env-entry-value>value1</env-entry-value>
         </env-entry>
      </entity>
   </enterprise-beans>
   <assembly-descriptor>
      <container-transaction>
         <method>
            <ejb-name>ExampleTwo</ejb-name>
            <method-name>*</method-name>
         </method>
         <trans-attribute>Supports</trans-attribute>
      </container-transaction>
   </assembly-descriptor>
</ejb-jar>
```

3.5.6 Empacado del componente

Los componentes empresariales son empacados para publicación en un archivo ARCHIVE estándar de Java, llamado archivo **ejb-jar**. Este archivo debe contener:

- Los archivos de la clase componente

  Los archivos de clase para las interfaces remota y local, de la implementación del componente, de la clase de clave primaria del componente (si existe) y todas las clases necesarias.

- El descriptor de publicación del componente

El archivo **ejb-jar** debe contener los descriptores de publicación, formados por:

- El descriptor de publicación estándar XML, en el formato definido por la especificación EJB. Este descriptor de publicación debe ser almacenado con el nombre META-INF/ejb-jar.xml en el archivo **ejb-jar**.
- El descriptor de publicación específico del servidor EJB definido en su propio formato.

3.6 Ensamblaje de aplicaciones

El rol del Ensamblador de aplicaciones es desarrollar una aplicación completa utilizando algunos componentes empresariales existentes. Esto incluye el desarrollo de la aplicación cliente y el eventual desarrollo de componentes empresariales adicionales.

3.6.1 Ensamblaje de componentes en el servidor

El Ensamblador de aplicaciones trabaja desde la perspectiva cliente del componente empresarial (es decir, las interfaces local y remota) y no necesita tener ningún conocimiento sobre su implementación. Del lado del servidor, el ensamblador de aplicaciones podría definir nuevos componentes empresariales que hacen uso de los existentes. Generalmente, estos nuevos componentes son del tipo sesión y pueden ser requeridos por varias razones:

- Los componentes de sesión mantienen el contexto de la aplicación cliente del lado del servidor y a menudo son específicos de una aplicación cliente particular. Esto significa que, la mayoría del tiempo, no se los puede considerar como “componentes reutilizables de servidor” y que ellos deben ser desarrollados por el ensamblador de aplicaciones.

- Los componentes entidad representan datos almacenados en las bases de datos del servidor. Ellos son compartidos por los clientes. Puede suceder que una aplicación cliente no necesite manipular todos los datos retenidos por el componente entidad o que necesita manipular los datos extraídos de varios componentes entidad. En lugar de desarrollar la manipulación y recolección en la aplicación cliente, invocando métodos sobre la interfaz remota de los componentes entidad, podría ser más apropiado desarrollar un componente de sesión que haga éste trabajo en el servidor. Esto sería menos caro en términos de consumo de recursos en la red.

Puesto que tales componentes de sesión tienen un comportamiento de clientes para los componentes existentes. Ellos deben ser desarrollados como se describe en la sección *La Interfaz de Programación de Aplicaciones EJB*. Cómo desarrollar el código de los métodos que invocan los componentes existentes se explica en la sección *Desarrollo de aplicaciones cliente*.

3.6.2 Desarrollo de aplicaciones cliente

Dependiendo de la arquitectura de la aplicación, la parte cliente puede ser:

- una aplicación Java,
- un applet Java,
- un servlet Java,
- un componente empresarial.

Cada uno comunicándose con uno o varios servidores EJB vía RMI.

Una aplicación cliente invoca métodos sobre los objetos remoto y local EJB. Hay varias maneras en que una aplicación cliente consigue un objeto EJB:

1. El cliente tiene el nombre JNDI del objeto EJB. Este nombre es usado para conseguir el objeto EJB.
2. El cliente tiene el nombre JNDI del objeto local, éste es un caso más usual. Este nombre se utiliza para conseguir el objeto Local, luego un método de búsqueda es invocado sobre éste Local para obtener uno o varios objetos componente entidad. El cliente puede también invocar un método de "creación" sobre el objeto Local para crear un nuevo objeto EJB (de sesión o entidad).

3. El cliente posee un *handle* a un objeto EJB. Un *handle* es un objeto que identifica al objeto EJB; puede ser serializado, almacenado y usado posteriormente por el cliente para obtener una referencia al objeto EJB, usando el método *getEJBObject*().
4. El cliente tiene una referencia a un objeto EJB y algunos métodos definidos en la interfaz remota del componente empresarial retornan objetos EJB.

A continuación, las diferentes fases de una típica aplicación cliente serán ilustradas por el código correspondiente. El ejemplo se basa en el componente entidad tratado anteriormente.

Se consigue una referencia al objeto Local, teniendo su nombre JNDI ("CuentaHome1") y el contexto JNDI inicial:

```
CuentaHome home = (CuentaHome)
     PortableRemoteObject.narrow(
     initialContext.lookup("CuentaHome1"), CuentaHome.class);
```

Consigue una referencia al número de cuenta 102:

```
Cuenta a2 = home.findByNumber(102);
```

En éste objeto EJB, fijamos el atributo *balance* a 100:

```
a2.setBalance(100.0);
```

Crea un nuevo objeto EJB:

```
Cuenta a1 = home.create(109, "Juan Carlos", 0);
```

Para comenzar y terminar una transacción desde la aplicación cliente:

```
UserTransaction utx = (UserTransaction)
     PortableRemoteObject.narrow(
       initialContext.lookup("javax.transaction.UserTransaction"),
       UserTransaction.class
     );

utx.begin();
//...la transacción...
utx.commit();   // o utx.rollback();
```

Nota: La forma de obtener el objeto *utx*, implementando la interfaz **UserTransaction** en el servidor EJB, desde una aplicación cliente que no es un componente, no está determinada, ni en la Especificación EJB ni en la Especificación de la API de Transacciones Java. Por lo tanto el ejemplo mostrado puede no funcionar en ciertos servidores.

Si un componente de sesión intenta demarcar límites de transacción, la forma de hacerlo está especificada y descrita en la sección *Comportamiento en transacciones*.

3.7 Infraestructura de seguridad EJB

3.7.1 Modelo de seguridad declarativo EJB 1.1

El modelo de seguridad recomendado por la especificación EJB 1.1 es un modelo declarativo que evita la introducción de código de seguridad dentro de los métodos de la lógica de negocio EJB. La *Especificación para Componentes Empresariales Java versión 1.1, sección 15.1* dice:

La arquitectura EJB alienta al Proveedor de Componentes a implementar la clase del componente empresarial sin codificar las políticas y mecanismos de seguridad dentro de los métodos de la lógica del negocio. En la mayoría de los casos, los métodos de lógica del negocio del componente no deberían contener ninguna lógica relacionada con seguridad. Esto permite al Publicador configurar las políticas de seguridad para la aplicación en la manera más apropiada para el entorno operativo empresarial.

3.7.2 Configuración de la seguridad declarativa

El modelo de seguridad declarativa de EJB 1.1 se especifica en las marcas **security-role** y **method-permission** del descriptor de publicación.

El Ensamblador de Aplicaciones define los permisos de métodos requeridos para cada rol de seguridad. Un permiso de método es un permiso para invocar un grupo específico de métodos en las interfaces local y remota del componente. Un rol de seguridad es un grupo semántico de permisos de métodos. Un usuario debe tener al menos un rol de seguridad asociado con un método para poder invocarlo. Debido a que el ensamblador de la aplicación, en general, no conoce el entorno de seguridad del ambiente operativo, los roles de seguridad están pensados como roles lógicos, donde cada uno representa un tipo de usuario que debería tener los mismos derechos de acceso al grupo de métodos.

3.7.2.1 ¿Qué es un rol?

La Especificación EJB 1.1, sección 15.3 dice:

Es importante mantener la idea de que los roles de seguridad son usados para definir la visión lógica de la seguridad de una aplicación. Estos no deberían confundirse con grupos de usuarios, usuarios y otros conceptos que existan en el entorno operativo de la empresa específica.

Un rol de seguridad se define usando los siguientes elementos en el descriptor de publicación:

security-role Contiene la definición de un rol de seguridad

role-name Contiene el nombre del rol de seguridad

description Una descripción opcional sobre el rol de seguridad

El siguiente descriptor de publicación muestra un ejemplo de definición de roles de seguridad.

```
...
<assembly-descriptor>
<security-role>
<description>
Este rol incluye a los empleados de la empresa que tienen acceso a la
aplicación general de empleados. Solo se permite usar su propia
información
</description>
<role-name>empleado</role-name>
</security-role>
<security-role>
<description>
Este rol incluye a los empleados del departamento de recursos humanos. El
rol está permitido de mirar y actualizar todos los datos de empleados.
</description>
<role-name>hr-departamento</role-name>
</security-role>
<security-role>
<description>
Este rol incluye a los empleados de financiero. Se le permite mirar y
actualizar las entradas en la nómina de cualquier empleado.
</description>
<role-name>nómina-departamento</role-name>
</security-role>
...
</assembly-descriptor>
```

De la *Especificación EJB 1.1, sección 15.3.2*. Si el ensamblador de la aplicación ha definido roles de seguridad para los componentes empresariales en el archivo **ejb-jar.xml**, estos también pueden especificar los métodos de las interfaces local y remota que cada rol

de seguridad está permitido de invocar. El ensamblador define la relación de permisos de métodos en el descriptor de publicación usando marcas **method-permission** como sigue:

- Cada marca **method-permission** incluye una lista de uno o más roles de seguridad y una lista de uno o más métodos. Todos los roles de seguridad están permitidos de invocar todos los métodos listados.
- Cada rol de seguridad en la lista es identificado por la marca **role-name** y cada método (o conjunto de métodos, como se describe debajo) es identificado por la marca **method**. Una descripción opcional puede ser asociada con una marca **method-permission** usando la marca **description**.
- La relación de permisos para métodos está definida por la unión de todos los permisos de método definidos en las marcas individuales **method-permission**.
- Un rol de seguridad o un método pudiese aparecer en varias marcas **method-permission**. Es posible que algunos métodos no sean asignados a ningún rol de seguridad. Esto significa que ninguno de los roles de seguridad definidos por el Ensamblador de la Aplicación necesita acceso a estos métodos.

Hay tres estilos legales para componer la marca **method**:

Estilo 1.- Es utilizado para referir a todos los métodos de las interfaces local y remota del componente especificado.

```
<method>
<ejb-name>EJBNAME</ejb-name>
<method-name>*</method-name>
</method>
```

Estilo 2.- Es utilizado para referir a un método específico de las interfaces remota o local. Si existe varios métodos con el mismo nombre, este estilo se refiere a todos ellos.

```
<method>
<ejb-name>EJBNAME</ejb-name>
<method-name>METHOD</method-name>
</method>
```

Estilo 3.- Es utilizado para referir un método específico dentro del conjunto de métodos con el mismo nombre. El método debe estar definido en la interfaz remota o local del componente. La marca opcional **method-intf** puede ser usada para diferenciar métodos con el mismo nombre y firma definidos en las interfaces remota y local.

```
<method>
<ejb-name>EJBNAME</ejb-name>
```

```
<method-name>METHOD</method-name>
<method-params>
<method-param>PARAMETER_1</method-param>
...
<method-param>PARAMETER_N</method-param>
</method-params>
</method>
```

El siguiente ejemplo (para EJB 1.1) ilustra como los roles de seguridad se asigna a permisos en el descriptor de publicación.

```
...
<method-permission>
<role-name>empleado</role-name>
<method>
<ejb-name>ServicioEmpleados</ejb-name>
<method-name>*</method-name>
</method>
</method-permission>

<method-permission>
<role-name>empleado</role-name>
<method>
<ejb-name>Nómina</ejb-name>
<method-name>findByPrimaryKey</method-name>
</method>

<method>
<ejb-name>Nómina</ejb-name>
<method-name>getInfoEmpleado</method-name>
</method>

<method>
<ejb-name>Nómina</ejb-name>
<method-name>actualizaInfoEmpleado</method-name>
</method>
</method-permission>

<method-permission>
<role-name>Administarción</role-name>
<method>
<ejb-name>AdimServicioEmpleado</ejb-name>
<method-name>*</method-name>
</method>
</method-permission>
...
```

3.7.3 Mejora de la seguridad EJB

Aunque es un buen concepto, en general el modelo de seguridad declarativa es a menudo muy simplista como para cubrir los requerimientos de aplicaciones empresariales. Las razones incluyen:

- Los permisos para método pueden ser una función de los argumentos o del estado del componente.

- Los permisos para método pueden ser una función del usuario que llama o de algún estado de la aplicación.
- Los roles del usuario que llama pudiesen ser asignados después de la publicación y por lo tanto no disponibles para indicarse en el descriptor de publicación.

La especificación EJB 1.1 define un mecanismo por el cual el proveedor del componente puede mejorar la seguridad. Esto requiere la introducción de lógica de seguridad dentro de los métodos de implementación de la lógica de negocio del componente. Por ésta razón, la *Especificación EJB 1.1, Sección 15.2.5* dice:

Nota: En general la administración de la seguridad debería ser ejercida por el Contenedor en una manera que sea transparente a los métodos de lógica del negocio del componente. La API de seguridad descrita en ésta sección debería ser utilizada en situaciones poco frecuentes en las cuales los métodos de la lógica de negocio del componente necesitan acceder a información del contexto de seguridad.

La API EJB 1.1 para personalizar la seguridad consiste de los siguientes métodos de la interfaz **javax.ejb.EJBContext**:

- `java.security.Principal` **`getCallerPrincipal`** ()
- `boolean` **`isCallerInRole`**(`java.lang.String roleName`)

Usando estos métodos junto con código extra se puede añadir cualquier tipo de validación sobre seguridad al componente empresarial. Cuando se utiliza *isCallerInRole* debe existir una marca **security-role-ref** en el descriptor de publicación para todos los nombres de roles de seguridad empleados por el código del componente. Declarar referencias a los roles de seguridad en el código permite al Ensamblador de Aplicaciones o Publicador enlazar los nombres de los roles de seguridad usados en el código con los roles de seguridad definidos por una aplicación ensamblada a través de las marcas **security-role**. Una marca **role-link** debe ser utilizada aun si el valor de **role-name** es usado como el valor de referencia **role-link**. El siguiente descriptor muestra como enlazar la referencia a rol de seguridad llamada *nómina* al rol de seguridad llamado *nómina-departamento*.

```
..
<enterprise-beans>
...
<entity>
<ejb-name>Nómina</ejb-name>
```

```
<ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
...
<security-role-ref>
<description>
Este rol debería ser asignado a los empleados del departamento
financiero. Los miembros de éste rol tienen acceso a la información de
nómina de cualquiera. El rol ha sido enlazado con el rol del departamento
financiero.
</description>
<role-name>nómina</role-name>
<role-link>nómina-departamento</role-link>
</security-role-ref>
...
</entity>
...
</enterprise-beans>
...
```

Un problema con éste enfoque es que la seguridad tiende a ser independiente de la lógica de negocio de la aplicación y es más una función del entorno de publicación.

Capítulo 4

Entorno Linux para Java Empresarial

Linux es un sistema operativo semejante a Unix que actualmente se ejecuta sobre Intel x86 y otras arquitecturas como Alpha o Sparc. Viene apoyado por una enorme cantidad de software disponible bajo diferentes licencias, aunque la mayoría está sujeta a la ***Licencia Pública General (GPL)*** de GNU.

Las empresas instalan Linux en sus redes de computadoras, usan el sistema operativo como base de la administración de datos financieros, hospitalarios y comerciales. Además se utiliza en entornos distribuidos y en telecomunicaciones. Las universidades alrededor del mundo utilizan Linux para enseñar en cursos sobre programación y diseño de sistemas operativos. Y, por supuesto, una enorme cantidad de entusiastas lo utilizan en sus casas para programar y en tareas de productividad.

Lo que hace diferente a Linux es que fue y continua siendo desarrollado por un grupo de voluntarios, principalmente en la Internet, intercambiando código, reportando errores y corrigiendo problemas dentro de un ambiente completamente abierto. Cualquiera puede unirse al esfuerzo de desarrollar Linux.

4.1 Breve historia de Linux

Unix es uno de los sistemas operativos más populares en el mundo debido a sus fortalezas. Fue desarrollado originalmente como un sistema multitarea para minicomputadoras y mainframes en la mitad de los setentas y desde entonces ha crecido hasta ser uno de los sistemas operativos más ampliamente usados.

Linux es una versión de Unix de libre distribución desarrollada en principio por Linus Torvalds en la Universidad de Helsinki. Linux fue desarrollado con la ayuda de muchos programadores y expertos Unix en la Internet, permitiendo a cualquiera, con conocimientos suficientes, desarrollar y cambiar el sistema. El ***kernel*** de Linux no utiliza código propietario de AT&T u otra empresa. Mucho del software disponible para Linux se desarrolla dentro del proyecto GNU en la Fundación del Software Libre en Cambridge. Sin

embargo, programadores alrededor del mundo han contribuido a la creciente colección de software para Linux.

Linux se desarrolló originalmente por entretenimiento. Torvalds se inspiró en ***Minix***, un pequeño sistema Unix creado por Andy Tanenbaum y la primera discusión sobre Linux fue en el grupo de noticias USENET **comp.os.minix**. Estas discusiones se centraron en el desarrollo de un pequeño sistema Unix con fines académicos para los usuarios de Minix que deseaban más.

El 5 de octubre de 1991, Torvalds anuncia la primera versión "oficial" de Linux, la versión 0.02. En ese momento era posible ejecutar ***bash*** (el shell Bourne Again de GNU) y ***gcc*** (El compilador C de GNU) y casi nada más. El objetivo primario era desarrollar el kernel (nada de soporte de usuarios, documentación, distribución, etc.) Hoy mismo, la comunidad Linux da preferencia a la programación (el desarrollo del kernel) dejando en segundo plano otros aspectos.

4.2 Características de Linux

Linux tiene la mayoría de características que se encuentran en otras implementaciones de Unix más algunas otras que no se encuentran en ningún lado. A continuación anotamos algunas de las más relevantes:

- Es un sistema operativo completamente multitarea y multiusuario.
- Es compatible con cierto número de estándares Unix incluyendo IEEE POSIX 1, System V y BSD. Puesto que se desarrolló con portabilidad de código en mente, el software Unix libre se puede compilar sobre Linux.
- Todo el código fuente del sistema Linux incluyendo el kernel, controladores de dispositivos, bibliotecas, programas de usuario y herramientas de desarrollo se distribuyen libremente bajo alguna versión de la licencia GPL de GNU.
- Apoya diversos tipos de sistemas de archivos. El sistema de archivos nativo es ***ext2fs*** y se pueden utilizar unidades con sistemas de archivos FAT, VFAT, CD-ROM ISO 9660.
- Proporciona una implementación completa de redes TCP/IP. Esto incluye controladores de dispositivo para tarjetas Ethernet, SLIP, PLIP, PPP, NFS, etc. Todos los servicios y clientes TCP/IP como FTP, TELNET, SMTP, etc.

4.3 Linux como plataforma para Java

La relación de Java y Linux se genera de una combinación de peticiones del mercado, las ventajas percibidas de Java en Linux y la sensación generalizada de que las aplicaciones internet, especialmente las de comercio electrónico, van a ser escritas en Java y se ejecutaran sobre Linux.

Hasta aquellos que no le ven futuro a Java dentro del escritorio[6] creen que jugará un importante papel en los servidores. Linux es la plataforma escogida para comercio electrónico y el comercio electrónico camina hacia Java por la necesidad de estar basado en componentes.

La ventaja multiplataforma de Java también desempeña un papel importante en su uso creciente para desarrollar software Linux. Si un programador escribe una aplicación Java sobre Linux puede llevarla a otras plataformas sin modificaciones.

El hecho de que las aplicaciones Java sean portátiles beneficia también a Linux, puesto que los programadores de otros sistemas operativos pueden escribir sus programas y traerlos a Linux sin preocupación alguna sobre incompatibilidades.

Quizá en un principio no era muy práctico escribir aplicaciones con Java sobre Linux, mas ahora, las dos tecnologías están disfrutando de amplia aceptación en la industria y un desarrollador no se puede quejar por la carencia de buenas herramientas.

Hoy están disponibles máquinas virtuales para Linux, entornos de desarrollo integrado, sistemas de control de versión, bibliotecas y utilidades de diferentes compañías. De hecho, es casi imposible listar todas las herramientas y utilidades de desarrollo para Java sobre Linux porque cada semana llegan nuevas y lo mejor es que la mayoría de ellas están disponibles bajo la licencia GPL de GNU.

---

[6] Existen varios factores que inhiben el florecimiento de Java en un escritorio donde imperan las aplicaciones nativas Windows.

4.3.1 Sobre el desempeño

Las compañías que crean herramientas para Java sobre Linux no solo están transportando sus aplicaciones a éste sistema operativo sino que, además, se preocupan por proporcionar productos afinados que muestren un buen desempeño.

El modelo de hilos de ejecución es un cuello de botella para el desempeño de Java sobre Linux dado que un solo hilo de Java se corresponde con un solo proceso Linux. Eso evita algunas complicaciones para los que escriben la JVM o el compilador, pero hace a Java sobre Linux más vulnerable a sobrecargas cuando una aplicación utiliza muchos hilos, como sería por ejemplo el caso de una aplicación de conversación. Una solución parcial es el uso de *hilos verdes*; o sea, hilos en el espacio del usuario que corresponden muchos hilos Java a un solo proceso Linux. Esta solución se utiliza en la JVM de Sun.

La técnica de los hilos verdes tiene ciertas desventajas y no permite a la JVM usar multiprocesamiento simétrico. La solución ideal sería que Linux implemente un modelo de muchos hilos a muchos hilos, permitiendo que varios hilos Java correspondan con varios hilos Linux, pero eso requeriría la reescritura del kernel[7].

4.4 Elementos del entorno Java Empresarial sobre Linux

Dentro de un entorno empresarial se debe disponer de las siguientes herramientas:

1. Un paquete de desarrollo Java para compilar y depurar los programas.
2. Soporte para Java servlets y páginas Java de servidor. Estos dos elementos trabajan junto a un servidor Web.
3. Una base de datos para persistencia de objetos. Se debe disponer de un controlador JDBC para utilizar la base de datos.
4. Un servidor de aplicaciones que funcione como contenedor de componentes Java empresariales.
5. Otras herramientas de desarrollo

4.4.1 Paquete de desarrollo Java

Existen varios paquetes de desarrollo Java disponibles para Linux. Estos incluyen:

[7] El lector interesado en este tema puede revisar el documento de IBM llamado "Java, Threads, and Scheduling on Linux" en http://www-4.ibm.com/software/developer/library/java2/index.html.

- Blackdown
- IBM Java Developer Kit
- Sun Java 2 Standard Edition (J2SE)
- Kaffe

La opción más recomendable es utilizar el SDK de Sun, a menos que se indique lo contrario en consideración del software específico que se utilice. Si se requiere de una implementación de código abierto, la alternativa debe ser Kaffe.

4.4.1.1 Sun Java 2 Edición Estándar

La Edición Estándar Java 2 de Sun (J2SE) es la versión oficial de Sun para el sistema operativo Linux. El SDK Java 2 es un entorno de desarrollo para crear aplicaciones, applets y componentes que pueden ser utilizados en la plataforma Java. Incluye todas las herramientas necesarias para compilar y depurar programas escritos en el lenguaje de programación Java y además incluye un entorno de ejecución. Excepto por el ***appletviewer***, no se proporcionan herramientas gráficas.

4.4.1.1.1 Instalación

Para proceder a la instalación se deben seguir los siguientes pasos:

1. Descargar el SDK desde el sitio oficial de Java en http://java.sun.com. Al momento de escribir estas líneas la versión más reciente para Linux es la 1.3.1 en la arquitectura Intel x86 disponible en http://java.sun.com/j2se/1.3/download-linux.html. El SDK se encuentra disponible en el formato de un archivo binario ejecutable. Antes de realizar la descarga se debe aceptar el acuerdo de licencia de Sun.
2. El sistema donde va a instalar el SDK debe cumplir estos requisitos:
   - La versión de kernel debe ser 2.2.5 o superior
   - Se necesita que el sistema tenga instalada la biblioteca glibc v 2.1 o superior
   - 48MB de RAM es la cantidad recomendada.
3. El archivo descargado se debe ejecutar dentro del directorio donde se va a colocar todo el árbol de directorios que conforman la distribución. Luego de presentar el acuerdo de licencia comienza automáticamente el proceso de autoextracción.
4. Opcionalmente pueden definirse las siguientes variables de entorno:

JAVA_HOME

La variable JAVA_HOME puede ser utilizada para acortar la referencia al directorio raíz de la instalación SDK. Por ejemplo:

```
export JAVA_HOME=/usr/local/sun/jdk1.2.2
```

PATH

Para que la referencia a las herramientas de desarrollo sea más sencilla es recomendable incluir el directorio de los ejecutables Java en la ruta de búsqueda.

```
export PATH=$JAVA_HOME/bin:$PATH
```

CLASSPATH

Java 2 ya no necesita que CLASSPATH incluya la ruta hacia las clases del sistema. Sin embargo, el usuario sí querrá agregar a ésta sus propios archivos JAR y los de terceros.

```
export CLASSPATH=$CLASSPATH:$HOME/classes/mio.jar
```

4.4.1.2 Kaffe

Kaffe es una implementación de código abierto de la JVM. Al momento de escribir estas líneas, Kaffe apoya todas las características del SDK 1.1 y partes del SDK 1.2 (Java 2).

Actualmente Kaffe es incluido en varias distribuciones Linux así que su instalación es sencilla. La versión actual 1.0.6 liberada el 25 de julio del 2000 puede ser descargada desde http://www.kaffe.org.

4.4.2 Soporte para Java servlets

Existen varios agregados para servidor Web y para servidor de aplicaciones que proporcionan apoyo a Java servlets dentro de la plataforma Linux. Estos incluyen:

- Allaire Jrun
- Apache JServ
- BEA WebLogic
- Enhydra
- Locomotive
- IBM Websphere

4.4.2.1 Apache JServ

Apache JServ está ahora solo en modo de mantenimiento. Esto significa que ya no habrá nuevas versiones sino solo revisiones. Este ha sido reemplazado por Tomcat del proyecto Jakarta que se examina más adelante.

Apache JServ es un motor de servlets 100% Java que satisface la especificación Servlet 2.0. Los miembros del proyecto han puesto especial atención para asegurar la portabilidad de código así que Apache JServ funciona sobre cualquier máquina virtual Java que “cumpla” la versión 1.1 y puede ejecutar cualquier servlet Java que satisfaga la especificación 2.0.

Apache JServ puede ser descargado desde http://java.apache.org/jserv/index.html.

4.4.3 Soporte para páginas Java de servidor

4.4.3.1 Apache Jakarta Tomcat

Jakarta Tomcat es un contenedor de **servlets** y una implementación de **páginas Java de servidor**. Puede ser utilizado de forma autónoma o en unión con varios servidores Web populares como:

- Apache, versión 1.3 o superior
- Microsoft Internet Information Server, versión 4.0 o superior
- Microsoft Personal Web Server, versión 4.0 o superior
- Netscape Enterprise Server, versión 3.0 o superior

Tomcat usa algo del código escrito para JServ, especialmente el adaptador del servidor Apache. Por los demás es una versión totalmente nueva que satisface la versión 2.0 del API Servlet y la versión 1.1 de JSP.

4.4.3.1.1 Instalación

Para instalar Tomcat en un sistema Linux se deben seguir los siguientes pasos:

1. Descargar el archivo empaquetado (zip o tar.gz) desde la dirección http://jakarta.apache.org/downloads/binindex.html.

2. Desempaquetar el archivo descargado en cualquier directorio. Una ubicación apropiada podría ser en **/usr/local/**. Luego de desempaquetar debería haberse creado un nuevo directorio llamado **tomcat**.
3. Se debe especificar la variable de entorno **TOMCAT_HOME**. Por ejemplo:

```
export TOMCAT_HOME=/usr/local/tomcat
```

4. Adicionalmente el entorno Java; o sea, las variables JAVA_HOME y PATH ya deberían estar configuradas.

Estos pasos son suficientes para ejecutar Tomcat en modo independiente como un contenedor de servlets.

4.4.3.1.2 Probando la instalación

Para asegurarse de que el paquete Tomcat ha sido instalado correctamente se ejecuta el guión de arranque ubicado en **$TOMCAT_HOME/bin**:

```
./startup.sh
```

Si todo está bien especificado el contenedor debe ya estar ejecutándose. Para verificar el arranque se puede examinar el directorio **$TOMCAT_HOME/logs** donde se registran los acontecimientos relacionados con el contenedor.

Para detener el contenedor se utiliza el siguiente guión en **$TOMCAT_HOME/bin**:

```
./shutdown.sh
```

4.4.3.1.3 Integración con Apache

Existen algunos problemas relacionados con utilizar Tomcat como un contenedor independiente:

1. Tomcat no es tan rápido como Apache para tratar con páginas estáticas.
2. Tomcat tiene un conjunto reducido de opciones en contraste con las alternativas de configuración de Apache.
3. Tomcat no es tan robusto como Apache.
4. Las inversiones anteriores que se han hecho dentro de las empresas exigen que se mantenga compatibilidad para continuar usando guiones CGI, PHP, Perl, etc.

Por estas razones una empresa preferirá utilizar un servidor Web, como Apache, para manejar el contenido estático y utilizar Tomcat como un agregado para Servlets y JSPs.

El proceso de integración se puede reducir a dos pasos:

1. Modificar el archivo **httpd.conf** de Apache

   Puesto que Tomcat es un servidor Web independiente, hay que instruir a Apache como diferenciar las solicitudes de páginas estáticas de las solicitudes de servlets y JSPs. Lo que se debe hacer es aumentar la siguiente línea al final del archivo **httpd.conf**:

```
include $TOMCAT_HOME/conf/tomcat-apache.conf
```

   El archivo **tomcat-apache.conf** contiene directivas de configuración que Apache debe procesar antes de manejar servlets. Este archivo define el contexto para Tomcat, parámetros de comunicación y especifica el nombre del módulo que Apache debe cargar para manejar servlets y JSPs.

2. Instalar el adaptador para el servidor Web.

   El adaptador es una pieza de software que no pertenece ni a Apache ni a Tomcat sino que debe colocarse entre ambos para que trabajen juntos.

   El adaptador se llama **mod_jserv** y debe descargarse por separado desde http://jakarta.apache.org/downloads/binindex.html. Si, por alguna razón, la versión compilada de **mod_jserv.so** no funciona sobre un sistema Linux específico habrá que descargar el código fuente de Tomcat desde http://jakarta.apache.org/downloads/sourceindex.html y compilar el adaptador de la siguiente forma:

```
cd tomcat-source/src/native/apache/jserv
apsx -c -o mod_jserv.so *.c
```

   El adaptador debe ser colocado junto con los otros módulos de Apache

```
mv mod_jserv.so /usr/lib/apache
```

   Por las diferencias que existen entre las diferentes distribuciones de Linux pudiese ser necesario ajustar la ruta para que Apache encuentre el adaptador, por ejemplo

```
ln -s /etc/httpd/modules /etc/httpd/libexec
```

   Una vez completados los dos pasos anteriores se debe volver a arrancar el servidor Apache para que surtan efecto todos los cambios (se supone que Tomcat ya fue iniciado)

```
/etc/rc.d/init.d/apache restart
```

Para este momento ya debe ser posible utilizar servlets y JSPs junto con el servidor Web Apache.

La documentación de Tomcat que acompaña a los ejecutables constituye una valiosa ayuda al momento de solucionar cualquier problema.

4.4.4 Bases de datos para persistencia

Existen varias bases de datos que se ejecutan sobre Linux y permiten un acceso utilizando JDBC. Ellas incluyen:

- IBM DB2
- MiniSQL
- MySQL
- Oracle
- PostgreSQL
- Sybase
- Interbase

Por el momento, tres programas, PostgreSQL, MySQL e InterBase son los sistemas más importantes dentro del campo de los RDBMS de código abierto. Pruebas de desempeño han mostrado que PostgreSQL ejecuta de cuatro a cinco veces más rápido que otras bases de datos. MySQL e InterBase muestran buena velocidad en aplicaciones con poco número de usuarios pero su desempeño se deteriora proporcionalmente al aumentar los usuarios y las transacciones.

4.4.4.1 PostgreSQL

PostgreSQL es un sofisticado DBMS objeto-relacional, que apoya la mayoría de construcciones SQL, incluyendo subconsultas, transacciones y tipos y funciones definidas por el usuario. Definitivamente es la base de datos de código abierto más avanzada disponible hoy.

La versión más reciente (por ahora PostgreSQL 7.1) puede ser descargada de http://www.postgresql.org. No obstante, dada su popularidad, está ya incluida en varias distribuciones Linux.

4.4.4.1.1 Controlador JDBC

Antes de instalar el controlador JDBC se debe confirmar que PostgreSQL este aceptando conexiones a través de TCP/IP en lugar de solo sockets de Unix. Esto es necesario para que el controlador JDBC pueda conectarse a la base de datos. Hay más de una forma de hacer esto, una de ellas es cambiar la configuración en **postmaster.init** para que desde el arranque acepte conexiones TCP/IP.

El sitio más recomendable para conseguir el controlador JDBC es en http://jdbc.postgresql.org/downloads.html. Se debe poner especial atención al archivo a descargar puesto que debe estar conforme con la versión de PostgreSQL y la JVM instaladas en el sistema Linux específico.

Una vez que se ha descargado el controlador, debe ser colocado al alcance de las aplicaciones usando CLASSPATH. Por ejemplo:

```
export CLASSPATH=$CLASSPATH:/usr/lib/pgsql/jdbc6.5-1.2.jar
```

Para usar el controlador se deben conocer dos elementos: su clase y el protocolo a utilizarse para la conexión. En el caso de PostgreSQL serán como se muestra en la Tabla 3.2.

Tabla 3.2: Nombre de la clase y el protocolo de acceso JDBC a PostgreSQL

Nombre del controlador	**Protocolo**
Postgresql.Driver	jdbc:postgresql://nombre-host/nombre-base-datos

Si todos los pasos se han efectuado apropiadamente ya está listo para usar PostgreSQL desde Java a través de JDBC.

4.4.5 Servidores para componentes empresariales Java

Existen varias opciones a la hora de elegir un servidor de aplicaciones que soporte componentes empresariales Java. Entre ellas tenemos:

- Sun Java 2 Enterprise Edition (J2EE)
- BEA WebLogic
- JBoss
- Bullsoft JOnAS EJB

4.4.5.1 Sun Java 2 Edición Empresarial

El SDK Java 2 Edición Empresarial (J2EE SDK) es la implementación de referencia para la plataforma Java 2 Edición Empresarial (J2EE). La versión estable más reciente es la 1.2.1. Su propósito es permitir a los desarrolladores probar la tecnología y a los vendedores poseer un elemento de comparación para sus propios productos.

J2EE SDK requiere que el sistema donde se va a instalar posea ya instalado el SDK Java 2 Edición Estándar. La versión necesaria es 1.2.2 o 1.3.

Esta implementación de referencia es bastante grande e incluye lo siguiente:

Servicios

El servidor J2EE inicia los siguientes servicios:

- EJB
- HTTP para Servlets y JSP
- HTTP con SSL para Servlets y JSP
- Autenticación

Utilidades

- Una herramienta para publicar componentes
- Una herramienta para administrar usuarios
- Una herramienta para verificar componentes
- Una herramienta para empacar componentes

El DBMS CloudScape

Esta versión incluye CloudScape 3.0.4 que es un DBMS relacional escrita completamente en Java; sin embargo, se pueden usar otros sistemas.

Para información completa y descarga de este paquete se debe visitar la dirección http://java.sun.com/products/j2ee.

4.4.5.2 JBoss

JBoss es una implementación de la especificación EJB 1.1 (y partes de 2.0), esto es, es un servidor y contenedor de componentes empresariales Java. En esto es similar a la implementación de referencia de Sun J2EE, aunque el núcleo de JBoss proporciona únicamente un servidor EJB y no incluye un contenedor Web para servlets y páginas JSP, aunque existen paquetes que lo juntan con Tomcat o Jetty.

El hecho de que el núcleo sea mínimo significa que JBoss tiene mínimos requerimientos de memoria y espacio en disco. JBoss se ejecutará eficientemente en una máquina de 64 MB de RAM y solo requiere pocos megabytes en disco (incluyendo el código fuente). La implementación de Sun requiere un mínimo de 128 MB de RAM y 31 MB de espacio en disco. Debido a su pequeña marca en la memoria, JBoss arranca hasta 10 veces más rápido que J2EE SDK. Posee un servidor de bases de datos SQL interno para manejar componentes persistentes y éste arranca automáticamente con el servidor. (El servidor CloudScape tiene que ser iniciado separadamente.)

Una de las características más agradables de JBoss es su apoyo para publicación en "caliente". Lo que significa que publicar un componente es tan simple como copiar su archivo JAR dentro del directorio de publicación. Si esto se hace mientras el componente está ya cargado, JBoss lo descarga automáticamente y luego carga la nueva versión. JBoss es distribuido bajo la LGPL, lo que significa que es libre y gratuito, incluso para uso comercial.

JBoss está escrito enteramente en Java y requiere un sistema Java compatible con el SDK 1.3. esto es esencial, no una opción.

4.4.5.2.1 Instalación

1. Descargar el paquete binario desde el sitio oficial de JBoss en http://www.jboss.org o desde http://sourceforge.net/projects/jboss. Al momento de escribir esto la última versión estable es la 2.4.1.
2. Desempaquetar el archivo descargado en cualquier directorio. No se requieren permisos de raíz para ejecutar JBoss y tampoco ninguno de los puertos por omisión están por debajo del rango de puertos privilegiados 1024.
3. Definir las variables de entorno. Además de la ruta hacia el SDK se debe definir una nueva variable **JBOSS_DIST** que señala hacia la raíz de la reciente instalación de JBoss.
4. Para probar la instalación se puede usar el archivo **run.sh** que está dentro del directorio **bin** de la instalación de JBoss.

4.4.5.2.2 Configuración

JBoss se empaca preconfigurado, así que no hay que hacer mucho para ponerlo a funcionar. Sin embargo, posiblemente será necesario hacer cambios menores de configuración para apoyar las aplicaciones y entornos específicos.

La configuración por omisión de JBoss se encuentra localizada en el directorio **conf/default** de la distribución. JBoss permite al Administrador mantener más de una configuración. Todo lo que se tiene que hacer es copiar todos los archivos de la configuración **default** dentro de un nuevo subdirectorio de **conf** y hacer los ajustes necesarios.

Existen varios archivos de configuración. A continuación sigue una descripción de ellos:

jboss.properties

Contiene ciertas propiedades que se cargan en el momento del arranque de JBoss.

jboss.conf

Contiene ciertos MBeans que son necesarios para arrancar JBoss. Usualmente, nunca se tendría que cambiar este archivo.

jboss.jcml

Lista todos los MBeans JMX de servicio que se necesitan incluir en la instancia de JBoss que se ejecuta.

jboss-auto.jcml

JBoss tiene la poderosa capacidad de registrar una instantánea de la ejecución de todos los MBeans en operación, incluyendo sus atributos y luego reproducir ésta instantánea en otra instancia de JBoss. Se tiene la opción de no tomar instantáneas; para hacerlo, simplemente se borra el archivo **jboss-auto.jcml** antes de iniciar JBoss. Así JBoss no registrará ninguno de los cambios durante la ejecución.

mail.properties

Siguiendo la especificación EJB, JBoss proporciona acceso a un recurso de correo a través de la API estándar JavaMail. Este archivo especifica las propiedades del proveedor de correo, como la ubicación de los servidores SMTP y POP e igualmente otras configuraciones relacionada con correo.

jndi.properties

Especifica las propiedades JNDI para los clientes.

standardjaws.xml

Representa una configuración por omisión para la maquinaria CMP de JBoss. Contiene el nombre JNDI de la fuente de datos por omisión, correspondencia de objetos JDBC-SQL por base de datos, especificaciones por omisión para componentes entidad con persistencia manejada por el contenedor, etc.

auth.conf

Este es un archivo de configuración de módulo de conexión JAAS. Contiene especificaciones de autenticación de ejemplo del lado de servidor que son aplicables cuando se usa seguridad basada en JAAS.

standardjboss.xml

Este archivo proporciona la configuración por omisión del contenedor.

Para comprender y manipular cada archivo se deberá revisar detenidamente la documentación incluida en la distribución de JBoss.

4.4.5.2.3 Acceso a bases de datos

Uno de los requerimientos más comunes es crear una o más ***fuentes de datos*** para los componentes empresariales. Una fuente de datos es obligatoria para componentes entidad CMP y es la manera recomendada para interactuar con una base de datos para componentes entidad BMP y componentes de sesión.

Las fuentes de datos de JBoss proporcionan un fondo común de conexiones para bases de datos. Esto significa que cuando una aplicación cierra una conexión, realmente no se cierra, simplemente retorna al estado *ready*. La próxima vez que una aplicación solicite una conexión a base de datos, podría reutilizar la misma conexión.

JBoss apoya cualquier base de datos con un controlador JDBC. Se recomienda usar controladores Java puros (tipos 3 y 4) y especialmente se recomienda no utilizar el puente JDBC ODBC (tipo 1).

4.4.5.2.3.1 Instalando controladores JDBC

Para instalar un controlador JDBC debe estar empacado en uno o más archivos ZIP o JAR.

- Copiar los archivos ZIP o JAR del controlador al directorio **lib/ext** de la distribución JBoss.
- Añadir instrucciones para cargar el controlador en el archivo **jboss.jcml**

  El ejemplo que sigue muestra como son incluidos los dos controladores de base de datos por omisión. Si se utiliza una base de datos diferente de aquella por omisión, hay que añadir el nombre de la clase del controlador a la lista de controladores en éste elemento MBean del **jboss.jcml** personal.

```
<mbean code="org.jboss.jdbc.JdbcProvider"
name="DefaultDomanin:service=JdbcProvider">
<attribute name="Drivers">org.hsql.jbdcDriver,
org.enhydra.instantdb.jdbc.idDriver</attribute>
```

```
</mbean>
```

Si, por ejemplo, se va a utilizar la base de datos Oracle y no las bases de datos por omisión, entonces solo se debe incluir la referencia al controlador Oracle así:

```
<mbean code="org.jboss.jdbc.JdbcProvider"
name="DefaultDomanin:service=JdbcProvider">
<attribute
name="Drivers">oracle.jdbc.driver.OracleDriver</attribute>
</mbean>
```

Si se está utilizando más de un controlador no se debe incluir más de un MBean *org.jboss.jdbc.JdbcProvider*. Todos los controladores deben estar especificados en uno solo de estos.

La próxima vez que arranque JBoss, debería verse una lista con cada uno de los controladores cargados. Si se encuentra un error posiblemente el archivo ZIP o JAR no está en el directorio **lib/ext**.

```
[JDBC] Loaded JDBC driver: oracle.jdbc.driver.OracleDriver
[JDBC] Could not load driver: com.sybase.jdbc2.jdbc.SyBDriver
```

4.4.5.2.3.2 Fondo común de conexiones Minerva

Una vez que se ha instalado el controlador JDBC, se puede añadir uno o más fondos comunes de conexiones (FCC) que lo usen. Cualquier número de EJBs podrían compartir un FCC, pero podría ser conveniente crear varios FCC por cierto número de razones. Por ejemplo, se podría crear un FCC dedicado a una aplicación que requiere muy altos tiempos de respuesta, mientras que otras aplicaciones comparten un FCC de tamaño limitado.

Todas las entradas del FCC son especificadas en el archivo **jboss.jcml**. No se exige la especificación de todos los parámetros puesto que Minerva provee las omisiones. Lo mínimo necesario es PoolName que define el nombre JNDI de la fuente de datos, DataSourceClass, URL de JDBC, usuario y clave de acceso:

```
<mbean code="org.jboss.jdbc.XADataSourceLoader"
name="DefaultDomain:service=XADataSource,name=OracleDB">
<attribute name="PoolName">OracleDB</attribute>
<attribute name="DataSourceClass">
org.opentools.minerva.jdbc.xa.wrapper.XADataSourceImpl</attribute>
<attribute name="URL">
jdbc:oracle:thin:@serverhostname:1521:ORCL</attribute>
<attribute name="JDBCUser">scott</attribute>
<attribute name="Password">tiger</attribute>
</mbean>
```

Para una explicación de detallada de los parámetros y recomendaciones de configuración se debe examinar la documentación de JBoss.

4.4.5.2.3.3 Ejemplo para PostgreSQL

A continuación se indica la configuración mínima para echar a andar una fuente de datos usando PostgreSQL.

lib/ext: En este directorio se coloca el archivo con el controlador JDBC, en este caso `jdbc7.0 1.2.jar`.

jboss.jcml

```
<mbean code="org.jboss.jdbc.JdbcProvider"
name="DefaultDomain:service=JdbcProvider">
<attribute name="Drivers">org.postgresql.Driver</attribute>
</mbean>

<mbean code="org.jboss.jdbc.XADataSourceLoader"
name="DefaultDomain:service=XADataSource,name=PostgresDB">
<attribute name="DataSourceClass">
org.opentools.minerva.jdbc.xa.wrapper.XADataSourceImpl
</attribute>
<attribute name="PoolName">PostgresDS</attribute>
<attribute name="URL">
jdbc:postgresql://host.domain.com/database</attribute>
<attribute name="JDBCUser">username</attribute>
<attribute name="Password">password</attribute>
</mbean>
```

Nota: Se debe incluir un nombre de usuario y una clave de acceso.

Nombre para correspondencia tipo CMP (para jaws.xml): PostgreSQL. Esto se define usando JAWS (véase la sección siguiente).

4.4.5.2.3.4 JAWS

JAWS es el encargado de la correspondencia objeto-relacional usada por JBoss para administrar componentes entidad CMP. JAWS se configura con un archivo llamado **standardjaws.xml**, localizado dentro del directorio de configuración localizado dentro del directorio **conf** de la distribución JBoss. La configuración por omisión es **default**.

Las especificaciones de éste archivo tienen alcance a todo JBoss. Posteriormente se puede extender ésta configuración para cada aplicación colocando un archivo **jaws.xml** dentro del directorio **META-INF** de la aplicación. Después de publicar los componentes, JAWS

procesará primero el archivo **standardjaws.xml**, y luego el archivo **jaws.xml** de cada aplicación si existe.

He aquí lo que se puede hacer con estos dos archivos:

- Especificar una fuente de datos y la correspondencia de tipos a usarse con ella
- Establecer un grupo de opciones relacionadas con el comportamiento de JAWS
- Especificar como JAWS debería crear y usar las tablas
- Definir métodos de búsqueda para acceder a los componentes entidad
- Definir una correspondencia de tipo

Para conocer a fondo las opciones disponibles la referencia absoluta es la DTD para JAWS. Sin embargo, todas las partes son opcionales y solo se definen las necesarias.

4.4.5.2.4 Seguridad en JBoss basada en JAAS

Todo contenedor EJB en JBoss incluye un Interceptor de Seguridad que delega sus verificaciones de seguridad a una implementación de administrador de seguridad. ¿Cómo se decide cuales implementaciones usa determinado contenedor? Esto se especifica por medio del descriptor de publicación de JBoss.

4.4.5.2.4.1 El descriptor de publicación de JBoss (jboss.xml y standardjboss.xml)

El descriptor de publicación de JBoss es el archivo de configuración para publicación específico de una aplicación JBoss. Describe comportamientos que están fuera del alcance del descriptor de publicación de la Especificación EJB. La versión **standardjboss.xml** del archivo se localiza en el directorio **JBOSS_HOME/conf/conf_name** donde *conf_name* es la configuración de ejecución que se especifica a **run.sh** cuando arranca el servidor. El valor por omisión de *conf_name* es **default**. El archivo **standardjboss.xml** especifica los valores por omisión para la configuración global. Se pueden crear descriptores específicos **jboss.xml** que redefinan propiedades específicas o todas ellas según se requiera para la aplicación. Existe un gran número de propiedades que pueden establecerse en el archivo, pero todas son opcionales. Para todos los detalles se debe examinar la DTD de JBoss. Aquí solo nos interesan las tres marcas específicas de seguridad: `security-domain`, `role-mapping-manager` y `authentication-module`.

security-domain

La marca `security-domain` especifica una implementación de las interfaces **org.jboss.security.RealmMapping** y **org.jboss.security.EJBSecurityManager** a ser usada por todas las unidades J2EE publicadas en un EAR o **ejb-jar**. El valor es especificado como el nombre JNDI donde el objeto está localizado.

role-mapping-manager

La marca `role-mapping-manager` especifica la implementación de la interfaz **org.jboss.security.RealmMapping** que va a ser utilizada por el Interceptor de Seguridad. El valor es especificado como el nombre JNDI donde el objeto está localizado. Por lo que a la configuración del contenedor concierne, una implementación de **org.jboss.security.RealmMapping** existe en el espacio de nombres JNDI del servidor JBoss y ésta marca proporciona su posición.

authentication-module

La marca `authentication-module` especifica la implementación de la interfaz **org.jboss.security.EJBSecurityManager** que va a ser utilizada por el Interceptor de Seguridad. El valor es especificado como el nombre JNDI de donde está localizado el objeto.

Ejemplo

```
<?xml version="1.0"?>
<jboss>
    <!--Todos los contenedores de componentes usan éste
       administrador de seguridad por omisión -->
    <security-domain>java:/jaas/ejemplo1</security-domain>  (1)
      <container-configurations>
      <!--Redefine la función de correspondencia de roles de
      security-domain para componentes de sesión sin estado -->
      <container-configuration>
      <!-- Usamos el nombre container-name de standardjboss.xml
      para especificar los elementos que deseamos redefinir -->
      <container-name>Standard Stateless SessionBean</container-name>
            <role-mapping-manager>
                  java:/jaas/session-roles
            </role-mapping-manager> (2)
      </container-configuration>
    </container-configurations>
</jboss>
```

(1) Establece un administrador global de seguridad a través de la marca `security-domain`.

(2) Redefine la función de correspondencia de roles del administrador global de seguridad para componentes de sesión sin estado.

Aquí se está asignando un administrador global de seguridad para todos los componentes al objeto localizado en **java:/jaas/example1** y se indica un administrador diferente para correspondencia de roles para el contenedor "**Standard Stateless SessionBean**". Esto significa que cualquier componente de sesión sin estado empacado en el archivo EAR o JAR utilizará **RealmMapper** localizado en **java:/jaas/session-roles** en lugar del indicado por la marca `security-domain`. Se examinan los nombres de la forma **java:/jaas/XXX** más adelante.

4.4.5.2.4.2 Configuración de la implementación del Administrador de Seguridad en JNDI

Lo que sigue es cómo ligar las implementaciones dentro del espacio de nombres del servidor JBoss. La respuesta está en crear un componente JMX que crea y liga las implementaciones deseadas durante el arranque del servidor. El componente **JaasSecurityManagerService** ha sido escrito para realizar ésta configuración.

Para configurar **JaasSecurityManagerService**, se debe ubicar la siguiente entrada en el archivo **JBOSS_HOME/conf/default/jboss.jcml**:

```
<!-- Administrador de seguridad JAAS y correspondencia de reino -->
<mbean code="org.jboss.security.plugins.JaasSecurityManagerService"
      name="Security:name=JaasSecurityManager">
    <attribute name="SecurityManagerClassName">
      org.jboss.security.plugins.JaasSecurityManager</attribute>
    <attribute name="SecurityProxyFactoryClassName">
      org.jboss.security.SubjectSecurityProxyFactory</attribute>
  </mbean>
```

Si está comentada o no existe, habrá que ponerla. El servicio **JaasSecurityManagerService** crea una referencia a un contexto JNDI en **java:/jaas** que lía débilmente instancias de **org.jboss.security.plugins.JaasSecurityManager** bajo **java:/jaas** según sean solicitados vía JNDI. Los detalles de cómo se logra esto no son importantes. Lo que interesa es la configuración de **JaasSecurityManagerService**, cualquier búsqueda en el contexto inicial JNDI del servidor JBoss usando un nombre de la forma **java:/jaas/xyz** resulta en un objeto del tipo

org.jboss.security.plugins.JaasSecurityManager que tiene el nombre *xyz*. Traducido a código, esto es:

```
InitialContext ctx = new InitialContext();
JaasSecurityManager jsm1 =
      (JaasSecurityManager) ctx.lookup("java:/jaas/xyz");
String securityDomain = jsm1.getSecurityDomain();
// securityDomain == "xyz"
```

donde `jsm1` es una instancia de **JaasSecurityManager** que fue creada usando el nombre "*xyz*". Se utiliza ésta facilidad para ligar una única instancia de **JaasSecurityManager** para usarse como la implementación de **RealmMapping** y **EJBSecurityManager** en el descriptor **jboss.xml** anterior. Esto se puede hacer porque **JaasSecurityManager** implementa ambas interfaces. A continuación se examina cómo se autentifican realmente los usuarios y se especifica los roles/identidades que ellos poseen con un **JaasSecurityManager**.

4.4.5.2.4.3 Uso de JaasSecurityManager

Como se puede suponer, **JaasSecurityManager** usa JAAS (Servicio Java de Autenticación y Autorización) para implementar la función de autenticación de usuarios y correspondencia de roles de ambas interfaces **RealmMapping** y **EJBSecurityManager**.

Esto se hace creando un Sujeto JAAS usando el mecanismo **javax.security.auth.login.LoginContext**. Cuando **JaasSecurityManager** necesita autenticar un usuario, hace una conexión (*login)* JAAS usando los siguientes pasos:

```
Principal principal = ... pasado por SecurityInterceptor;
Object credential = ... pasado por SecurityInterceptor;
/* Acceso al dominio de seguridad al cual está ligado el administrador de
seguridad. Este es el componente xyz del nombre java:/jaas/xyz usado en
la definición de las marcas security-domain o role-mapping-manager. */
String name = getSecurityDomain();
CallbackHandler handler = new
org.jboss.security.plugins.SecurityAssociationHandler();
handler.setSecurityInfo(principal, credential);
LoginContext lc = new LoginContext(name, handler);
// Validar al usuario, credencial usando LoginModules
// configurado para 'name'
lc.login();
Subject subject = lc.getSubject();
Set subjectGroups = subject.getPrincipals(Group.class);
// Obtiene el Grupo cuyo nombre es 'Roles'
Group roles = getGroup(subjectGroups, "Roles");
```

Si se conoce JAAS, se apreciará que el nombre que fue usado en la creación de **JaasSecurityManager** armoniza con el nombre de configuración del Contexto de

Conexión (*LoginContext*). El objeto de Contexto de Conexión JAAS mira a un objeto de configuración que está hecho de las secciones nombradas que describen los Módulos de Conexión necesarios de ejecutar para realizar la autenticación. Esta abstracción permite a la API de autenticación ser independiente de la implementación particular. La autenticación de usuarios y la asignación de roles de usuario recae en implementar **javax.security.auth.spi.LoginModule** y crear la entrada de configuración de conexión que armoniza con el nombre **JaasSecurityManager**. Existe cierto número de implementaciones de Módulo de Conexión en el paquete **org.jboss.security.auth.spi**. Aquí examinaremos el caso sencillo de **UsersRolesLoginModule** para demostrar como configurar los Módulos de Conexión y que trabajen con **JaasSecurityManager**.

4.4.5.2.4.4 Uso de UsersRolesLoginModule

La clase **UsersRolesLoginModule** es una implementación sencilla basada en archivo de propiedades Java (**users.properties** y **roles.properties**) para realizar la autenticación y correspondencia de roles respectivamente.

users.properties

El archivo **users.properties** es un archivo de propiedades Java que especifica la correspondencia entre nombre de usuario y clave de acceso. Su formato es:

```
nombreusuario1=claveacceso1
nombreusuraio2=claveacceso2
...
```

con una entrada por línea.

roles.properties

El archivo **roles.properties** es un archivo de propiedades Java que especifica la correspondencia de nombre de usuario a rol (o roles). Su formato es:

```
nombreusuario1=rol1[,rol2,...]
nombreusuario2=rol1
...
```

con una entrada por línea. Si un usuario tiene varios roles estos se especifican usando una lista separada por comas. Se puede especificar grupos de roles usando la sintaxis:

```
nombreusuario1.NombreGrupo1=rol1[,rol2,...]
nombreusuario2.NombreGrupo2=rol1
...
```

Cuando no se especifica un nombre de grupo el nombre implícito es “*Roles*”.

4.4.5.2.4.5 El archivo de configuración de Módulo de Conexión

Por omisión JAAS utiliza un archivo de configuración de Módulo de Conexión para describir cuales instancias de Módulo de Conexión necesitan ser ejecutadas durante una conexión. La configuración por omisión para el servidor JBoss está en **JBOSS_HOME/conf/default/auth.conf**. La sintaxis es:

```
Nombre {
      Nombre_clase_módulo_conexión (required | optional|...)
      [opciones]
      ;
};
```

Ejemplo de un archivo de configuración

```
ejemplo1 {
      org.jboss.security.auth.spi.UsersRolesLoginModule required
      ;
};

ejemplo2 {
/* Un Módulo de Conexión basado en JDBC
Opciones del módulo:
      dsJndiName: El nombre de la fuente de datos de la base de datos que
      contiene las tablas Principales, Roles
      principalsQuery: La sentencia preparada equivalente a:
    "select Password from Principales where PrincipalID=?"
      rolesQuery: La sentencia preparada equivalente a:
    "select Role, RoleGroup from Roles where PrincipalID=?"
*/
    org.jboss.security.auth.spi.DatabaseServerLoginModule required
    dsJndiName="java:/DefaultDS"
    principalsQuery="select Password from Principals where PrincipalID=?"
    rolesQuery="select Role, RoleGroup from Roles where PrincipalID=?"
    ;
};
```

Esto indica que el **UsersRolesLoginModule** que se va a utilizar está fijado a la configuración llamada “*ejemplo1*”. Este nombre también concuerda con la porción de dominio de seguridad del nombre JNDI en **java:/jaas/ejemplo1** usado en la marca `security-domain` del ejemplo mostrado para el archivo **jboss.xml**. La correlación entre el valor de la marca `security-domain` y la entrada en el archivo de configuración de conexión determina cuales Módulos de Conexión ejecuta **JaasSecurityManager** para realizar la autenticación y autorización. Cuando el usuario intenta ejecutar métodos en EJBs asegurados bajo el dominio de seguridad **java:/jaas/ejemplo1**, el usuario será

autenticado contra **UsersRolesLoginModule** puesto que éste es el Módulo de Conexión establecido bajo el nombre **ejemplo1** en el archivo **auth.conf** del servidor.

También existe una versión de **auth.conf** del lado del cliente que utiliza el cliente que se conecta a JBoss. Este está localizado en **JBOSS_HOME/client/auth.conf** y el contenido de la versión por omisión se muestra a continuación. La entrada clave aquí es la entrada "*other*" que contiene "`org.jboss.security.ClientLoginModule required;`".

```
srp {
    // Ejemplo de auth.conf cliente para usar SRPLoginModule
    org.jboss.srp.jaas.SRPLoginModule required
        password-stacking="useFirstPass"
        principalClassName="org.jboss.security.SimplePrincipal"
        srpServerJndiName="SRPServerInterface"
        debug=true
        ;

    // JBoss LoginModule
    org.jboss.security.ClientLoginModule  required
        password-stacking="useFirstPass"
        ;

    // Ponga aquí sus módulos de conexión que necesita jBoss
};

other {
    // Ponga aquí sus módulos de conexión que trabajan sin jBoss

    // jBoss LoginModule
    org.jboss.security.ClientLoginModule  required;

    // Ponga aquí sus módulos de conexión que necesita jBoss
};
```

Nota: La configuración llamada "*other*" es usada por JAAS siempre que no pueda encontrar una entrada correspondiente al nombre pasado en el constructor del Módulo de Conexión.

4.4.5.3 Bullsoft JOnAS EJB

JOnAS es un proyecto de código abierto que implementa un servidor de aplicaciones 100% Java. Este servidor de aplicaciones proporciona un ambiente para alojar varios contenedores, uno por cada componente empresarial Java.

Puesto que se trata de una implementación 100% Java, JOnAS utiliza código Java para todos los servicios de que es responsable un servidor de aplicaciones; es decir: medios de almacenamiento, servicios de transacciones, servicios de seguridad y servicios de nombres.

Por ejemplo, para comunicarse con una base de datos utiliza la API JDBC y para apoyar el uso de transacciones utiliza la API JTM.

4.4.5.3.1 Instalación

Para instalar el servidor JOnAS se deben realizar las siguientes acciones:

1. Descargar desde el sitio de Bullsoft en http://www.bullsoft.com/ejb la versión compilada del servidor de aplicaciones JOnAS. Alternativamente se puede descargar el código fuente completo para compilarlo posteriormente.
2. Asegurar la disponibilidad de las siguientes bibliotecas de clases en el sistema Linux específico:
   - La API JNDI: **jndi.jar, providerutil.jar, rmiregistry.jar**. Estas bibliotecas pueden ser descargadas desde http://java.sun.com/products/jndi/.
   - El paquete opcional JDBC 2.0 (las extensiones javax.sql): **jdbc2_0-stdext.jar**. Puede ser descargado desde http://java.sun.com/products/jdbc/downloads.html.
   - La API de transacciones Java (JTA): **jta-spec1_0_1.jar**. Puede ser descargada desde http://java.sun.com/products/jta/. Antes de realizar la descarga se debe tomar en cuenta que, al menos para JTA versión 1.01, las clases JTA están unidas a la especificación. Por lo tanto lo que se debe descargar es el archivo de la especificación llamado **jta-spec1_0_1.zip** y luego extraer de éste la API.
   - La API EJB: **ejb.jar**. Esta no se encuentra disponible de forma independiente, así que para obtenerla habrá que descargar todo el paquete J2EE desde http://java.sun.com/j2ee/download.html y luego crear el archivo **ejb.jar** con todas las clases que están dentro del directorio **javax/ejb** dentro del paquete J2EE.

   JOnAS va a utilizar todas estas clases y por omisión las buscará en el directorio **/usr/local/lib/**, así que es allí donde deberían ser colocadas.
3. JOnAS necesita que se definan las siguientes variables de entorno:

   JONAS_ROOT

   Debe contener la ruta hacia el directorio raíz de la instalación del servidor. Por ejemplo:

```
export JONAS_ROOT=/usr/local/jonas_jdk1.2
```

PATH

Para facilitar la invocación de los archivos binarios es conveniente agregar el directorio **bin** de JOnAS a la ruta de búsqueda. Así:

```
export PATH=$JONAS_ROOT/bin:$PATH
```

CLASSPATH

Contiene la ruta hacia varias clases que necesita JOnAS. Debería contener la ruta hacia el controlador JDBC e incluir la raíz de la distribución.

```
export CLASSPATH=/usr/lib/jdbc.jar:$JONAS_ROOT:$CLASSPATH
```

XTRA_CLASSPATH

Se puede utilizar para indicar a JOnAS donde se encuentran clases adicionales requeridas por los componentes.

4. Editar los archivos de configuración del servidor de aplicaciones de acuerdo con las particularidades del sistema Linux específico. Los archivos de configuración son:

   jonas.properties

   Es utilizado para indicar entre otras cosas cuales son los componentes a cargar dentro del servidor y el nombre de la fuente de datos a usar vía JDBC.

   jndi.properties

   El acceso a JNDI está especificado dentro de este archivo que debe ser visible desde CLASSPATH. El cambio que se debe hacer es retirar ***localhost*** y indicar el nombre de la computadora donde va a funcionar el servidor.

4.4.5.3.2 Acceso a bases de datos

Con el propósito de usar una o más bases de datos relacionases, JOnAS creará y utilizará uno o más objetos **DataSource**. Debe existir un archivo de configuración por cada base de datos a emplearse donde se indican los parámetros de conexión JDBC como el nombre del controlador y la ruta hacia la base de datos. Por ejemplo, supongamos que se va a utilizar

PostgreSQL, el archivo se podría llamar **PostgreSQL.properties**. Dentro de la raíz de la distribución existen plantillas guía que indican los parámetros mínimos de esta clase de archivos.

No se debe olvidar especificar el nombre de estos archivos en la entrada **jonas.datasources** del archivo **jonas.properties**. Por ejemplo si se va a utilizar PostgreSQL y se ha creado **PostgreSQL.properties** la entrada será así:

```
jonas.datasources              PostgreSQL
```

En caso de tener más de un archivo la lista se separa con comas.

Si se han realizado apropiadamente estas acciones JOnAS está ya listo para funcionar. A modo de verificación se pueden ejecutar los ejemplos que acompañan la distribución.

4.4.6 Otras herramientas de desarrollo

Aunque las herramientas referidas a continuación no son esenciales para la creación de un entorno empresarial, sí son de gran utilidad a la hora de desarrollar aplicaciones de nivel empresarial.

4.4.6.1 NetBeans

NetBeans es un entorno de desarrollo integrado modular, basado en estándares y escrito en Java. Permite la creación de aplicaciones Java del lado cliente y del lado servidor, con un amplio rango de características desde el desarrollo y depuración de páginas Java de servidor hasta el apoyo integrado CVS y mucho más. Todas estas piezas de funcionalidad están implementadas en forma de módulos que se pueden enchufar al núcleo de NetBeans.

Desde junio del 2000, NetBeans se desarrolla bajo código abierto. El código base fue cedido por Sun Microsystems.

NetBeans puede ser extendido añadiendo o reemplazando módulos. Productores independientes pueden crear sus propias versiones de NetBeans combinando y agregando módulos. Luego pueden comercializar u ofrecer libremente el producto final.

La versión más reciente de NetBeans se puede descargar del sitio http://www.netbeans.org. Existen versiones para diferentes plataformas incluida una para Linux. La instalación es muy sencilla y mayormente automática.

4.4.6.2 Ant

Ant es un herramienta de construcción basada en Java. En teoría, es una clase de *make* sin los inconvenientes de *make*.

¿Por qué otra herramienta de construcción cuando ya existe *make*? Pues porque todas las herramientas de esa clase tienen limitaciones con las que el autor original de Ant no podía vivir cuando desarrollaba software de plataforma cruzada. Las herramientas de construcción como *make* están inherentemente basadas en el interprete de comandos de cada sistema operativo: evalúan un conjunto de dependencias y luego ejecutan comandos no muy diferente de como un programador lo haría dentro de un interprete de comandos. Esto significa que se pueden extender fácilmente usando o escribiendo cualquier programa para el sistema operativo en el que se trabaja; sin embargo, esto también significa que se crea una dependencia hacia el sistema operativo.

Ant se diferencia porque utiliza un modelo donde la funcionalidad se extiende usando clases Java. En lugar de escribir comandos del interprete, los archivos de configuración están basados en XML y especifican un árbol donde existen varias tareas. Cada tarea es ejecutada por un objeto que implementa una interfaz **Task** particular.

Ant es un proyecto que se desarrolla bajo código abierto. La versión estable más reciente se puede descargar desde http://jakarta.apache.org/ant/.

La instalación de la distribución no es difícil pero para lograr usar Ant en toda su capacidad se requiere de un buen conocimiento sobre cómo crear el archivo de configuración XML y especialmente cómo especificar las tareas.

4.4.6.3 Merlot

Merlot es una aplicación para crear y editar archivos XML que se ejecuta sobre una máquina virtual Java 2 (JDK 1.2.2 o JDK 1.3). Simplifica la interacción con un archivo XML ocultando la apariencia tosca del texto XML.

Proporciona una interfaz gráfica para añadir, remover y acomodar elementos XML. Igualmente proporciona una interfaz agradable para editar los atributos de los elementos. Merlot ha sido desarrollada para usuarios que pudiesen no estar muy familiarizados con XML pero también proporciona una interfaz útil para gente que conoce mucho de XML.

Es muy fácil de instalar. La última versión siempre está disponible en http://www.merlotxml.org. Existen versiones para diferentes plataformas incluyendo una para Linux. Puesto que Merlot se desarrolla bajo código abierto es posible también conseguir el código fuente si se desea.

4.4.6.4 Poseidón para UML

Poseidón para UML está basado en ArgoUML. ArgoUML fue concebida como una herramienta y entorno para usarse en el análisis y diseño orientado a objetos de sistemas de software. Posee un número de distinciones muy importantes respecto de otras herramientas:

- ArgoUML incluye cierto número de características que apoyan las necesidades cognitivas de los diseñadores y arquitectos de software orientado a objetos.
- ArgoUML apoya estándares abiertos como UML, XMI, SVG, OCL y otros. En esto está por delante de muchas herramientas comerciales.
- ArgoUML es una aplicación 100% Java. Esto le permite ejecutarse sobre cualquier plataforma donde existe una máquina virtual Java.
- ArgoUML es un producto de código abierto. La disponibilidad del código asegura el continuo desarrollo sustentado por una comunidad de usuarios interesados.

Muchas características de la herramienta han evolucionado dentro del proyecto de código abierto, pero algunas más avanzadas requieren de un esfuerzo de tiempo completo que solo una compañía comercial puede proporcionar. Gentleware AG proporciona extensiones para ArgoUML dentro de un producto llamado Poseidón para UML que es más maduro y más estable y se mantiene como código abierto.

Con Poseidón para UML se puede trabajar con todos los diagramas UML y todos los elementos de diagramas están implementados. Se pueden guardar y almacenar proyectos,

generar código Java, usar ingeniería inversa sobre código Java y mucho más. En pocas palabras, proporciona todo lo que se necesita para aprender y usar UML.

Para utilizar Poseidón para UML se lo puede descargar desde el sitio http://www.gentleware.com. La instalación es sencilla y prácticamente automática.

Capítulo 5

Prototipo de Sistema Financiero

En los capítulos anteriores se han estudiado los sistemas distribuidos de múltiples capas y se ha profundizado en su implementación dentro de una plataforma Java Empresarial apoyada por el sistema operativo Linux. Para aplicar en un caso concreto todo lo aprendido, se ha creado un prototipo de un sistema financiero para una cooperativa de ahorro y crédito. Tal sistema fue elegido durante la elaboración del Plan de Disertación por las características que posee como por ejemplo la manipulación de datos de miles de socios por parte de decenas de usuarios dispersos en diferentes áreas.

Este capítulo está dividido en tres secciones principales. La primera se ocupa de la planificación del proyecto (especificando el ámbito del software, objetivos, funciones, etc.). La segunda sección recoge los artefactos producidos como resultado de la fase del análisis. La última sección especifica los artefactos producidos por el diseño requeridos para una implementación específica.

5.1 Planificación del proyecto

A continuación se recopila la información generada durante el proceso de planificación del proyecto “PROTOTIPO DE UN SISTEMA FINANCIERO PARA UNA COOPERATIVA DE AHORRO Y CRÉDITO”.

Este sistema sirve a dos propósitos fundamentales: (1) llevar un registro de todos los socios de la cooperativa y (2) permitir el registro de las transacciones de depósito y retiro.

5.1.1 Objetivos

- Permitir la administración de los datos de los socios de la cooperativa.
- Procesar el registro de las transacciones con libretas de ahorros.
- Proveer un entorno seguro donde todas las operaciones (tanto de datos de socios como de transacciones) sean confiables.
- Generar un pequeño grupo de reportes básicos.

5.1.2 Funciones

1. Registrar los datos personales del socio.
2. Permitir consultar los datos personales de un socio.
3. Actualizar los datos personales de un socio.
4. Abrir una libreta de ahorros para un socio ya registrado.
5. Registrar el depósito de dinero en efectivo en la libreta de un socio.
6. Registrar el retiro de dinero en efectivo en la libreta de un socio.
7. Acreditar intereses en todas las libretas.
8. Consultar saldos de una libreta.
9. Proporcionar una infraestructura segura para todos los datos.

5.1.3 Riesgos del proyecto

5.1.3.1 Riesgos identificados

1. Dado el contexto en que se está desarrollando el proyecto (la disertación de grado), la agenda es apretada y se dispone de tiempo limitado para el desarrollo.
2. Otro riesgo está relacionado con la utilización de la plataforma Java Empresarial para la implementación del proyecto. Puesto que la disertación involucra el aprendizaje de la mencionada plataforma, las capacidades de ella no podrán ser maximizadas.
3. Siempre existe un riesgo potencial en el diseño del componente de interacción humana pues quizá no se puedan satisfacer las expectativas del usuario, especialmente en consideración al punto anterior.

5.1.4 Recursos del proyecto

5.1.4.1 Recursos hardware

Para el desarrollo de éste proyecto se requiere de un sistema de cómputo (PC) de alto rendimiento con las siguientes características:

- Microprocesador Pentium III o superior
- Un mínimo de 96 MB de RAM
- Unidad de disco duro con un espacio libre de 600 MB
- Monitor SVGA capaz de una resolución de al menos 800 por 600 pixeles y una capacidad de colores de 16 bits.
- Unidad de disco flexible de 3.5", 1.44 MB.

- Unidad de CD-ROM

5.1.4.2 Recursos software

El software de desarrollo debe ser de alta ejecución.

5.1.4.2.1 Herramientas CASE

- Poseidón para UML
- Merlot

5.1.4.2.2 Herramientas para el entorno de desarrollo

- NetBeans
- Ant
- JBoss
- PostgreSQL
- Paquete de desarrollo Java. Se utiliza la versión 1.3 de la plataforma Java 2.

5.2 Metodología de desarrollo de software

En el mundo mercantil se utiliza J2EE para resolver problemas comerciales, para desarrollar software de negocios, o para proporcionar servicios a otros proyectos. Si una compañía desea construir un sitio Web para comercio electrónico, usando una arquitectura multicapa, usualmente requiere de gerentes, arquitectos, diseñadores, programadores, revisores y expertos de bases de datos durante todo el ciclo de vida de desarrollo.

Para que las diferentes partes trabajen eficiente y efectivamente, se utiliza una metodología de desarrollo de software. Algunos modelos clásicos de desarrollo son el modelo cascada, Desarrollo Rápido de Aplicaciones (RAD), y Programación Extrema (XP). Para el Sistema Financiero se ha escogido un proceso de desarrollo llamado **Proceso Unificado Racional** (**RUP**). RUP proporciona un enfoque disciplinado en la asignación de tareas y responsabilidades a los diferentes roles. Sus objetivos aseguran la producción de software de alta calidad que satisface las necesidades del usuario dentro de un calendario y presupuesto predecibles.

Existen tres buenas razones por las que usar RUP para J2EE:

1. RUP está centrado en la arquitectura. Desarrolla un prototipo arquitectónico ejecutable antes de asignar recursos al desarrollo de escala completa.
2. RUP es iterativo y está basado en componentes. La arquitectura básica a menudo incluye un marco o infraestructura para facilitar añadir componentes a través de iteraciones que personalicen y extiendan la funcionalidad del sistema sin afectar al resto de elementos.
3. RUP emplea un lenguaje estándar, UML, para modelar visualmente la arquitectura de un sistema y sus componentes.

RUP tiene cuatro fases diferentes de desarrollo:

- **Origen**.- La idea inicial. Aquí se especifica la visión del producto final y se delimita el alcance del proyecto.
- **Elaboración**.- La planeación de las actividades necesarias y los recursos requeridos, especificando las características y diseñando la arquitectura.
- **Construcción**.- La creación del producto y la evolución de la visión, la arquitectura y los planes hasta que el producto esta listo para ser entregado a los usuarios.
- **Transición**.- Se realiza la transición del producto a su comunidad de usuarios, lo que incluye embalaje, entrega, entrenamiento, apoyo y mantenimiento del producto hasta que el usuario está satisfecho.

5.3 Resumen del análisis orientado a objetos

Las siguientes páginas contienen la especificación de requisitos del software y los artefactos del AOO para el sistema objeto de estudio.

De nuestro análisis se desprende que se deben construir tres piezas de software. (ver Figura 5.1)

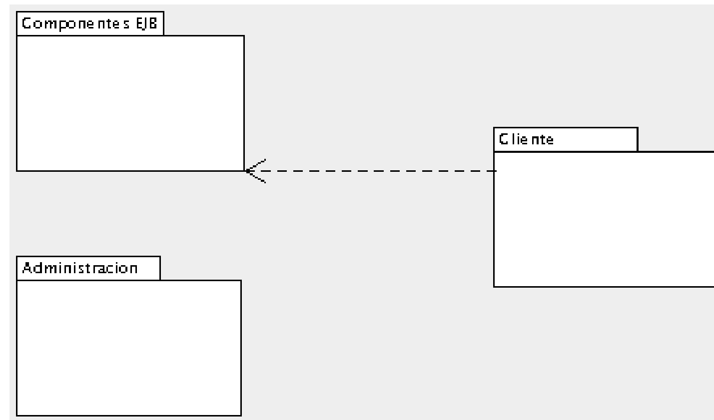

Figura 5.1: Componentes principales del Sistema Financiero

La primera agrupa todos los componentes empresariales que se van a colocar dentro del servidor de aplicaciones. La segunda está dedicada a una aplicación cliente que utiliza los componentes empresariales para satisfacer los requisitos funcionales. Y la última consiste en una consola de administración del sistema encargada del manejo de los usuarios y funciones de la base de datos.

5.3.1 Casos de uso de alto nivel

Para mejorar la comprensión de los requerimientos se han creado varios casos de uso, que están representados en la Figura 5.2.

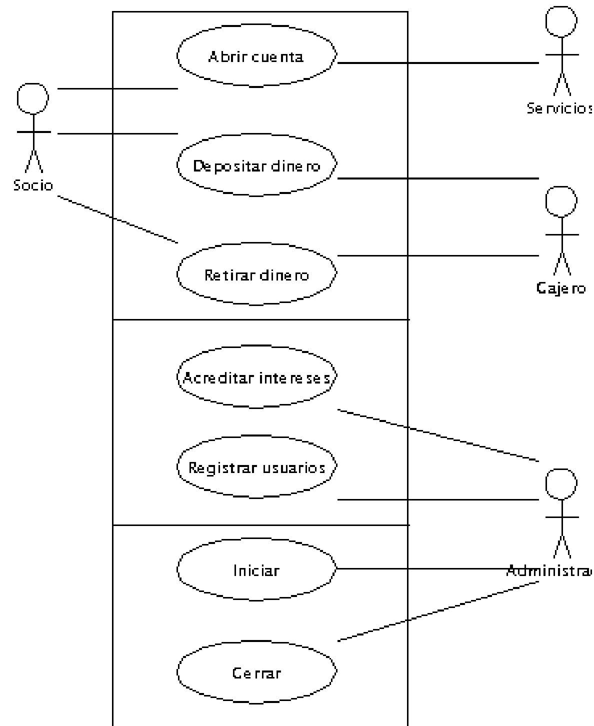

Figura 5.2: Diagrama de casos de uso del Sistema Financiero

Caso de uso: Acreditar intereses

Actores: Administrador

Tipo: Primario

Descripción: El Administrador del sistema solicita que se consigne la cantidad apropiada en la libreta de cada Socio.

Caso de uso: Registrar usuarios

Actores: Administrador

Tipo: Primario

Descripción: El Administrador registra y asigna roles a un nuevo usuario en el sistema.

5.3.2 Casos de uso expandidos

5.3.2.1 Caso de uso: Abrir cuenta

Actores: Socio (iniciador), Servicios al Cliente.

Propósito: Registrar un nuevo socio y crear una nueva cuenta.

Resumen: Un Socio llega hasta Servicios al Cliente y solicita abrir una cuenta. La persona de Servicios al Cliente toma sus datos personales y hace la apertura. El Socio realiza el depósito inicial.

Tipo: Primario y esencial

Referencias: *Funciones*: 1, 4

Casos de uso: Terminado éste caso de uso se debe seguir al caso de uso Depositar dinero en efectivo

Curso normal de los eventos

Acción de los actores	Respuesta del sistema
1. Inicia con una persona que llega a servicios al Cliente para abrir una cuenta	
2. La persona en Servicios al Cliente solicita un nuevo número de Socio.	3. Se genera un nuevo número basándose en los registros anteriores.
4. La persona en Servicios al Cliente ingresa los datos personales usando el formulario llenado por el Socio.	5. Se reciben los nuevos datos y se realizan las validaciones necesarias.
	6. Se procede a almacenar los datos del nuevo Socio y se emite la solicitud de ingreso.

Acción de los actores	Respuesta del sistema
7. El Socio firma la solicitud y se dirige a la Caja para hacer el depósito de apertura.	

5.3.2.2 Caso de uso: Depositar dinero en efectivo

Actores: Socio (iniciador), Cajero

Propósito: Consignar la cantidad de dinero en la cuenta de un socio

Resumen: Un Socio llega hasta la ventanilla de un Cajero con una papeleta y el dinero a depositar. El Cajero hace el depósito e iguala la libreta del Socio.

Tipo: Primario y esencial

Referencias: *Funciones*: 5

Curso normal de los eventos

Acción de los actores	Respuesta del sistema
1. Comienza con un Socio que llega a una Caja con una papeleta de depósito, una cantidad de dinero en efectivo y su libreta de ahorros.	
2. El Cajero recibe la papeleta, el dinero y la libreta.	
3. El Cajero cuenta el dinero y revisa los datos de la papeleta. Curso alterno: Si la papeleta está incorrecta se lo indica al Socio para que la corrija.	

Acción de los actores	Respuesta del sistema
4. El Cajero inicia una transacción de depósito para el número de cuenta dado.	5. Verifica el número de cuenta y su estado
	6. Muestra al Cajero los datos del titular de la cuenta
7. El Cajero indica el total de dinero a depositar.	8. Se actualiza el saldo de la cuenta.
	9. Se registra la transacción de depósito a nombre de este Cajero.
	10. Se actualiza la libreta del Socio.
11. El Cajero entrega la libreta actualizada.	

Curso alterno:

- Si el Socio no trae su libreta de ahorros, se realiza la operación pero queda pendiente la actualización.

5.3.2.3 Caso de uso: Retirar dinero en efectivo.

Actores: Socio (iniciador), Cajero

Propósito: Retirar de una cuenta cierta cantidad de dinero

Resumen: Un Socio llega hasta la ventanilla de un cajero con una papeleta de retiro y su libreta. El Cajero usa la papeleta para hacer el retiro, iguala la libreta y entrega al Socio el dinero.

Tipo: Primario y esencial

Referencias: *Funciones*: 6

Curso normal de los eventos

Acción de los actores	Respuesta del sistema
1. Comienza con un Socio que llega a la ventanilla portando una papeleta de retiro, su documento de identidad y la libreta de ahorros.	
2. El Cajero verifica la papeleta contra el documento de identidad. Curso alterno: Si no coinciden los datos se lo indica al Socio.	
3. El Cajero inicia la transacción de retiro para el número de cuenta dado.	4. Verifica el número de cuenta y su estado.
	5. Con el número de cuenta el sistema busca los datos de su titular.
6. El Cajero revisa las rúbricas en la papeleta, en el documento de identidad, y aquella registrada en el sistema. Curso alterno: Si las rúbricas no coinciden se indica el error y se detiene.	
7. El Cajero indica la cantidad a retirar.	8. El sistema verifica que el valor solicitado no exceda el saldo disponible (y opcionalmente un mínimo necesario). Curso alterno: Si la cantidad excede al saldo disponible se detiene y notifica el

Acción de los actores	Respuesta del sistema
	error.
	9. Se realiza el descuento del dinero solicitado y se actualiza la libreta.
10. El Cajero entrega el dinero solicitado junto con la libreta actualizada.	

5.3.3 Modelo Conceptual

Un modelo conceptual explica los conceptos significativos en un dominio; es el artefacto más importante a crear durante el análisis orientado a objetos. Los siguientes diagramas de clases representan el modelo conceptual para cada uno de los módulos del Sistema Financiero.

5.3.3.1 Clases del módulo Servidor

La Figura 5.3 muestra el diagrama de las clases necesarias para implementar las funciones del servidor.

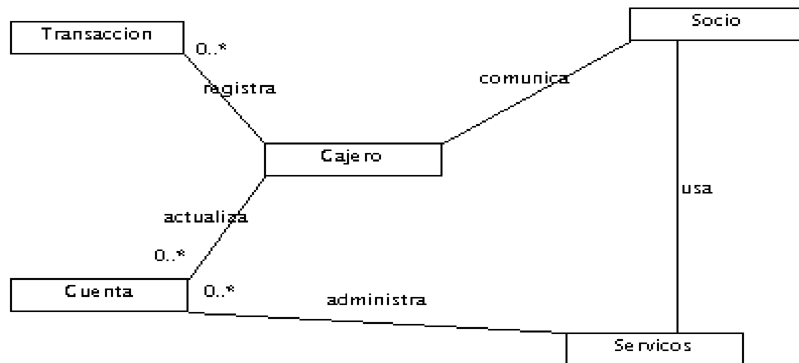

Figura 5.3: Diagrama de clases inicial del Módulo Servidor

Las figuras que siguen a continuación muestran de mejor manera cómo cada clase se transforma en un componente empresarial Java para implementar la funcionalidad requerida. Nótese que para los componentes entidad se muestra el esquema de la tabla correspondiente.

La Figura 5.4 presenta el componente Socio. Su funcionalidad está dividida entre su interfaz local *SocioHome* que proporciona un método de creación de instancias y uno de búsqueda por clave primaria. El resto de la funcionalidad está especificada por la interfaz remota *Socio* con un método para leer (**getAll**) y otro para escribir datos (**setAll**).

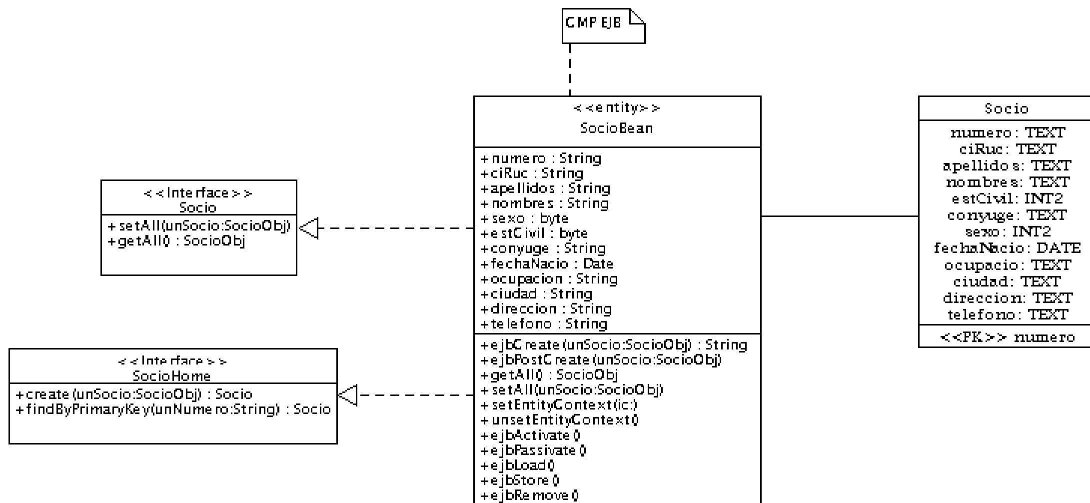

Figura 5.4: El componente Socio

La Figura 5.5 muestra el componente Cuenta. Su propósito es permitir que los clientes puedan crear, actualizar y eliminar instancias de la entidad *Cuenta*. La interfaz local *CuentaHome* proporciona los métodos de ciclo de vida. La interfaz remota *Cuenta* especifica los métodos para actualización.

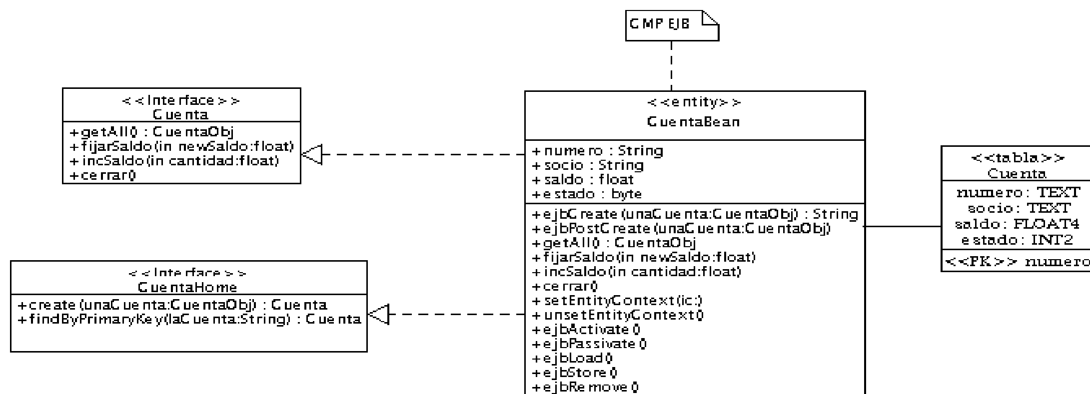

Figura 5.5: El componente Cuenta

La Figura 5.6 muestra el componente Transacción encargado de implementar el registro y consulta de las transacciones que va generando el Cajero durante su ciclo de trabajo.

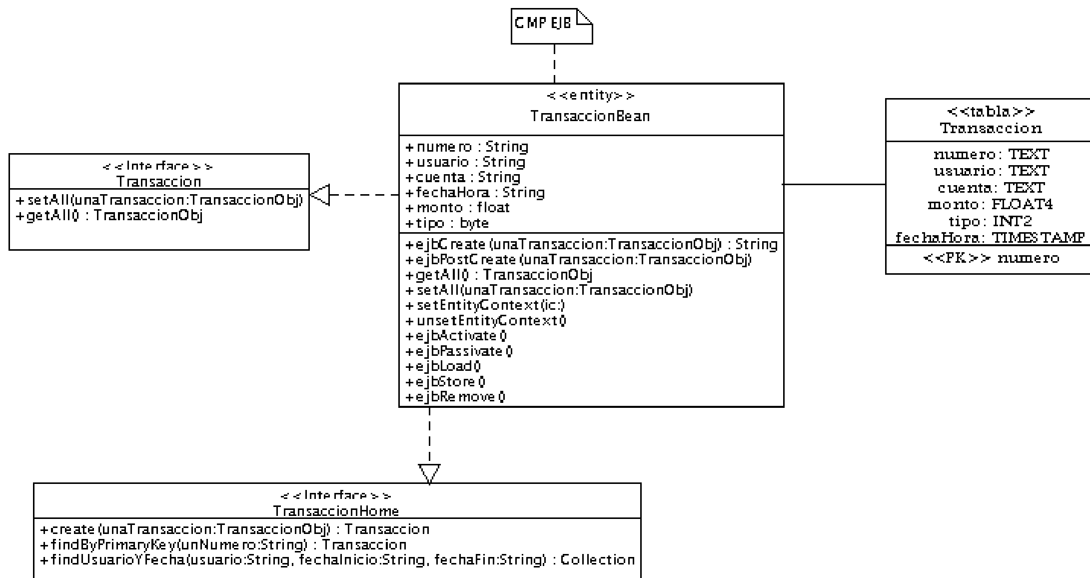

Figura 5.6: El componente Transacción

La Figura 5.7 muestra el componente Semilla que permite llevar un registro de los números de socio, cuenta y transacción que utiliza el sistema al momento de crear instancias.

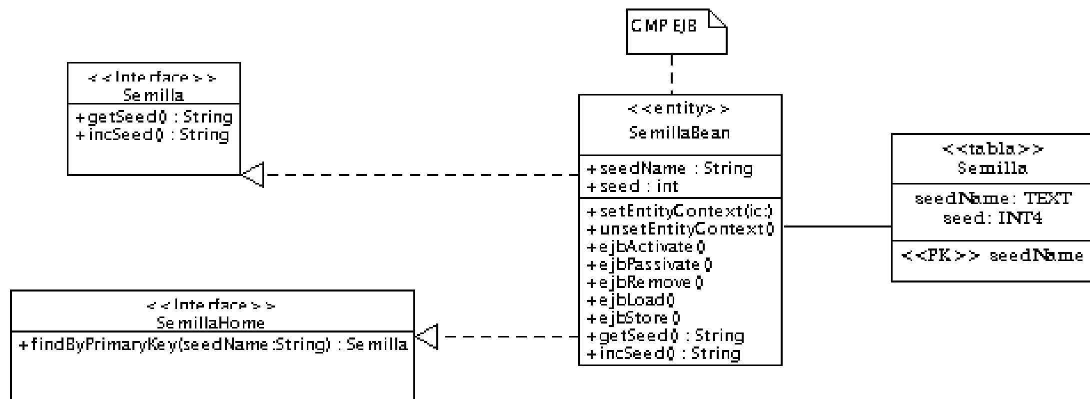

Figura 5.7: El componente Semilla

5.3.3.2 Clases del módulo Cliente

La Figura 5.8 muestra las principales clases del módulo Cliente y cómo se han de

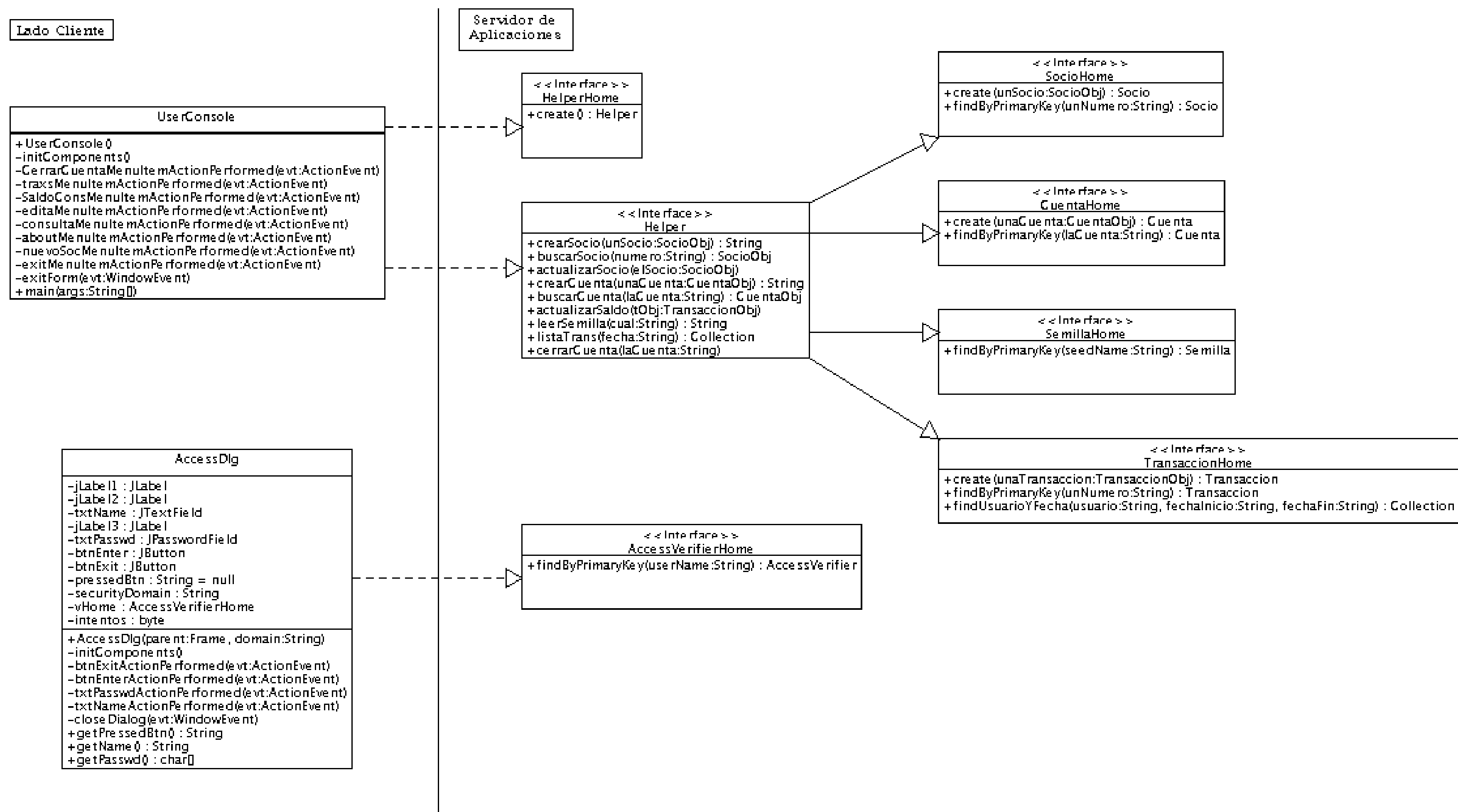

Figura 5.8: Las clases del módulo Cliente y su interacción con el Servidor

comunicar con las interfaces del Servidor para hacer uso de sus servicios.

5.3.3.3 Clases del Módulo Administración

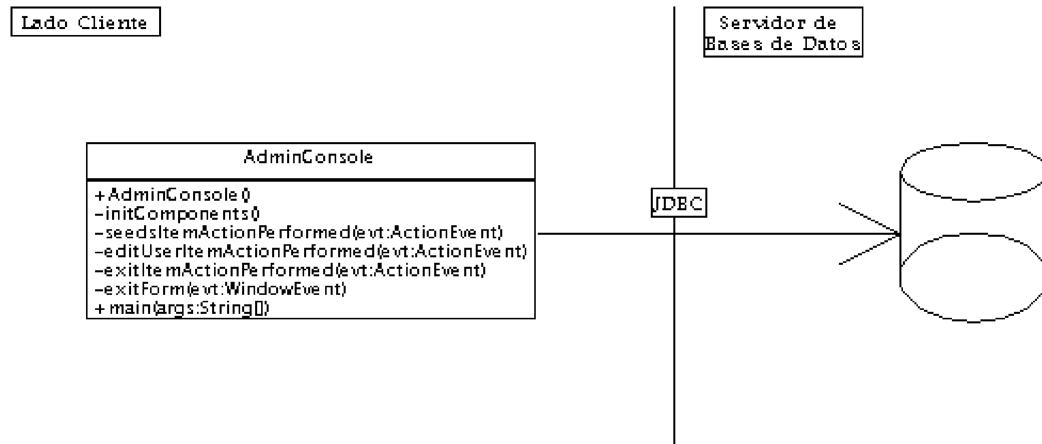

Figura 5.9: La clase principal del módulo Administración

5.3.4 Especificación del comportamiento de los módulos

Los siguientes diagramas de secuencia en combinación con los contratos[8] especifican el comportamiento del sistema según se ha identificado durante el análisis.

5.3.4.1 Caso de uso: Abrir cuenta

Figura 5.10: Diagrama de secuencia del sistema para el caso de uso *Abrir cuenta*

[8] Los diagramas de secuencia se preparan siguiendo el curso normal de los eventos de cada caso de uso. Los contratos describen las operaciones indicadas en los diagramas de secuencia.

5.3.4.1.1 Contratos

Nombre del Contrato: obtenerNumeroSocio

Responsabilidades: Realiza una búsqueda en la base de datos para devolver el siguiente número de Socio disponible.

Tipo: Sistema

Precondiciones:

- Ya están disponibles las tablas del Sistema Financiero dentro de una base de datos.
- Ya están asignados los valores iniciales de las semillas[9].

Nombre del Contrato: crearSocio(datosSocio)

Responsabilidades: Toma todos los datos del socio y crea una nueva instancia de socio con ellos dentro de la base de datos.

Tipo: Sistema

Excepciones: Si el número de Cédula de Identidad no es válido, indicar que se cometió un error.

Salida: Una solicitud de apertura de cuenta.

Precondiciones:

- Ya están disponibles las tablas del Sistema Financiero dentro de una base de datos.
- Ya están asignados los valores iniciales de las semillas.

Poscondiciones:

- Se creó una instancia de *Socio.*
- Se aumentó en uno la semilla de *Número de Socio.*
- Se notificó del número de socio recién asignado al Cajero

<u>Nombre del Contrato</u>: crearCuenta(numSocio)

Responsabilidades: Crea una nueva instancia de cuenta: es decir, abre una nueva libreta de ahorros.

Tipo: Sistema

Excepciones: Si el número de socio indicado no existe se indica un error.

Precondiciones:

- Ya están disponibles las tablas del Sistema Financiero dentro de una base de datos.
- El número de Socio indicado representa un Socio ya registrado.

Poscondiciones:

- Se creó una nueva instancia de *Cuenta* para el *Socio* indicado.
- Se incrementó en uno la semilla de *Número de Cuenta*.
- Se notificó del número de cuenta recién creado.

5.3.4.2 Caso de uso: Depositar dinero y Retirar dinero

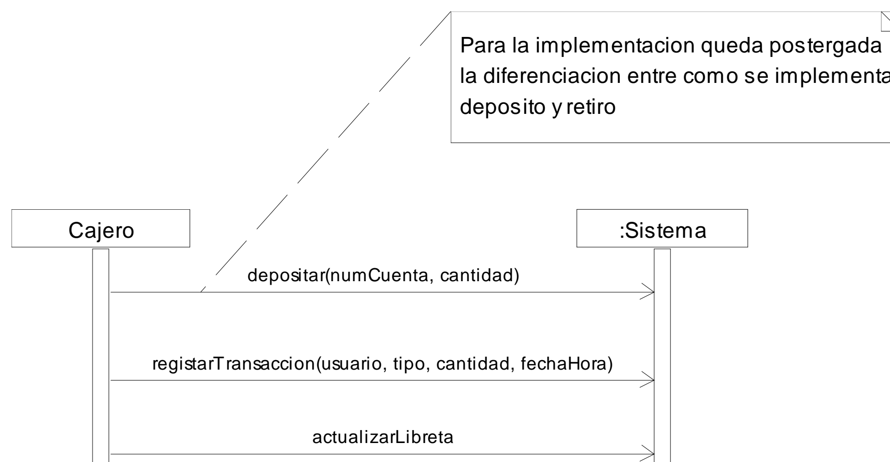

Figura 5.11: Diagrama de secuencia del sistema para los caso de uso *Depositar dinero* y *Retirar dinero*

---

[9] Semilla: Dentro del vocabulario del Sistema Financiero, Semilla es un valor numérico usado como un contador para número de socio, número de cuenta y número de transacción.

5.3.4.2.1 Contratos

Nombre del Contrato: depositar (numCuenta, cantidad)

Responsabilidades: Realiza la actualización de la cuenta del socio de acuerdo a lo que se haya solicitado; esto es, para el caso de depósito y retiro.

Tipo: Sistema

Excepciones:

- Si la *cantidad* especificada es negativa se notifica el error.
- Si la *cantidad* solicitada para retirar es mayor al saldo disponible se notifica el error.

Precondiciones:

- Ya existe una cuenta dentro del sistema.

Poscondiciones:

- Si se trata de un depósito de dinero se realiza el aumento en el saldo de la cuenta *numCuenta* por el monto de dinero establecido en *cantidad*.
- Si se trata de un retiro de dinero se realiza el descuento en el saldo de la cuenta *numCuenta* por el monto de dinero establecido en *cantidad*.

Nombre del Contrato: registrarTransaccion(usuario, tipo: (*depósito, retiro*), cantidad, fechaHora)

Responsabilidades: Realiza el registro (en la tabla correspondiente de la base de datos) de la transacción recién realizada.

Tipo: Sistema

Precondiciones:

- El *usuario* en nombre del cual se registra la transacción ha sido autentificado.
- La operación *depositar* se completó exitosamente.

Poscondiciones:

- Se creó una instancia de *Transacción* con todos los datos especificados en los parámetros.
- Se incrementó en uno el valor de la semilla para *Número de transacción*.

5.4 Resumen del diseño orientado a objetos

Con los artefactos producidos durante el análisis (los casos de uso, el modelo conceptual, los diagramas de secuencia y los contratos) avanza el desarrollo del software hacia la fase de diseño enfocada en la descripción del comportamiento del sistema considerando una implementación real. Los artefactos de ésta fase son los diagramas de interacción y la descripción de la interfaz con el usuario.

5.4.1 Diagramas de interacción

Un diagrama de interacción explica gráficamente las interacciones existentes entre las instancias del modelo conceptual. El punto de partida de las interacciones es el cumplimiento de las poscondiciones de los contratos de operación.

El UML define dos tipos de estos diagramas: los diagramas de secuencia y los diagramas de colaboración. En éste documento se utilizan los diagramas de colaboración por su excepcional expresividad y su capacidad de comunicar más información en menos espacio.

Los diagramas de colaboración que siguen a continuación se han creado basándose en los diagramas de secuencia y describen cada operación del sistema allí especificada.

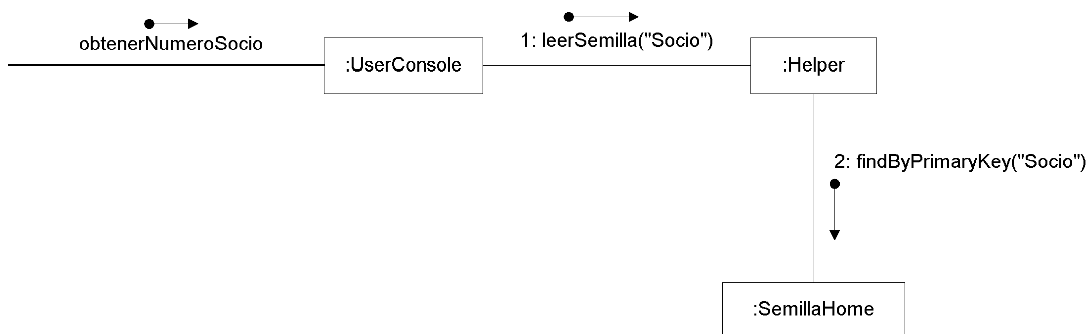

Figura 5.12: Diagrama de colaboración para obtenerNumeroSocio

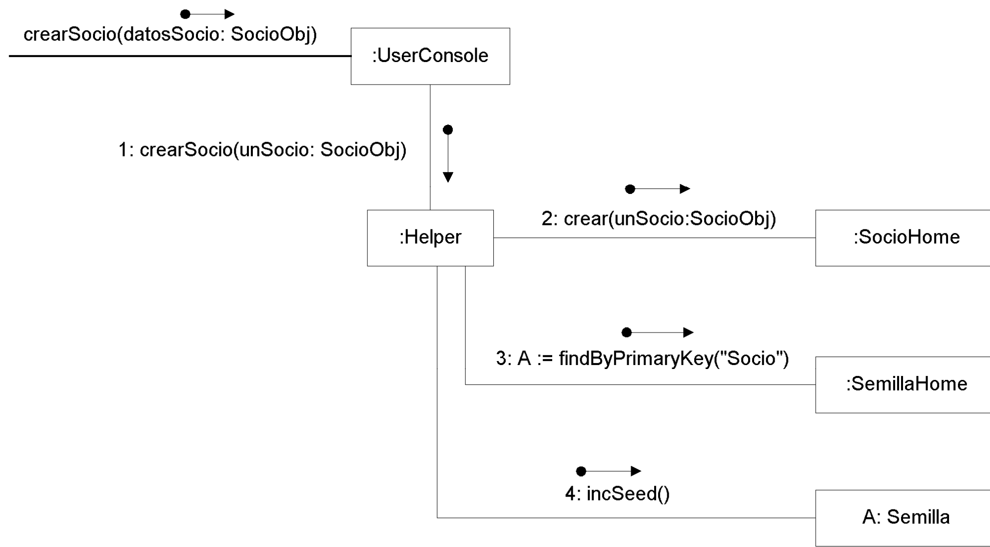

Figura 5.13: Diagrama de colaboración para crearSocio

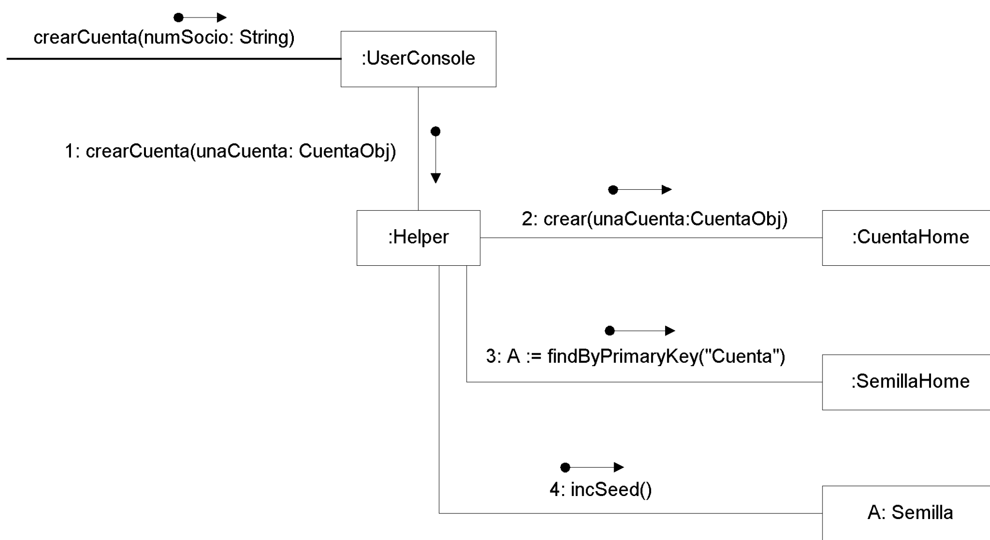

Figura 5.14: Diagrama de colaboración para crearCuenta

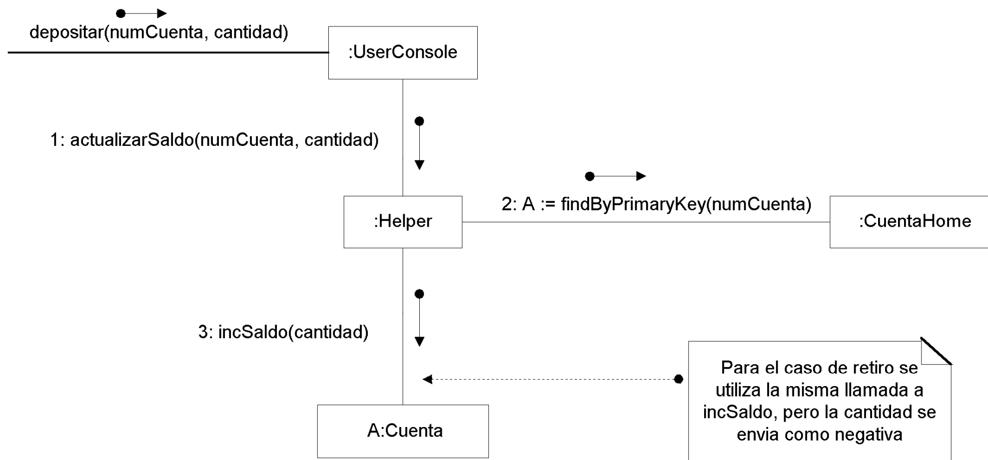

Figura 5.15: Diagrama de colaboración para depositar

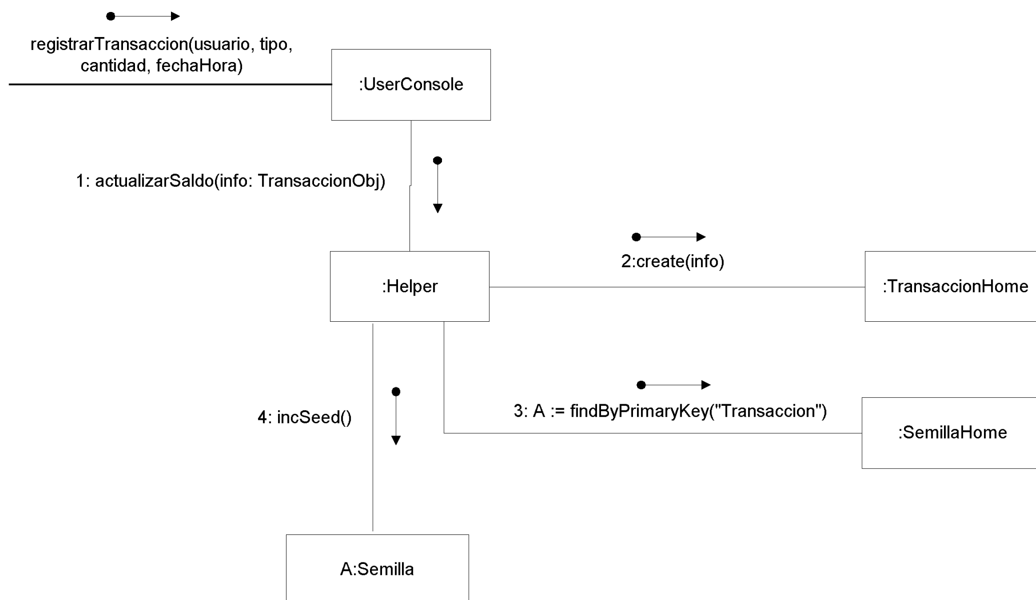

Figura 5.16: Diagrama de colaboración para registrarTransaccion

5.4.2 Componente de interacción humana

Dentro del diseño del componente de interacción humana (CIH) se realizaron varias tareas cuyos resultados se describen a continuación.

5.4.2.1 Jerarquía de ordenes

Usando la información recopilada en fases anteriores se ha determinado la siguiente jerarquía de ordenes.

Para el módulo Cliente tenemos:

Tabla 5.1: Menú de ordenes para la aplicación Cliente

Socio	Cuenta	Transacciones	Reportes	Ayuda
Nuevo socio y cuenta...	Consulta de saldo...	Transacciones de caja...	Lista clientes	Contenido
Consultar socio...	Cerrar cuenta...		Estado de cuenta...	Acerca de...
Editar socio...				
Salir				

Para el módulo Administración tenemos:

Tabla 5.2: Menú de ordenes para la aplicación de Administración

Archivo	Datos
Salir	Roles y usuarios...
	Valores iniciales...

5.4.2.2 Interacción detallada

En esta sección se muestra como deberá ser el diálogo entre el sistema y el usuario.

5.4.2.2.1 Módulo cliente

La Figura 5.17 muestra la ventana de acceso al módulo cliente.

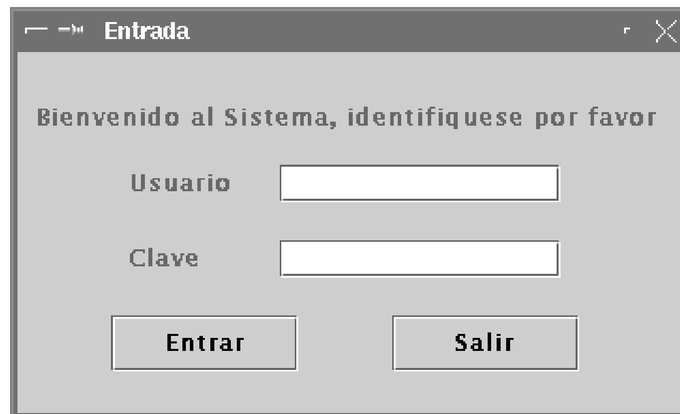

Figura 5.17: Ventana de acceso al módulo cliente

La Figura 5.18 indica como debe lucir la ventana donde se ingresan los datos de un nuevo socio. Puede ser reutilizada para edición y consulta.

Figura 5.18: Ventana para ingreso de un nuevo socio

La Figura 5.19 muestra como se debe presentar la ventana para consulta de saldo.

Figura 5.19: Ventana para la consulta del saldo de una cuenta

La Figura 5.20 muestra como debe aparecer la lista de las transacciones según las va registrando el Cajero.

Transacciones de Caja

FECHA	HORA	CUENTA	MONTO	TIPO
2002-1-16	10:40:10	107	$200.00	DEPOSITO
2002-1-16	10:40:40	100	$200.00	DEPOSITO
2002-1-16	10:40:49	101	$51.00	DEPOSITO
2002-1-16	10:40:58	105	$95.00	DEPOSITO
2002-1-16	10:41:44	107	$40.00	RETIRO
2002-1-16	10:42:32	104	$250.00	DEPOSITO
2002-1-16	10:42:44	104	$25.00	RETIRO

Figura 5.20: Ventana para la lista de las transacciones del Cajero

La Figura 5.21 presenta como será la ventana donde el Cajero registrará una transacción de depósito.

Deposito de ahorros en efectivo
Cuenta 107 0001823651458
GOMEZ BAZANTES
LUCIA ISABEL
Monto 200
Depositar Cerrar

Figura 5.21: Ventana para registrar la transacción de depósito

La Figura 5.22 muestra como será la ventana donde se registra la transacción de retiro.

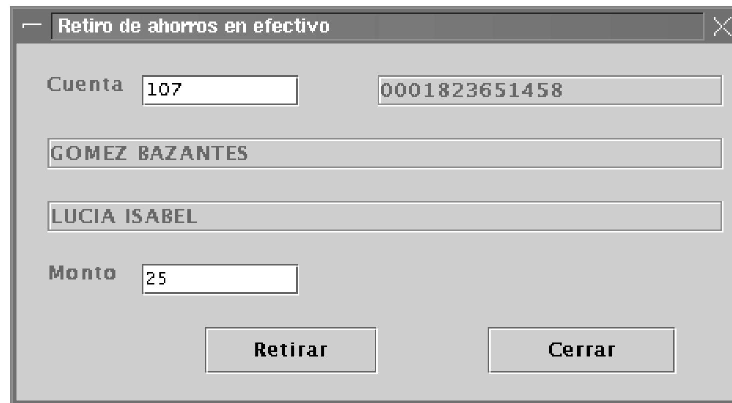

Figura 5.22: Ventana para el registro de una transacción de retiro

5.4.2.2.2 Módulo de administración

La Figura 5.23 muestra la ventana donde se especifican los parámetros de conexión a la base de datos usada por el módulo de administración.

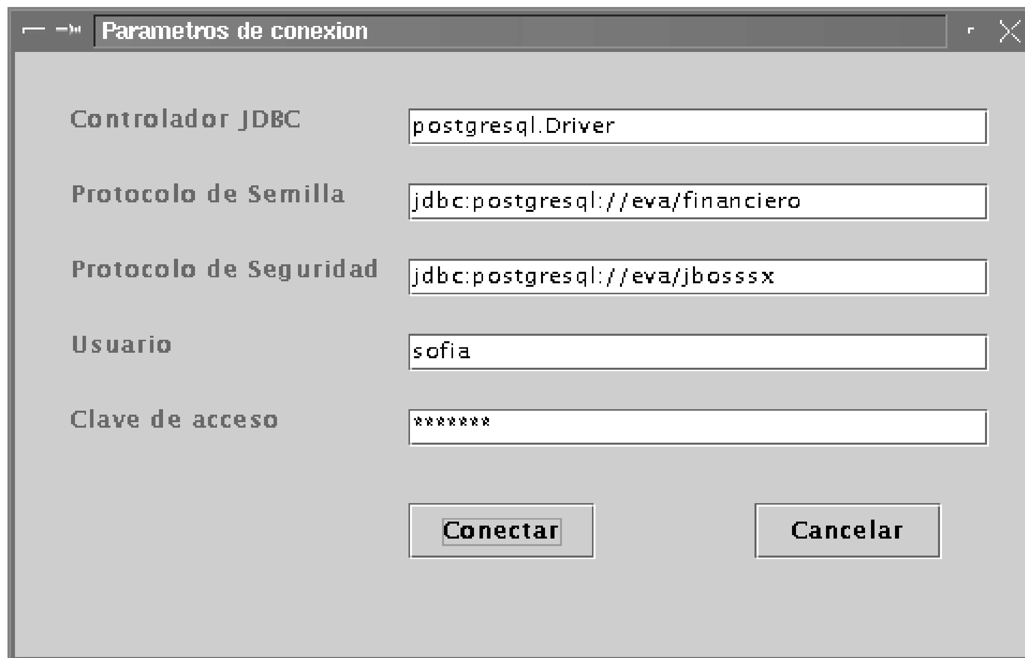

Figura 5.23: Ventana para especificar los parámetros de conexión del módulo de administración

La ventana consiste de los siguientes campos:

- Controlador JDBC.- Se utiliza para especificar el nombre de la clase Java que implementa el controlador JDBC a utilizar durante la sesión que se está iniciando.
- Protocolo de Semilla.- Se utiliza para indicar la dirección completa del protocolo JDBC a utilizar para conectarse a la base de datos que contiene la información de la tabla Semilla.
- Protocolo de Seguridad.- Se utiliza para indicar la dirección completa del protocolo JDBC a utilizar para conectarse a la base de datos que contiene la información de la tabla de permisos.

La Figura 5.24 muestra la ventana desde donde se administra la información de los roles del sistema y los usuarios asignados a cada rol.

Figura 5.24: Ventana para administración de roles y usuarios

La Figura 5.25 muestra la ventana donde se registra un nuevo rol. También se puede utilizar para edición.

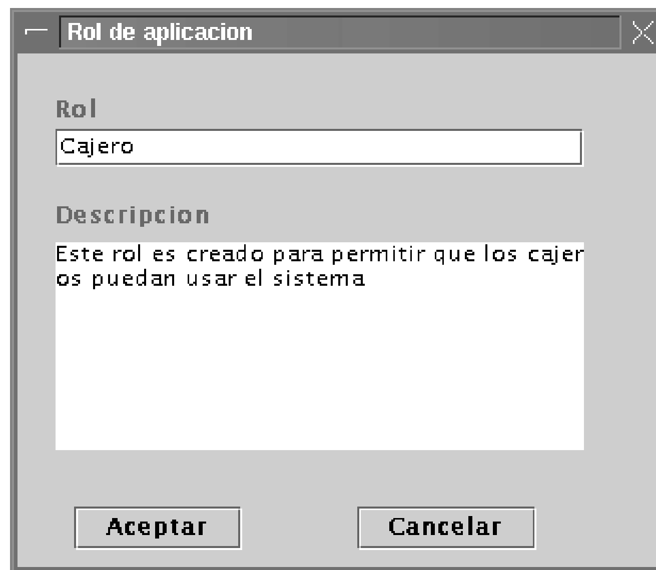

Figura 5.25: Ventana para registro de roles de la aplicación

La Figura 5.26 muestra la ventana donde se registra la información de los usuarios del sistema.

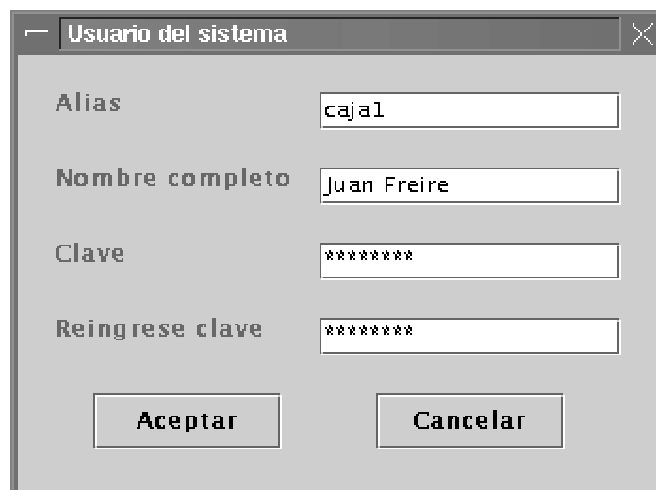

Figura 5.26: Ventana para edición de usuarios del sistema

La Figura 5.27 muestra la ventana donde se registran los valores iniciales (semillas) para los contadores del sistema.

Figura 5.27: Ventana para registro de valores iniciales

Cada una de las ventanas de las figuras anteriores se traduce en una clase específica del diseño de Componente de Interacción Humana.

5.5 Resumen de la implementación

Finalmente, con los artefactos producidos durante el análisis y el diseño, se procede a la implementación en el lenguaje escogido, que en éste caso es Java. Las secciones que se muestran a continuación documentan los aspectos más relevantes de ella.

5.5.1 Distribución de las clases de la implementación en paquetes

Las clases que implementan el comportamiento del sistema se organizan dentro de paquetes con el propósito de facilitar su manipulación, acomodarlas dentro de grupos relacionados y promover su futura reutilización. La Tabla 5.3 muestra los paquetes Java que se han definido y su propósito.

Tabla 5.3: Los paquetes Java que implementan el Sistema Financiero

Paquete	Descripción
edu.pucesa.financiero.admin	Paquete principal del módulo de Administración.
edu.pucesa.financiero.admin.gui	Contiene todas las clases que implementan las ventanas

Paquete	Descripción
	del módulo de Administración.
edu.pucesa.financiero.admin.utils	Contiene clases que el módulo de Administración utiliza para accesar a bases de datos.
edu.pucesa.financiero.beans	Contiene todas las clases que implementan los componentes empresariales del Sistema Financiero. Pertenece al módulo Servidor.
edu.pucesa.financiero.cliente	Paquete principal del módulo Cliente.
edu.pucesa.financiero.cliente.gui	Contiene todas las clases que implementan las ventanas del módulo Cliente.
edu.pucesa.financiero.cliente.utils	Contiene clases de propósito general que utiliza el módulo Cliente.
edu.pucesa.financiero.data	Define las clases que se utilizan como *Objeto Valor* para la comunicación entre cliente y servidor.
edu.pucesa.financiero.interfaces	Define todas las interfaces (locales y remotas) de los componentes empresariales del Sistema Financiero.
edu.pucesa.financiero.utils	Contiene clases de propósito general utilizadas por el módulo Servidor.

5.5.2 Resumen de clases por paquete

Cada paquete contiene una o más clases para implementar la funcionalidad del sistema. Las tablas que siguen a continuación muestran la descripción de cada clase para los paquetes ya presentados. El Anexo A contiene el detalle de cada clase.

Tabla 5.4: Clases del paquete edu.pucesa.financiero.admin

Clase	Descripción

Clase	Descripción
AdminConsole	Implementa la consola para la Administración del Sistema Financiero.

Tabla 5.5: Clases del paquete edu.pucesa.financiero.admin.gui

Clase	Descripción
CntxParmsDlg	Esta clase implementa la ventana de la Figura 5.23. Se usa para especificar los parámetros de conexión con las bases de datos que tienen la tabla Semillas y la tabla Roles.
EditaRolesDlg	Esta clase implementa la ventana de la Figura 5.24. Se usa para manejar la asignación de roles y usuarios.
RoleDlg	Esta clase implementa la ventana de la Figura 5.25. Edita la información de cada rol de la aplicación.
SeedsDlg	Esta clase implementa la ventana de la Figura 5.27. Edita los valores iniciales para números de socio, de cuenta y de transacción.
UserEditDlg	Esta clase implementa la ventana de la Figura 5.26. Se usa para editar la información de los usuarios.

Tabla 5.6: Clases del paquete edu.pucesa.financiero.admin.utils

Clase	Descripción
ConnectionMngr	Esta clase sirve como acceso central a las conexiones que necesita la consola de administración para manipular la base de datos. *Tiene una fuerte dependencia de los nombres de*

Clase	Descripción
	campos y tablas en la base de datos.

Tabla 5.7: Clases del paquete edu.pucesa.financiero.beans

Clase	Descripción
AccessVerifierBean	Esta clase implementa el componente entidad AccessVerifier. Su propósito es permitir autentificar a un usuario a través de una clave.
CuentaBean	Esta clase implementa el componente entidad Cuenta. Proporciona los servicios necesarios para manipular las cuentas de ahorro.
HelperBean	Esta clase se creó para actuar como un centro de despacho de la interacción entre el cliente y los componentes del lado del servidor. Específicamente, es el único componente con el que interactúa directamente el cliente.
SemillaBean	Esta clase implementa el componente entidad SemillaBean. Este componente tiene dos propósitos muy específicos: (1) leer la semilla para crear un nuevo socio o una nueva cuenta y (2) leer la semilla e incrementarla en uno para que quede actualizada.
SocioBean	Esta clase implementa el componente entidad Socio. Proporciona los servicios necesarios para manipular la información de los socios.
TransaccionBean	Esta clase implementa el componente entidad Transaccion. Proporciona los servicios necesarios para manipular el registro de transacciones.

Tabla 5.8: Clases del paquete edu.pucesa.financiero.cliente

Clase	Descripción
UserConsole	Es la clase principal para el módulo Cliente.

Tabla 5.9: Clases del paquete edu.pucesa.financiero.cliente.gui

Clase	Descripción
AccessDlg	Esta clase implementa la ventana de la Figura 5.17. Controla el paso de los usuarios al sistema.
BorraCtaDlg	Esta clase implementa la operación de eliminación de una cuenta de ahorros.
ConsSocioDlg	Esta clase implementa una ventana donde se pueden consultar los datos de un socio.
EditaSocioDlg	Esta clase se encarga de la interfaz para editar los datos de un socio.
SaldoCons	Esta clase implementa la ventana de la Figura 5.19. Permite consultar el saldo de una cuenta de ahorros.
SocioDlg	Esta clase implementa la ventana de la Figura 5.18. Se encarga de la interfaz para ingresar nuevos socios al Sistema Financiero.
TransDlg	Esta clase implementa una ventana como la de la Figura 5.21. Representa una caja genérica a partir de la cual se ajustan cajas específicas para depósito y retiro en efectivo.
TxsListDlg	Esta clase implementa la ventana de la Figura 5.20. Consiste

Clase	Descripción
	en una caja de diálogo donde se muestran las transacciones según se van realizando en Caja.

Tabla 5.10: Interfaces del paquete edu.pucesa.financiero.cliente.utils

Interfaz	Descripción
ClientJNDINames	Se utiliza para almacenar los nombres JNDI de los componentes que usa el cliente.

Tabla 5.11: Clases del paquete edu.pucesa.financiero.cliente.utils

Clase	Descripción
TxsTableModel	Esta clase define el modelo de datos que se muestra durante el registro de transacciones.

Tabla 5.12: Clases del paquete edu.pucesa.financiero.data

Clase	Descripción
CuentaObj	Esta clase se utiliza como un *Objeto Valor* para pasar como un solo bloque toda la información de la cuenta.
SocioObj	Esta clase se utiliza como un *Objeto Valor* para pasar como un solo bloque toda la información del socio.
TransaccionObj	Esta clase se utiliza como un *Objeto Valor* para pasar como un solo bloque toda la información de la transacción.

Tabla 5.13: Interfaces del paquete edu.pucesa.financiero.interfaces

Interfaz	Descripción
AccessVerifier	Esta es la interfaz remota para el componente entidad AccessVerifier.
AccessVerifierHome	Esta es la interfaz local para el componente entidad AccessVerifier.
Cuenta	Esta define la interfaz remota para el componente Cuenta.
CuentaHome	Esta define la interfaz local del componente Cuenta.
Helper	Esta define la interfaz remota para el componente Helper.
HelperHome	Esta define la interfaz local del componente Helper.
Semilla	Esta es la interfaz remota del componente entidad Semilla.
SemillaHome	Esta es la interfaz local para el componente entidad Semilla.
Socio	Esta define la interfaz remota para el componente Socio.
SocioHome	Esta define la interfaz local del componente Socio.
Transaccion	Esta define la interfaz remota para el componente Transacción.
TransaccionHome	Esta define la interfaz local del componente Transacción.

Tabla 5.14: Interfaces del paquete edu.pucesa.financiero.utils

Interfaz	Descripción
DBNames	Esta interfaz guarda los nombres de tablas y de claves usados dentro de los componentes.

Interfaz	Descripción
JNDINames	Esta interfaz guarda los nombres JNDI usados desde dentro de los componentes.
TiposTrans	Esta interfaz guarda los tipos de transacciones usados en la aplicación.

Conclusiones

1. Una aplicación multicapa es aquella que ha sido dividida en varios componentes, posiblemente distribuidos entre múltiples máquinas.
2. Una arquitectura de múltiples capas proporciona un número significativo de ventajas sobre arquitecturas tradicionales cliente/servidor, incluyendo mejoras en la capacidad de crecimiento, desempeño, confiabilidad, administración, reutilización, y flexibilidad.
3. Las arquitecturas cliente/servidor tradicionales aún son empleadas donde, luego de un análisis, se determina que es más conveniente un enfoque liviano. Por ejemplo, en el caso de consultas a un servidor de bases de datos desde un cliente en la misma red.
4. El lenguaje de programación Java posee una clara ventaja sobre otros, al brindar la capacidad de escribir una sola vez el programa y ejecutarlo en cualquier máquina que tenga una Máquina Virtual Java, sin tener que volver a compilar.
5. El eje principal de la plataforma Java 2, Edición Empresarial es un conjunto de tecnologías de componentes (Componentes Empresariales Java, Páginas Java de Servidor y Servlets) que simplifican el proceso de desarrollo de aplicaciones empresariales.
6. La plataforma J2EE es completamente funcional en el sentido de que es posible desarrollar grandes aplicaciones de nivel empresarial usando solo las tecnologías J2EE. La plataforma proporciona un número de beneficios para las empresas incluyendo un modelo simplificado de desarrollo, escalabilidad de nivel industrial, apoyo para los sistemas de información existentes, opciones de servidores, herramientas, y componentes, y un modelo de seguridad sencillo y flexible.
7. Un Componente Empresarial Java es una pieza de software que implementa una tarea o una entidad del negocio.
8. Un Componente Empresarial Java divide la lógica del negocio en tres partes: (1) una interfaz local (*Home*) que especifica los métodos de creación y/o búsqueda, (2) una interfaz remota (*Remote*) que especifica los métodos de la lógica del negocio, y (3) una clase del componente (*EJB*) que implementa el comportamiento señalado en las dos interfaces.

9. La Especificación de los Componentes Empresariales Java extiende la característica multiplataforma de Java al definir a un componente como independiente de una plataforma, protocolo, o infraestructura especifica.
10. El sistema operativo Linux y el conjunto de aplicaciones GNU hacen la combinación ideal para crear un entorno confiable, multiusuario y con los servicios de alta calidad que requieren las empresas de la era de Internet.
11. Los países en vías de desarrollo, como Ecuador, pueden colocarse a la par de los más avanzados gracias a que software para Linux está disponible a precio asequible y hasta en forma gratuita.
12. Existen muchas ventajas en disponer del código fuente de un programa, por ejemplo: (1) puede ser modificado y agregársele funcionalidad según se necesite, (2) la continuidad del programa está asegurada aun cuando el creador original cese su actividad, (3) ya que generalmente se desarrolla en el contexto de una comunidad, los errores se encuentran y corrigen con rapidez.
13. La utilización de Linux junto a herramientas de código fuente abierto y a la plataforma Java Empresarial hacen posible que organizaciones con bajo presupuesto puedan implementar un entorno seguro para sus aplicaciones empresariales.

Recomendaciones

1. El equipo de desarrollo debe seleccionar una metodología sólida, que especifique un proceso sistemático, con el cual todo el mundo se sienta cómodo y otorgue la libertad suficiente para hacer innovaciones o variaciones según se necesiten.
2. Se recomienda la utilización de estándares para el desarrollo de proyectos informáticos como el estándar IEEE 830 para la especificación de los requerimientos de software o la aplicación del estándar ISO 9000-3 para asegurar la calidad del desarrollo del software.
3. Es aconsejable que toda organización que desarrolle software posea, o cree, un Comité de Estandarización. Éste se encargará de supervisar la aplicación de los estándares elegidos, regular las propuestas de documentación y facilitar la comunicación entre las diferentes partes involucradas.
4. Java, en general, y Java Empresarial, en particular, poseen un gran número de APIs, cuyo número de clases e interfaces puede resultar simplemente abrumador. Es recomendable ir aprendiendo poco a poco cada API según se vaya necesitando en el proyecto.
5. Java Empresarial no es fácil de usar. Una organización que se decida por ésta plataforma deberá delinear una plan que involucre la capacitación de los programadores y revisión de procesos de desarrollo, por ejemplo.
6. La construcción y utilización de clases reutilizables es una meta que los desarrolladores siempre deben tener presente, aunque tampoco se ha de convertir en una obsesión.
7. Cuando se está modelando un sistema no se debe temer en recurrir a todo tipo de fuentes de información, tanto dentro como fuera de la organización, con el propósito de optimizar el modelo.
8. Cuando se desarrollan sistemas, y especialmente en aquellos de nivel empresarial, lo más aconsejable es realizar extensas pruebas antes de la entrega final. Las pruebas se dirigen a buscar fallas que solo podrían aparecer al colocar la aplicación en manos de usuarios reales con datos reales.

9. En vista de acelerado avance de la Tecnología de Información, quienes se dedican a desarrollar sistemas de software deben estar permanentemente investigando las ventajas y desventajas de los nuevos métodos, herramientas, modelos y estándares.
10. La Universidad, y específicamente la Escuela de Ingeniería de Sistemas, como parte de su responsabilidad de difundir los avances tecnológicos, debería tomar para sí la tarea de fomentar entre las organizaciones de Ambato el uso de herramientas de código abierto que son confiables y reducen el costo total de propiedad.

Bibliografía

BOOCH Grady. (1994). Object-oriented Analysis and Design with Applications. Segunda Ed. Redwood City. Ed. Benjamin/Cummings.

COAD P. y YOURDON E. (1991). Object-oriented Design. Englewood Cliffs. Ed. Prentice-Hall.

KASSEM Nicholas et al. (2000). Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition. Ed. Sun Microsystems Inc.

LARMAN Craig. (1999). UML y Patrones. Introducción al análisis y diseño orientado a objetos. México. Ed. Pearson.

PRESSMAN Roger S. (1999). Ingeniería del Software. Un enfoque práctico. Cuarta Ed. Madrid. Ed. McGraw-Hill.

RATIONAL SOFTWARE CORPORATION. (1998). Rational Unified Process. Best Practices for Software Development Teams. Cupertino.

SMITH Patrick. (1992). Client/server Computing. Carmel, Ind. Ed. SAMS.

SUN MICROSYSTEMS INC. (1999). Enterprise JavaBeans Specification, Version 1.1.

Sitios Web

http://java.sun.com

http://www.itworld.com

http://www.javaworld.com

http://www.onjava.com

http://www.oreillynet.com

Páginas Web

COMP.CLIENT-SERVER. (199x). Client-server FAQ. http://www.abs.net/~lloyd/csfaq.txt

COOK Rick. (2000). Why Linux is the platform of choice for many Java developers? LinuxWorld. http://www.linuxworld.com/lw-2000-03/lw-03-javalinux.html

GOULD Steven. (2000). Developing n-tier applications with J2EE. JavaWorld. http://www.javaworld.com/javaworld/topicalindex/jw-ti-ssj.html

HUSTEAD Robert. (2000). Mapping XML to Java. JavaWorld. http://www.javaworld.com/jw-08-2000/jw-0804-sax.html

KRUCHTEN Philippe. (2000). A Rational Development Process. http://www.rational.com/products/rup/prodinfo/whitepapers

MEYER Gary. (2000). Enterprise Java for Linux HOWTO. http://gary.meyer.net

MICROMAX INFORMATION SERVICES LTD. (1999). Client/Server and the n-tier model of distributed computing. http://www.n-tier.com/articles/csovervw.html

SCOTKIN Joel. (2000). Java in the Enterprise, Part 2. Beyond n-tier: what you really get from Java. JavaWorld. http://www.javaworld.com/javaworld/topicalindex/jw-ti-ssj.html

STEELE SCHARBACH ASSOCIATES L.L.C. (2000). Client-server architecture: Bringing order to the bramble bush. http://www.ssa-lawtech.com/wp/wp.htm

SUN MICROSYSTEMS, INC. (1997). The Java Enterprise Server Platform. A Java Adoption White Paper for Developers. Palo Alto, CA. http://www.sun.com/index.html

SUN MICROSYSTEMS, INC. (1998). Implementing a Multitier, Services-Based Architecture on the Java Platform at Sun -- A Case Study. http://www.sun.com/980602/wp/Exec_Overview.html

SUN MICROSYSTEMS, INC. (2000). Enterprise JavaBeans, Frequently Asked Questions. http://java.sun.com/products/ejb/faq.html

THOMAS Anne. (1998). Enterprise JavaBeans Technology. Server Component Model for the Java Platform. http://java.sun.com/products/ejb/docs.html

WELSH Matt. (1995). Linux Installation and Getting Started. Versión 2.2.2. http://sunsite.unc.edu/mdw/LDP/install-guide-2.2.2.html.tar.gz

Anexo A

En éste anexo se muestra la documentación de las clases Java para el Sistema Financiero. Ésta documentación se ha creado automáticamente a partir de los comentarios existentes dentro del código fuente de las clases usando la utilidad ***javadoc***. Por el volumen de información generado solo se han incluido una parte de ellas. Para examinar la documentación completa del sistema se ha de acudir al respaldo adjunto a ésta disertación.

Paquete edu.pucesa.financiero.admin

Clase AdminConsole

`public final class AdminConsole extends javax.swing.JFrame`

Esta clase implementa una consola para el Administrador del Sistema Financiero.

Resumen de campos	
private javax.swing.JMenu	**dataMenu**
private javax.swing.JMenuItem	**editUserItem**
private javax.swing.JMenuItem	**exitItem**
private javax.swing.JMenuBar	**menuAdmin**
private javax.swing.JMenuItem	**seedsItem**
private javax.swing.JMenu	**userMenu**

Constructor	
AdminConsole() Creates new form AdminConsole	

Resumen de métodos	
private void	**editUserItemActionPerformed**(java.awt.event.ActionEvent evt)
private void	**exitForm**(java.awt.event.WindowEvent evt)
private void	**exitItemActionPerformed**(java.awt.event.ActionEvent evt)
private void	**initComponents**()

Resumen de métodos

	This method is called from within the constructor to initialize the form.
static void	**main**(java.lang.String[] args) Ejecuta la consola de administración
private void	**seedsItemActionPerformed**(java.awt.event.ActionEvent evt)

Paquete edu.pucesa.financiero.admin.gui

Clase CntxParmsDlg

`public final class CntxParmsDlg extends javax.swing.JDialog`

Esta clase sirve para leer los parametros necesarios para establecer una conexion con las bases de datos que tienen las Semillas y los Roles.

Resumen de campos

private javax.swing.JButton	**btnCancel**
private javax.swing.JButton	**btnOk**
private boolean	**btnPressed**
private javax.swing.JLabel	**jLabel1**
private javax.swing.JLabel	**jLabel2**
private javax.swing.JLabel	**jLabel3**
private javax.swing.JLabel	**jLabel4**
private javax.swing.JLabel	**jLabel6**
private javax.swing.JTextField	**txtJdbcDriver**
private javax.swing.JPasswordField	**txtJdbcPasswd**
private javax.swing.JTextField	**txtJdbcURLRoles**
private javax.swing.JTextField	**txtJdbcURLSeed**
private javax.swing.JTextField	**txtJdbcUser**

Constructor

CntxParmsDlg(java.awt.Frame parent) Creates new form CntxParmsDlg	

Resumen de métodos	
private void	**btnCancelActionPerformed**(java.awt.event.ActionEvent evt)
private void	**btnOkActionPerformed**(java.awt.event.ActionEvent evt)
private void	**closeDialog**(java.awt.event.WindowEvent evt) Closes the dialog
boolean	**execute**() Este método sirve para ejecutar la caja de diálogo e informar a quien llama sobre el resultado de la ejecución.
java.lang.String	**getJdbcDriver**()
java.lang.String	**getJdbcPasswd**()
java.lang.String	**getJdbcURLRoles**()
java.lang.String	**getJdbcURLSeed**()
java.lang.String	**getJdbcUser**()
private void	**initComponents**() This method is called from within the constructor to initialize the form.

Clase EditaRolesDlg

`public final class` **`EditaRolesDlg`** `extends javax.swing.JDialog`

Esta clase para manejar la asignación de roles y usuarios .

Resumen de campos	
private javax.swing.JButton	**btnClose**
private javax.swing.JButton	**btnDelRol**
private javax.swing.JButton	**btnDelUser**
private javax.swing.JButton	**btnEditRol**
private javax.swing.JButton	**btnEditUser**
private javax.swing.JButton	**btnNewRol**
private javax.swing.JButton	**btnNewUser**
private javax.swing.JLabel	**jLabel1**
private javax.swing.JLabel	**jLabel2**
private javax.swing.JList	**lstRoles**

Resumen de campos	
private javax.swing.JList	**lstUsers**

Constructor
EditaRolesDlg(java.awt.Frame parent) Creates new form EditaRolesDlg

Resumen de métodos	
private void	**btnCloseActionPerformed**(java.awt.event.ActionEvent evt)
private void	**btnDelRolActionPerformed**(java.awt.event.ActionEvent evt)
private void	**btnDelUserActionPerformed**(java.awt.event.ActionEvent evt)
private void	**btnEditRolActionPerformed**(java.awt.event.ActionEvent evt)
private void	**btnEditUserActionPerformed**(java.awt.event.ActionEvent evt)
private void	**btnNewRolActionPerformed**(java.awt.event.ActionEvent evt)
private void	**btnNewUserActionPerformed**(java.awt.event.ActionEvent evt)
private void	**closeDialog**(java.awt.event.WindowEvent evt) Closes the dialog
private void	**initComponents**() This method is called from within the constructor to initialize the form.

Clase RoleDlg

`public final class RoleDlg extends javax.swing.JDialog`

Esta clase para mostrar una caja de diálogo donde se edita la información de cada rol de la aplicación.

Resumen de campos	
private javax.swing.JButton	**btnCancel**
private javax.swing.JButton	**btnOk**
private boolean	**btnPressed**
private javax.swing.JLabel	**jLabel1**

Resumen de campos	
private javax.swing.JLabel	jLabel2
private javax.swing.JTextArea	txtRolDesc
private javax.swing.JTextField	txtRolName

Constructor
RoleDlg(java.awt.Frame parent) Crea una nueva forma

Resumen de métodos	
private void	btnCancelActionPerformed(java.awt.event.ActionEvent evt)
private void	btnOkActionPerformed(java.awt.event.ActionEvent evt)
private void	closeDialog(java.awt.event.WindowEvent evt) Closes the dialog
boolean	getBtnPressed() Indica opción se tomó.
java.lang.String	getRoleDesc() Devuelve la descripción del rol
java.lang.String	getRoleName() Devuelve el nombre del rol
private void	initComponents() This method is called from within the constructor to initialize the form.
void	setRoleDesc(java.lang.String aRoleDesc) Fija lo que aparece como descripción del rol
void	setRoleName(java.lang.String aRoleName) Fija lo que aparece como nombre del rol

Clase SeedsDlg

`public final class SeedsDlg extends javax.swing.JDialog`

Esta clase permite editar los valores iniciales (conocidos como Semillas) para números de Socio, de Cuenta y de Transacción.

Resumen de campos	
private javax.swing.JButton	btnCancel
private javax.swing.JButton	btnOK
private boolean	btnPressed

Resumen de campos	
private javax.swing.JLabel	**jLabel1**
private javax.swing.JLabel	**jLabel2**
private javax.swing.JLabel	**jLabel3**
private int	**numCuenta**
private int	**numSocio**
private int	**numTrans**
private javax.swing.JTextField	**txtNumCuenta**
private javax.swing.JTextField	**txtNumSocio**
private javax.swing.JTextField	**txtNumTrans**

Constructor
SeedsDlg(java.awt.Frame parent) Crea una nueva forma SeedsDlg

Resumen de métodos	
private void	**btnCancelActionPerformed**(java.awt.event.ActionEvent evt)
private void	**btnOKActionPerformed**(java.awt.event.ActionEvent evt)
private void	**closeDialog**(java.awt.event.WindowEvent evt) Closes the dialog
boolean	**cuentaChanged**() Indica si el valor de la semilla se editó
boolean	**getBtnPressed**()
java.lang.String	**getNumCuenta**()
java.lang.String	**getNumSocio**()
java.lang.String	**getNumTrans**()
private void	**initComponents**() This method is called from within the constructor to initialize the form.
boolean	**socioChanged**() Indica si el valor de la semilla se editó
boolean	**transChanged**() Indica si el valor de la semilla se editó

Clase UserEditDlg

`public class` **`UserEditDlg`** `extends javax.swing.JDialog`

Esta clase para editar la información de los usuarios.

Resumen de campos	
private javax.swing.JButton	**btnCancel**
private javax.swing.JButton	**btnOk**
private boolean	**btnPressed**
private javax.swing.JLabel	**jLabel1**
private javax.swing.JLabel	**jLabel2**
private javax.swing.JLabel	**jLabel3**
private javax.swing.JLabel	**jLabel4**
private javax.swing.JTextField	**txtUserFullName**
private javax.swing.JTextField	**txtUserName**
private javax.swing.JPasswordField	**txtUserPasswd**
private javax.swing.JPasswordField	**txtUserRePasswd**

Constructor
UserEditDlg(java.awt.Frame parent) Creates new form UserEditDlg

Resumen de métodos	
private void	**btnCancelActionPerformed**(java.awt.event.ActionEvent evt)
private void	**btnOkActionPerformed**(java.awt.event.ActionEvent evt)
private void	**closeDialog**(java.awt.event.WindowEvent evt) Closes the dialog
boolean	**getBtnPressed**() Para obtener la respuesta del usuario
java.lang.String	**getUserFullName**()
java.lang.String	**getUserName**()
java.lang.String	**getUserPasswd**()

Resumen de métodos	
private void	**initComponents**() This method is called from within the constructor to initialize the form.
void	**setUserFullName**(java.lang.String userFullName)
void	**setUserName**(java.lang.String userID)
void	**setUserPasswd**(java.lang.String userPasswd)

Paquete edu.pucesa.financiero.admin.utils

Clase ConnectionMngr

`public final class ConnectionMngr extends java.lang.Object`

Esta clase sirve como un acceso central a las conexiones que necesita la consola de administración para manipular la base de datos. T*iene una fuerte dependencia de los nombres de campos y tablas en la base de datos.*

Resumen de campos	
private static java.sql.Connection	**RolesConnection**
private static java.sql.PreparedStatement	**rolesStm**
private static java.sql.Connection	**SeedConnection**
private static java.sql.PreparedStatement	**usersStm**

Constructor	
ConnectionMngr()	

Resumen de métodos	
static void	**closeConnection**() Cierra las conexiones disponibles
static void	**deleteUser**(java.lang.String userID) Elimina al usuario del registro del sistema
private static void	**doBatchCmds**(java.lang.String[] cmds)
static void	**doConnection**(java.lang.String jdbcDriver, java.lang.String jdbcURLSeed, java.lang.String jdbcURLRoles, java.lang.String jdbcUser,

Resumen de métodos	
	java.lang.String jdbcPasswd) Trata de crear las conexiones a las base de datos con los parametros
static java.lang.String	**findRoleDesc**(java.lang.String aRole) Busca la definición para el rol indicado
static java.util.Vector	**getRoles**() Recupera una lista de los roles definidos
static int	**getSeedValue**(java.lang.String seedKey) Obtiene el valor de la semilla indicada
static java.lang.String	**getUserFullName**(java.lang.String userID) Busca el nombre completo asignado al usuario. Devuelve un valor *nulo* cuando no se encuentra el nombre completo.
static java.lang.String	**getUserPasswd**(java.lang.String userID) Busca la clave asignada al usuario. Devuelve un valor *nulo* cuando no se encuentra la clave.
static java.util.Vector	**getUsers**(java.lang.String role) Recupera una lista de usuarios según el rol indicado.
static void	**insertRole**(java.lang.String aRoleName, java.lang.String aRoleDesc) Inserta un nuevo rol en la tabla respectiva
static void	**insertUser**(java.lang.String role, java.lang.String userName, java.lang.String fullName, java.lang.String passwd) Inserta todas las entradas necesarias para un nuevo usuario.
static void	**updateRole**(java.lang.String aRoleName, java.lang.String aRoleDesc) Para actualizar la descripción del rol
static void	**updateSeedValue**(java.lang.String seedKey, java.lang.String seed) Actualiza el valor de la semilla indicada
static void	**updateUser**(java.lang.String userID, java.lang.String newFullName, java.lang.String newPasswd) Realiza la operación de actualización de un usuario. Esto consiste en actualizar el nombre completo y/o cambiar su clave.

Paquete edu.pucesa.financiero.beans

Clase AccessVerifierBean

public class **AccessVerifierBean** extends java.lang.Object implements javax.ejb.EntityBean

Esta clase implementa el componente entidad AccessVerifier CMP. Su propósito es permitir autentificar a un usuario a través de su clave.

Resumen de campos	
java.lang.String	**password**
java.lang.String	**principalid**

Constructor	
AccessVerifierBean()	

Resumen de métodos	
boolean	**autenticar**(java.lang.String passwd)
void	**ejbActivate**()
void	**ejbLoad**()
void	**ejbPassivate**()
void	**ejbRemove**()
void	**ejbStore**()
void	**setEntityContext**(javax.ejb.EntityContext ic)
void	**unsetEntityContext**()

Clase CuentaBean

```
public class CuentaBean extends java.lang.Object implements
javax.ejb.EntityBean
```

Clase de implementación del Componente Cuenta. Es un CMP Entity Bean.

Resumen de campos	
byte	**estado**
java.lang.String	**numero**
float	**saldo**
java.lang.String	**socio**

Constructor	
CuentaBean()	

Resumen de métodos	
void	**cerrar**() Cierra una cuenta
void	**ejbActivate**()
java.lang.String	**ejbCreate**(CuentaObj unaCuenta)
void	**ejbLoad**()
void	**ejbPassivate**()
void	**ejbPostCreate**(CuentaObj unaCuenta)
void	**ejbRemove**()
void	**ejbStore**()
void	**fijarSaldo**(float newSaldo)
CuentaObj	**getAll**()
void	**incSaldo**(float cantidad)
void	**setEntityContext**(javax.ejb.EntityContext ic)
void	**unsetEntityContext**()

Clase HelperBean

```
public class HelperBean extends java.lang.Object implements
javax.ejb.SessionBean
```

Esta clase se crea para que actue como centro de despacho de la interacción entre el cliente y los componentes. STATEFUL SESSION BEAN.

Resumen de campos	
private javax.ejb.SessionContext	**helperCtx**
private javax.naming.InitialContext	**iCtx**

Constructor
HelperBean()   Constructor para creación de instancias por el contenedor

Resumen de métodos	
void	**actualizarSaldo**(TransaccionObj tObj) Para realizar depósitos y retiros
void	**actualizarSocio**(SocioObj elSocio) Para actualizar los datos de un socio. Primero localiza el socio y luego actualiza sus datos
CuentaObj	**buscarCuenta**(java.lang.String laCuenta) Para acceder al componente Cuenta y solicitar una búsqueda
SocioObj	**buscarSocio**(java.lang.String numero) Para acceder al EJB Socio y solicitar la búsqueda de un socio por su clave
void	**cerrarCuenta**(java.lang.String laCuenta) Para cerrar una cuenta
java.lang.String	**crearCuenta**(CuentaObj unaCuenta) Para crear una cuenta
java.lang.String	**crearSocio**(SocioObj unSocio) Recibe los datos del socio y solicita al EJB la creación de un nuevo socio con esos datos.
void	**ejbActivate**()
void	**ejbCreate**() Crea un nuevo objeto para control de datos
void	**ejbPassivate**()
void	**ejbRemove**()
private java.lang.Object	**interfazLocal**(java.lang.String beanName) Busca dentro del espacio JNDI la interfaz local solicitada
java.lang.String	**leerSemilla**(java.lang.String cual) Para acceder al componente Semilla y leer el próximo número disponible
java.util.Collection	**listaTrans**(java.lang.String fecha) Busca las transacciones del día
void	**setSessionContext**(javax.ejb.SessionContext ic)

Glosario

A

Applet. Pequeña aplicación escrita en el lenguaje Java que se ejecuta dentro de un navegador de Internet.

Actor. Es una entidad externa del sistema que de alguna manera participa en un caso de uso.

Administrador de recursos. Proporciona acceso a un conjunto de recursos compartidos.

Aplicación distribuida. Una aplicación constituida por componentes distintos corriendo en entornos de ejecución separados, usualmente sobre plataformas diferentes conectadas a una red. Las aplicaciones distribuidas pueden ser de dos capas (cliente-servidor), de tres capas (cliente-intermedio-servidor) o multicapa (cliente-varios intermedios-varios servidores).

Aplicación Web. Una aplicación escrita para la Internet, incluyendo a aquellas escritas con la tecnología Java como páginas Java de servidor y servlets.

Archivo EJB JAR. Un archivo JAR que contiene un módulo EJB.

Autenticación. El proceso por el cual una entidad prueba a otra que está actuando en nombre de una identidad específica. La plataforma J2EE requiere tres tipos de autenticación: básica, basada en formas y mutua.

B

BMP, Bean Managed Persistence. Es una forma de administrar la persistencia de un componente empresarial en la cual ésta se delega al propio componente.

Bytecodes. Código binario intermedio producido por un compilador Java.

C

Caso de uso. Descripción narrativa textual de la secuencia de eventos y acciones que ocurren cuando un usuario entabla un diálogo con un sistema durante un proceso significativo.

CGI, Common Gateway Interface. Es un estándar para conectar aplicaciones externas con servidores de información como por ejemplo servidores HTTP. Un documento HTML llano es estático; lo que significa, que su contenido no cambia. Un programa CGI, por el otro lado, es ejecutado en tiempo real, por lo que puede producir contenido dinámico.

CMP, Component Managed Persistence. Es una forma de administrar la persistencia de un componente empresarial en la cual ésta se delega al contenedor.

COM, Component Object Model. Es una forma en que los componentes de software se pueden comunicar entre si. Es un estándar binario y de red que permite que dos componentes cualesquiera se comuniquen sin importar donde estén.

Componente. Una unidad de software en el nivel de aplicación soportada por un contenedor. Los componentes son configurables en tiempo de publicación. La plataforma J2EE define cuatro tipos de componentes: componentes empresariales, componentes Web, applets y aplicaciones clientes.

Computación Cliente/Servidor. Es la técnica que se caracteriza por dividir el procesamiento de la información al menos en dos partes, una que actúa como cliente y otra como servidor.

Contenedor. Una entidad que proporciona administración del ciclo de vida, seguridad, publicación y servicios en tiempo de ejecución a componentes. Cada tipo de contenedor (EJB, Web, JSP, servlet, applet y aplicación cliente) proporciona servicios específicos.

Contrato. Define las responsabilidades y poscondiciones que se aplican al uso de una operación o método. También designa el conjunto de las condiciones relacionadas con una interfaz.

CORBA, Common Object Request Broquer Architecture. Es una arquitectura independiente del vendedor que usan los programas para trabajar juntos sobre una red.

D

DES. Norma de encriptación de datos desarrollada por el gobierno de los Estados Unidos.

Descriptor de publicación. Un archivo XML proporcionado con cada módulo y aplicación que describe cómo estos deberían ser publicados.

Diagrama de secuencia. Muestra gráficamente los eventos que fluyen desde los actores al sistema.

Dirección IP. Dirección de 32 bits asignada a cada computadora que participa en una red TCP/IP.

DOM, Document Object Model. Un árbol de objetos con interfaces para navegarlo y escribir una versión XML de él, según la especificación del World Wide Web Consortium.

DTD, Document Type Definition. La descripción de la estructura y propiedades de una clase de archivos XML.

E

EJB, Enterprise JavaBean. Un objeto que se coloca dentro de un servidor de aplicaciones para implementar la lógica del negocio.

F

FTP. Protocolo de alto nivel que sirve para transferir archivos de una máquina a otra.

Fuente de datos. Es, primordialmente, una base de datos junto a un controlador y una conexión de fondo común.

J

J2EE, Java 2 Enterprise Edition. Nombre que se le da al paquete estándar Java versión 2 dedicado al desarrollo de aplicaciones de nivel empresarial junto con todas las APIs asociadas.

J2SE, Java 2 Standard Edition. Nombre que se le da al paquete estándar Java versión 2 junto con todas las APIs asociadas.

JAR, Java archive. Un formato de archivo independiente de la plataforma que permite que múltiples archivos se agrupen dentro de uno solo.

JDBC. API estándar de Java para la conexión y acceso a sistemas manejadores de base de datos.

JIT, Just In Time. En el contexto de Java se refiere a un compilador justo a tiempo que toma los bytecodes Java y los convierte en código nativo que se ejecuta más rápido.

JVM, Java Virtual Machine. Proporciona un entorno donde se ejecuta el código Java independiente (bytecodes) de la máquina.

L

Lógica del negocio. El código que implementa la funcionalidad de una aplicación. Dentro del modelo de Componentes Empresariales, ésta lógica es implementada por los métodos de un componente empresarial.

N

NFS. Protocolo que utiliza el IP para permitir que un conjunto de computadoras coopere para accesar los sistemas de archivos de otras, como si éstas fueran locales.

O

Objeto Valor. Es un objeto serializable Java que puede ser pasado por valor al cliente. La petición que un cliente hace de un objeto valor puede ser satisfecha por el servidor de una mejor forma, pues envía dentro de un solo paquete información que de otra manera tendría que enviar de a uno.

ODBC, Open Database Connectivity. API Estándar de Microsoft utilizada para la conexión de acceso a sistemas manejadores de datos.

OMG, Object Management Group. Es una organización comercial internacional, no lucrativa, formada por cerca de 200 compañías. Su objetivo principal es maximizar la portabilidad, la reutilización y la interoperabilidad del software.

ORB, Object Request Broquer. Agente de solicitud de objetos. Su objetivo es garantizar que los objetos puedan actuar entre sí, sin importar la máquina o red a la que estén conectados.

P

Persistencia. El protocolo para transferir el estado de un componente entidad entre sus variables de instancia y una base de datos subyacente.

PLIP. Protocolo utilizado para envíos IP a través de una línea paralela.

PPP. Protocolo para enmarcar al IP cuando se envía a través de una línea serial.

Publicación. El proceso por el cual un software es instalado dentro de un entorno operativo.

R

RMI, Remote Method Interface. API estándar de Java que se utiliza para invocar métodos en objetos que están disponibles en otras máquinas de la red.

S

SAX, Simple API for XML. Un mecanismo conducido por eventos de acceso serial para manipular documentos XML.

Servlet. Pequeña aplicación escrita en el lenguaje Java que funciona dentro de un servidor Web.

SLIP. Protocolo utilizado para envíos IP a través de una línea serial.

SMTP. Protocolo estándar para transferir mensajes de correo electrónico de una máquina a otra.

SNMP. Protocolo estándar utilizado para monitorear computadoras, ruteadores y las redes a las que están conectados.

SSL, Secure Socket Layer. Un protocolo de seguridad que provee privacidad sobre Internet. El protocolo permite a aplicaciones cliente/servidor comunicarse en una manera que no pueda ser saboteada.

T

TCP/IP. Nombre oficial para el conjunto de protocolos TCP/IP.

TELNET. Protocolo estándar del TCP/IP para servicio de terminal remota.

Transacción. Una unidad atómica de trabajo que modifica datos.

U

UML, Unified Modeling Language. Desde 1997 es un estándar de OMG que consiste en un lenguaje gráfico para modelar y desarrollar sistemas de software. Proporciona apoyo para todas las fases del ciclo de desarrollo de software.

W

WWW, World Wide Web. Servicio de Internet que organiza la información por medio de hiperenlaces. Cada documento puede contener referencias a imágenes, sonido y otros documentos.

X

XML, Extensible Markup Language. Un lenguaje de marcas que permite definir las etiquetas necesaria para identificar el contenido, los datos y el texto en documentos XML. Difiere de HTML en que éste posee etiquetas dedicadas mayormente a la presentación y estilo. Generalmente un documento XML debe ser transformado para presentarse como HTML. Aunque las etiquetas se pueden definir durante la generación de un documento XML, una DTD puede ser utilizada para definir los elementos permitidos en un tipo específico de documento. Un documento se puede comparar contra las reglas de una DTD

para comprobar su validez y localizar elementos particulares. Los programas que procesan XML utilizan las APIs SAX o DOM. Los descriptores de publicación de J2EE utilizan XML.