UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
"ACATLÁN"

CONSTRUCCIÓN DE UN SISTEMA DE INFORMACIÓN UTILIZANDO EL PARADIGMA ORIENTADO A OBJETOS

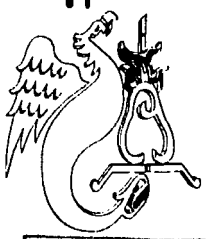
T E S I S

QUE PARA OBTENER EL TÍTULO DE

LICENCIADO EN MATEMÁTICAS APLICADAS Y COMPUTACIÓN

PRESENTA LA C. EDITH ADRIANA JIMENEZ CONTRERAS

DIRECTOR DE TESIS: ING. RUBÉN ROMERO RUIZ



MÉXICO, D.F., NOVIEMBRE 2002

A mi madre, María Esther

A mi abuelito, Pedro†

Contenido

Índice de figuras

INTRODUCCIÓN

El objetivo del presente trabajo es la construcción de un sistema de información distribuido[1] y orientado a objetos (OO) para la Línea de Estadística Aplicada y Diseño de Experimentos (LEADE) vinculada a la Investigación Básica de Procesos del Instituto Mexicano del Petróleo (IMP). El origen de este sistema de información surge durante mi estancia en el IMP como prestador de servicio social y de la necesidad de automatizar la información que LEADE maneja.

La información que se desea automatizar pertenece a los proyectos que LEADE realiza en las refinerías de PEMEX. Para llevar a cabo el proceso de construcción del sistema es necesario conocer los procedimientos que se efectúan en LEADE, así como toda la información relacionada con las herramientas que utilizan.

Para llegar al objetivo se tiene que realizar el análisis, diseño e implementación del sistema de información distribuido, soportado por herramientas que el Lenguaje de Modelado Unificado (UML) y la tecnología JAVA proporcionan.

[1] Un sistema distribuido es un conjunto de computadoras que se conectan a través de una red externa, para llevar a cabo la computación de una aplicación en común. A diferencia de los sistemas paralelos que se conforman por un conjunto de procesadores unidos por una red interna. Los sistemas distribuidos en general son utilizados para el cómputo basado en servicios, como por ejemplo sistemas de consulta o aplicaciones intensivas en cómputo de granularidad gruesa, los sistemas paralelos son utilizados para el cómputo intensivo en procesamiento, como por ejemplo la predicción del clima

La construcción de sistemas distribuidos es deseable para balancear la carga de trabajo en distintas máquinas que componen el sistema, para este sistema la aplicación cliente que serán los `applets`, la aplicación servidor que la conforman el TOMCAT (manejador de `servlets`) y las `servlets` y el motor de la base de datos que es ObjectStore, pueden separarse siguiendo un esquema de memoria distribuida por lo que se considera al sistema distribuido. Esto quiere decir que la aplicación servidor se puede ejecutar en una máquina distinta a donde se ejecutan los clientes, pudiendo hacer peticiones simultáneas al servidor, por lo que el servidor debe tener mecanismos para manejar la concurrencia. Es necesario mencionar que la base de datos no es distribuida, lo que implica que debe manejarse por una sola computadora. Es muy común encontrar que las aplicaciones sean centralizadas, es decir que la computación es llevada a cabo en una máquina, aún cuando existen aplicaciones en internet que basan sus clientes en páginas HTML, la computación se considera centralizada ya que la comprobación de los datos la sigue manteniendo el servidor, aumentando la carga de trabajo del servidor al ser una página HTML un cliente ligero.

El Lenguaje de Modelado Unificado o UML (Unified Modeling Language) es el sucesor de la oleada de métodos de análisis y diseño orientados a objetos (AOO & D) que surgió a finales de la década de los 80's y principios de la siguiente. Se puede definir a UML como un lenguaje de modelado, para especificar, visualizar, construir y documentar las diferentes etapas del desarrollo de un sistema de software orientado a objetos y procesos del negocio.

En la actualidad, gran parte del desarrollo de los sistemas de información tienen soporte en las bases de datos relacionales (BDR). Las bases de datos relacionales aparecen en los 70's y su base es el modelo relacional, por lo que su uso y conocimiento es bastante extenso, tal situación a conducido a que los programadores tengan preferencia y experiencia sobre este modelo. Sin embargo, existen otros modelos como el orientado a objetos, el cual proporciona nuevas características sobre el modelo relacional. Las ventajas que se pueden obtener del modelo orientado a objetos (MOO) son extensas pero podemos resaltar que se basan principalmente en no requerir tablas para

representar el modelo de datos, y la jerarquía de clases que surge de aplicar el modelo orientado a objetos representa el modelo de datos en conjunto con la lógica del negocio.

Las bases de datos orientadas a objetos (BDOO) soportan todas las características que el modelo orientado a objetos proporciona, que son principalmente: encapsulación, herencia y polimorfismo, por consecuencia tienen un mayor grado de expresividad que las bases de datos relacionales. Por tal motivo, se utiliza la metodología orientada a objetos para construir el sistema de información distribuido. Como manejador de bases de datos orientado a objetos (MBDOO) se usó Object Store PSE Pro, ya que este manejador es totalmente *orientado* a objetos y transportable a diferencia de otros manejadores más difundidos como son: Oracle e Informix que sólo soportan algunas características *basadas* en objetos.

La red internet surge en 1995 como la gran red mundial de comunicación entre computadoras, esta red ha sido aprovechada en la industria, la investigación, las instituciones, entre otros. Se utiliza la red internet como medio de comunicación del sistema para manejar a distancia la información de los proyectos LEADE, ya que éstos se elaboran en diferentes partes de la República Mexicana y la red internet es un medio económico de envío de información.

Las Metodologías que se necesitan para la construcción de este sistema son:

El Paradigma Orientado a Objetos, y

los conceptos de Bases de Datos Orientadas a Objetos

El software que se necesita para la construcción del sistema es:

Rational Rose como herramienta CASE,

ObjectStore como el Manejador de BDOO del sistema,

`Applets` como la interfaz gráfica.

`Servlets` como la capa de servicios del sistema, y

TOMCAT como el contenedor de `servlets`.

Todo el software de implementación es soportado por la plataforma JAVA, la cual consta de: lenguaje de programación con su compilador, máquina virtual y el conjunto de API's.

El capítulo 1 contiene la descripción de los procedimientos que se llevan a cabo en LEADE, así como la descripción de los requerimientos del sistema de información. Se introducen los diagramas de casos de uso de UML, para detallar la etapa de requerimientos del sistema. En el capítulo 2 se hace el diseño del sistema (jerarquía de clases) utilizando UML, asistido por Rational Rose. El capítulo 3 trata las BDOO comparándolas con las BDR, resaltando las ventajas que una BDOO tiene sobre una BDR. En el capítulo 4 se aborda la tecnología JAVA relacionada con `Applets`, componentes `Swing`, `Servlets` y TOMCAT, herramientas indispensables para la construcción del sistema. Se introduce el uso del manejador de bases de datos orientado a objetos, ObjectStore PSE Pro con sus características y beneficios. En el capítulo 5 se presenta la implementación, instalación y prueba del sistema.

Agradezco al Ing. Rubén Romero Ruiz por su apoyo en la dirección de esta tesis. También agradezco al Lic. Carlos Couder Castañeda por su asesoría en la tecnología JAVA, y a mi hermana Cuquis por su apoyo.

Capítulo 1

Requerimientos

En este capítulo se describen las necesidades de automatizar la información y los puntos clave que se tomaron en cuenta para construir el sistema. Se presenta cómo surgen los métodos usados en los proyectos LEADE que son parte importante del análisis de requerimientos.

Para el análisis de requerimientos del sistema se utilizó el Lenguaje de Modelado Unificado (UML) en específico los diagramas de casos de uso, ya que son una útil herramienta para el análisis de requerimientos en el diseño orientado a objetos.

1.1. Surgimiento de los métodos de LEADE

El objetivo de LEADE es brindar servicios a equipos en plantas de las refinerías de PEMEX y de otros países de América Latina. Para la realización de los proyectos, LEADE utiliza cinco métodos radiactivos, estadísticas y modelos matemáticos entre otros. Los métodos radiactivos son: Balance, Gammagrafía Longitudinal Electrónica (GLE), Nivel, Radiotrazado (RT) y Tomografía Axial Computarizada (TAC). Un proyecto puede contemplar el análisis de uno o más equipos.

A continuación se presenta de manera breve cómo surgen estos métodos.

LEADE lleva a cabo investigaciones de desarrollo tecnológico que incluyen la aplicación de las matemáticas en general y estadística desde los años 70's. A mediados de los 80's, se llevó a cabo un proyecto donde se vió el beneficio de utilizar trazadores radiactivos surgiendo así el primer método llamado Radiotrazado. La primera aplicación industrial del método de *Radiotrazado* fue en 1985 en Cozoleacaque, Veracruz.

A inicios de los 90's se agregaron cuatro métodos radiactivos adicionales al método de Radiotrazado, y son: *Nivel*, *Gammagrafía Longitudinal Electrónica*, *Tomografía* y *Balance*. Es importante mencionar que la aplicación de estas metodologías requiere de equipo sofisticado, caro y de importación, por lo que LEADE ha generado el desarrollo de diseños propios y patentes que protejan la propiedad intelectual.

Es relevante señalar que el uso de métodos radiactivos están estrictamente reglamentados por la Comisión Nacional de Seguridad Nuclear y Salvaguardia (CNSNS). Por lo que se han desarrollado instrumentos de trabajo, y normas de Seguridad Radiológica que protejan al personal. Estas normas de Seguridad Radiológica han permitido desarrollar el manual de Seguridad Radiológica, el manual de Procedimientos de Seguridad Radiológica y los formatos de Seguridad Radiológica. Estos manuales se actualizan al menos cada dos años. En la actualidad se está manejando la versión 1999 de dichos documentos.

Derivado de las metodologías para desarrollar los métodos radiactivos, estadísticas, gráficas, etc., se pueden llevar a cabo proyectos, por lo que un esquema que sintetiza la relación entre entidades que impulsan el desarrollo de estos proyectos, donde PEMEX es el cliente, se muestra en la figura I.

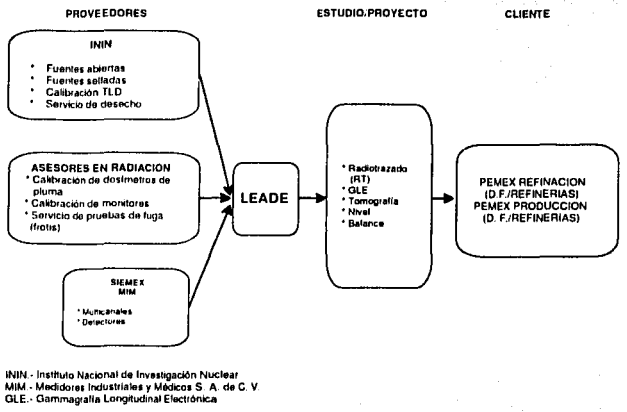


Figura 1.1. Diagrama general de proveedores y clientes para métodos LEADE

En la figura (1.1) también se observa, que LEADE requiere de entidades que le permiten efectuar proyectos, estas entidades son los proveedores.

Parte del fundamento teórico de los métodos radiactivos como el desarrollo de instrumentos y equipo específicos se hizo a través de tesis, servicios sociales y estancias académicas que alumnos de las diferentes universidades del país realizaron. Estos trabajos han participado en congresos, foros y simposiums.

Asimismo, partiendo de la experiencia en Seguridad Radiológica de LEADE, se participa con las autoridades normativas, para realizar investigación sobre la materia. También se colabora en la formación de recursos específicos de Seguridad Radiológica a nivel Institucional, Nacional, Regiones de Latino América e Internacional de Energía Atómica con cede en Viena Austria (en comunicación con la CNSNS).

1.2. Requerimientos del Sistema

La figura (1.2) muestra el organigrama, donde se puede localizar LEADE, lugar donde se llevará a cabo la automatización de información de sus proyectos. El área de LEADE está vinculada a la Gerencia de Investigación Aplicada de Procesos.

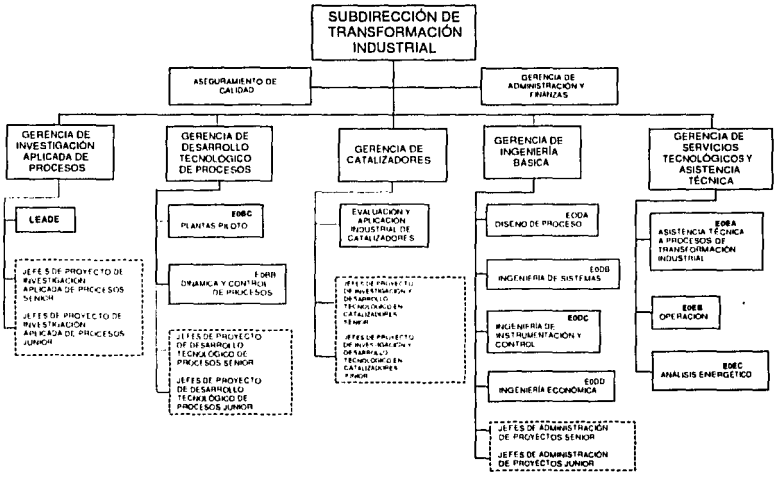


Figura 1.2. Organigrama de la Subdirección de Transformación Industrial

En la actualidad toda la información que se genera de los proyectos LEADE, así como sus resultados son almacenados en archiveros. Esta situación genera el inconveniente de tener volúmenes grandes de información y papeleo. Debido a que no existe un sistema de cómputo que lleve la organización de toda la información se dificulta el control y manejo de los datos. Las búsquedas aunadas con la localización de éstos requieren de mayor tiempo y existen pérdidas momentáneas de la información.

Dado que los documentos resultantes de los proyectos, ya sean reportes, formatos, estadísticas, etc., son generados con Excel y Word, origina el tener que llenar los formatos por medio de un archivo plantilla. Tanto los datos como los valores que se requieren para generar las estadísticas en Excel tienen que ser capturados por el usuario en el momento que se necesitan, generando pérdida de tiempo al verificar los errores que se pudieran cometer.

Al realizar un estudio a un equipo que pertenece a una planta y la planta a su vez a una refinería, se crea un proyecto y se le asigna una clave. Si surgiera el caso de realizar un estudio en la misma refinería en el tiempo que otro esta en proceso, pasará a formar parte de un proyecto existente o se abrirá un nuevo proyecto para este estudio. Por tal motivo un proyecto tiene al menos un estudio.

LEADE cumple con ciertos procedimientos para la realización de los proyectos. Estos procedimientos son:

a) Procedimiento de Radiotrazado, y

b) Procedimiento GLE, TAC, Nivel y Balance.

Estos dos procedimientos se detallan con diagramas de flujo de datos contenidos en el Apéndice A.

Las partes más relevantes de los procedimientos antes mencionados, que deben ser considerados como requerimientos del sistema y son los siguientes:

1. Actividad de Negociación
2. Dosimetría TLD
3. Dosimetría de pluma
4. Inventario de fuentes selladas

1. **Actividad de Negociación.** En esta etapa, el cliente ya sea PEMEX u otro, solicita al coordinador de LEADE lleve a cabo un estudio. El cliente debe indicar al coordinador el problema que tiene la refinería, la planta y el equipo donde se presentan las anomalías. En base a los datos proporcionados por el cliente, el coordinador de LEADE define el tipo de estudio que se va a efectuar. En caso de que el coordinador no tenga información sobre la refinería a estudiar el cliente proporciona los datos de la refinería.

Una vez establecido el tipo de estudio que se va a realizar, el coordinador establece la duración del estudio, número de personas que colaborarán en el estudio y número de métodos que se deben efectuar para detectar y corregir el problema, así como el costo del estudio.

La información requerida de esta etapa es:

El número de proyecto que se asigna a un estudio.

Estado en que se encuentra el estudio, es decir, si esta concluido o continúa.

La descripción del problema y el lugar, indicando la refinería, planta y equipo donde se presenta la anomalía. Además saber si se tiene información de la refinería donde está el problema.

La fecha de inicio y terminación del proyecto, ya que se requiere conocer el tiempo que dura cada proyecto.

La fecha de inicio y terminación de la negociación con la finalidad de saber el tiempo que consume la negociación de costos.

El tipo de solicitud que realizó el cliente, es decir, si fue por fax, teléfono, personal o a través de un oficio.

El número de personas que se requieren para el estudio, el número y costo de horas-hombre.

Los métodos radiactivos de cada estudio, así como saber cuántas veces se utilizó cada método y en qué estudio.

En caso de que el proyecto tenga guía para efectuar el estudio, saber el número de veces que se cambió la guía de formato teniendo las fechas de modificación de la guía por proyecto. También tener la relación de las facturas que se emitieron para el cobro del proyecto y los datos de éstas, como son: razón social, costo y fecha de cobro.

Si no tiene guía el proyecto sólo se contará con la relación de las facturas.

LEADE maneja dos modelos, ortho flow y ortho ortho flow, por lo que se necesita saber si alguno o ambos son requeridos.

Cuando se realizan los proyectos, cada uno debe contar con un informe, mismo que podrá tener una o varias versiones hasta llegar a un informe final. Por lo que se quiere conocer las fechas en que se modifica el informe y el tipo de informe, si es preliminar o final. También debe tener un título y observaciones del informe, como pueden ser: si se cumplió con el objetivo o alguna aclaración, etc.

Al personal que labora en LEADE se le da el nombre de POE, de quien se necesitan conocer sus datos personales, la clave del IMP, fecha en que ingresó a trabajar en LEADE, el cargo que ocupa, y si se encuentra actualmente activo o inactivo.

2. **Dosimetría TLD.** El POE que labora en LEADE necesariamente tiene asignado un dosímetro TLD. Ya que éste mide el nivel de radiación en el POE.

Este procedimiento lo realiza LEADE cada mes, quien recibe los dosímetros TLD del mes actual ya calibrados enviados por el ININ (Instituto Nacional de Investigación Nuclear), así como también, el reporte de dosimetría

personal del mes previo. Cabe mencionar que el reporte de dosimetría personal contiene el registro de dósis de exposición de cada dosímetro y el número de serie del dosímetro. La lectura de dósis de exposición que viene en el reporte, se archiva en un formato IMP.

LEADE debe recoger al POE los dosímetros TLD del mes previo y para ello elabora un formato de *Recolección de dosímetros TLD* que contiene número de serie del dosímetro, nombre del POE, fecha de entrega y firma. Al mismo tiempo se lleva a cabo la entrega por parte de LEADE al POE los dosímetros TLD del mes en curso, también elabora el formato de *Entrega de Dosímetros* que tiene número de serie del dosímetro, nombre del POE, fecha de recepción y firma del POE.

Una vez que se han recolectado todos los dosímetros TLD, se prepara un oficio para enviar a calibrar los dosímetros TLD al ININ. Tal oficio contiene el número de serie de cada dosímetro y el mes al que pertenecen.

Posteriormente, LEADE se comunica con el ININ para que envíe un mensajero al IMP y recoja los dosímetros para ser calibrados.

Tanto el procedimiento que se efectúa para los dosímetros de pluma como para los TLD, son requeridos para elaborar gráficas de dósis de exposición anual, con la finalidad de integrarse al reporte anual que elabora LEADE y el cual tiene que ser enviado a la Comisión Nacional de Seguridad Nuclear y Salvaguardia (CNSNS).

En esta etapa lo que se requiere es:

Los datos de los dosímetros TLD, como son: número de serie del dosímetro, dosis de exposición de cada dosímetro (por POE) correspondiente al mes previo.

Las dosis de exposición de los dosímetros, por mes y año.

Se quiere contar con un informe anual de este procedimiento por cada POE, el cual, en la actualidad se genera a través del formato

L/D 001B DOSIMETRÍA PERSONAL generado con Excel. Este informe deberá contar con la siguiente información:

Nombre del POE
RFC
Número de dosímetro
Año de la dosis acumulada
Lectura de dosis por cada mes

Generar una gráfica de la dosis de exposición registrada en los dosímetros, por cada POE. La gráfica debe contener dos parámetros, y son:

Mes de enero a diciembre y
Dosis por cada mes.

Generar dos formatos, uno para recolección de dosímetros TLD del mes anterior y otro para entrega de dosímetros TLD del mes en curso con los siguientes datos:

Nombre del POE
Número de serie del dosímetro

3. **Dosimetría de pluma.** Al igual que el procedimiento anterior, éste se efectúa por mes. Primero se recolectan los dosímetros de pluma del Personal Ocupacionalmente Expuesto (POE) con el formato de *Recepción de dosímetros* previamente hecho. La información que contiene el formato es: nombre del POE, la fecha en que entrega el dosímetro, número del dosímetro y su firma. El siguiente paso es tomar la lectura de dósis de exposición (en mR) por cada dosímetro, dicha lectura se registra en el formato de *Lectura de dosímetros* y se procede a archivar el documento.

Una vez que se calibran los dosímetros, éstos se regresan al POE con el formato *Entrega de dosímetros*, donde el POE debe anotar su nombre, fecha en que recibe el dosímetro y firma.

Cabe mencionar que puede darse el caso de que no se reúnan todos los dosímetros debido a que cierto personal esté realizando algún trabajo fuera del D. F. En ese caso, solo se realizará el procedimiento que arriba se mencionó con los dosímetros que se pudieron reunir, y después se hará lo mismo cuando se reúnan los dosímetros faltantes.

En esta etapa se requiere conocer:

El número de serie de cada dosímetro, las lecturas de dosis de exposición mensual y calcular las dosis anuales por cada dosímetro.

También se desea contar con un informe anual por cada POE de este procedimiento, el cual, en la actualidad se realiza a través del formato *L/D 001A Dosimetría Personal*, generado con Excel. Este informe deberá contar con la siguiente información:

Nombre del POE
El RFC del POE
Número del dosímetro
Año de la dosis acumulada.
Lectura por cada mes.

Generar una gráfica con dos parámetros:

Meses de enero a diciembre.
Dosis por cada mes.

Generar dos formatos, uno para recepción de dosímetros de pluma del mes anterior y otro para entrega de dosímetros de pluma del mes en curso.

4. **Inventario de fuentes selladas.** En el inventario se lleva una relación de las fuentes selladas (fuentes radioactivas con contenedor) que maneja LEADE para realizar los estudios. Los datos de la fuente son: actividad, ubicación, etc. Además se requiere conocer si la fuente fue donada o comprada. Así como saber los estudios donde se utilizan las fuentes y cuántas veces fueron utilizadas, en caso de que se pueda usar más de una vez.

Las fuentes abiertas (fuentes radioactivas sin contenedor) no se almacenan en LEADE, pero si se requiere una fuente de este tipo para algún estudio, se le pide a los proveedores y una vez que se deja de utilizar se regresa al ININ o al proveedor con el que se hizo el trato. Lo importante de estas fuentes, es que se lleva una relación de los estudios en que se utilizaron.

Lo que se necesita de las fuentes selladas (isótopo) es conocer:

El nombre de la fuente, su actividad radiactiva, la dimensión y otras características como son: su estructura, color, etc.

La marca, modelo, número de serie, fecha en que se hizo el inventario y fechas de pruebas de fuga por cada fuente. (No aplica a fuentes abiertas).

El número de licencia a la que pertenece la fuente en ese momento.

El nombre del representante legal del IMP que maneja lo referente al movimiento legal de las fuentes.

El nombre del representante de seguridad radiológica en la CNSNS.

Generar un reporte de las fuentes selladas con los siguientes datos:

1. Número de la fuente
2. Fuente
3. Marca
4. Número de serie

5. Actividad
6. Dimensión
7. Fecha de prueba de fuga
8. Fecha de últimos niveles de radiación
9. Ubicación

Lo que se requiere de las fuentes abiertas (isótopos) es:

Los mismos requerimientos que las fuentes selladas incluyendo los riesgos de inhalación, ingestión y contacto.

Tanto para fuentes abiertas y selladas se necesita la ubicación de las fuentes durante la realización del proyecto y la fecha en que cambia la responsabilidad. Es decir, controlar las instancias por las cuales pasan las fuentes en el momento en que se usa(n) en un proyecto.

1.3. Utilización de diagramas de casos de uso.

El diagrama de casos de uso es una de las técnicas de UML que ayuda a plantear los requerimientos de un sistema de cómputo en bloques significativos. Un caso de uso es una interacción típica entre el usuario y el sistema con el fin de lograr cierto objetivo, el punto de partida y los motores de todo el proceso de desarrollo. Y se consideran como una herramienta esencial para la captura, planificación y el control de requerimientos.

La mayoría de los casos de uso se generarán durante la primera instancia de creación del sistema, pero se pueden ir descubriendo otros a medida que se avanza. Ya que todo caso de uso es un requerimiento potencial y es importante capturarlo para poder plancar cómo manejarlo.

Tipos de vínculos para los diagramas de casos de uso.

Actor <actor>. Se emplea este término para llamar así a un usuario que interactúa con el sistema. Esto es, que los actores no forman parte del sistema sino que representan elementos que interactúan con él. En los diagramas de casos de uso, un actor es una entidad que lleva a cabo casos de uso. Un mismo actor puede realizar muchos casos de uso y a la inversa también. Éste puede introducir y/o recibir información desde o hacia el sistema.

Cuando se está trabajando con un sistema grande, es difícil obtener una lista de todos los casos de uso. Por lo que es más fácil definir la lista de actores y después tratar de determinar los casos de uso de cada actor.

Entre los elementos de un diagrama de casos de uso se pueden presentar *tres tipos de relaciones* y son las siguientes :

1. **comunica** <comunicates>, es una relación (asociación) entre un actor y un caso de uso que denota la participación del actor en dicho caso de uso.

2. **usa** <uses>, es una relación de dependencia entre dos casos de uso que denota la inclusión del comportamiento de un escenario en otro. Es decir, las relaciones **usa** ocurren cuando se tiene una porción de comportamiento que es similar en más de un caso de uso y no se quiere copiar la descripción de tal conducta.

3. **extiende** <extends>, es una relación de dependencia entre dos casos de uso, donde un caso de uso es una especialización de otro. Es decir, se utiliza cuando se tiene un caso de uso que es similar a otro, pero que hace un poco más.

Para los elementos (2) y (3) aplican las siguientes reglas:

- Se utiliza **extiende** cuando describa una variación de conducta normal.
- Se emplea **usa** para repetir cuando se trate de uno o varios casos de uso y desea evitar repeticiones.

La figura (1.3) presenta los elementos gráficos de un diagrama de casos de uso. Los incisos (a) y (b) son las dos posibles opciones para denotar una relación de asociación entre un actor y un caso de uso. Las relaciones **usa** y **extiende** se presentan en el inciso (c) y (d) respectivamente.

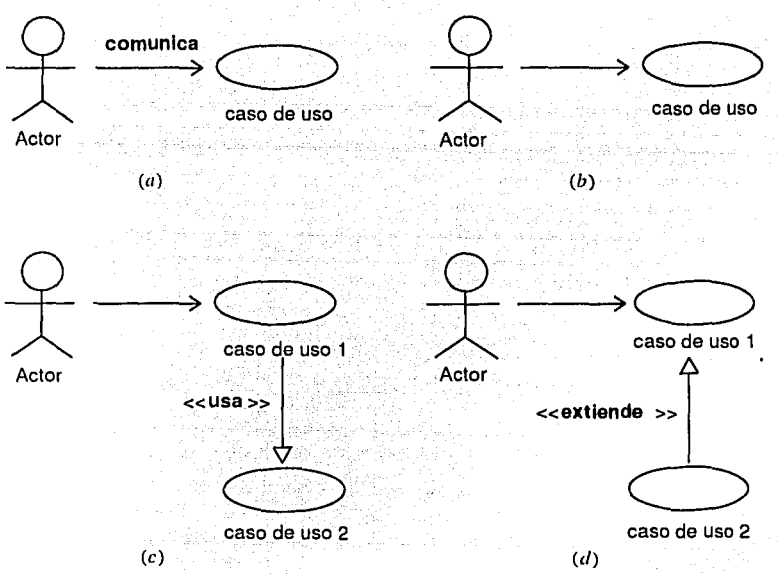


Figura 1.3. Elementos gráficos del diagrama de casos de uso

1.4. Diagramas de casos de uso del sistema

Los diagramas de casos de uso del sistema se construyeron en base a los requerimientos del sistema. Tomando en cuenta los procedimientos antes mencionados, éstos son:

Diagrama de casos de uso de la Actividad de negociación

Actores:

LEADE. Puede ser el coordinador de LEADE o algún subordinado del coordinador.

PEMEX. Empresa de gobierno que solicita estudios a LEADE.

GAF. Gerencia de Administración y Finanzas, cuyo papel es el de proporcionar a LEADE los costos horas-hombre y costo de los estudios.

Casos de uso:

Solicitud del estudio. Este caso de uso lo inicia PEMEX, ya sea por teléfono, fax, etc. Este caso de uso va a generar una solicitud de estudio con los datos correspondientes del problema.

Negociar costos. Este caso de uso lo inicia el coordinador de LEADE junto con PEMEX. Aquí deben establecer costos del proyecto.

Realiza trámites del proyecto. Caso de uso que inicia el coordinador de LEADE. Para llevarlo a cabo, usa la negociación de costos del proyecto.

Determina el costo y duración del proyecto. Caso de uso que inicia el jefe de LEADE.

Establece si el estudio se puede realizar. Lo inicia el coordinador de LEADE y debe verificar que existan las condiciones factibles para realizar el proyecto.

Determina costo de los recursos humanos. Caso de uso que inicia LEADE con ayuda de GAF quien proporciona los costos horas-hombre a LEADE

Establece recursos tecnológicos y humanos para realizar el estudio. Caso de uso que inicia LEADE con ayuda de la evaluación de la solicitud. LEADE necesita saber el material y el personal que requiere para la elaboración de los estudios en las refinerías.

Evaluación de la solicitud. Caso de uso que inicia LEADE, para verificar si se conoce toda la información del equipo a estudiar.

Evaluación directa al equipo. Caso de uso que realiza LEADE cuando no se tienen los datos suficientes para realizar el estudio en un equipo determinado.

La figura (1.4) presenta el diagrama de casos de uso anteriores.

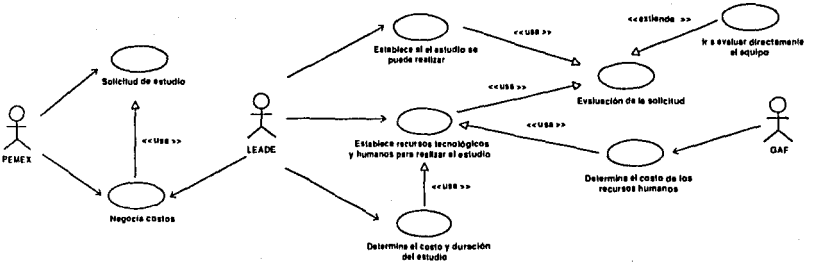


Figura 1.4. Diagrama de casos de uso de la Actividad de Negociación

Diagrama de casos de uso de Dosimetría TLD

Actores:

ININ. Es la Institución encargada de rentar y calibrar los dosímetros.

LEADE. Es el área que se encarga de generar los reportes necesarios.

Casos de uso:

Genera oficio para envío al ININ de dosímetros del mes. Este caso de uso lo inicia LEADE.

Registra reporte de dosímetros, el cual se apoya del reporte enviado por ININ. Caso de uso que inicia LEADE, se apoya en el caso de uso de generar reporte de dosimetría personal del mes que realiza el ININ.

Genera reporte de dosimetría personal del mes que corresponda. Este caso de uso lo inicia el ININ.

En la figura (1.5) se encuentra el diagrama de casos de uso de esta fase.

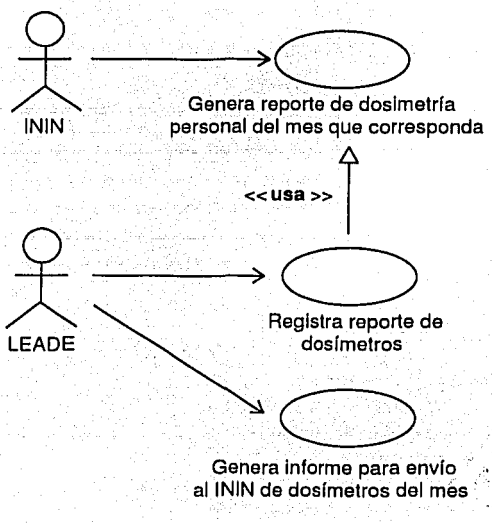


Figura 1.5. Diagrama de casos de uso de Dosimetría TLD

Diagrama de casos de uso de Dosimetría de Pluma

Actores:

LEADE. Es el área encargada de llevar el control de los dosímetros de pluma.

Casos de uso:

Los siguientes casos de uso los inicia LEADE

Generar formato Recepción de Dosímetros de pluma del POE.

Generar reporte mensual-anual de dosis de exposición del POE.

Registro de dosis de exposición de cada mes del POE.

Generar formato de entrega de dosímetros.

La figura (1.6) presenta gráficamente el diagrama de los casos de uso mencionados.

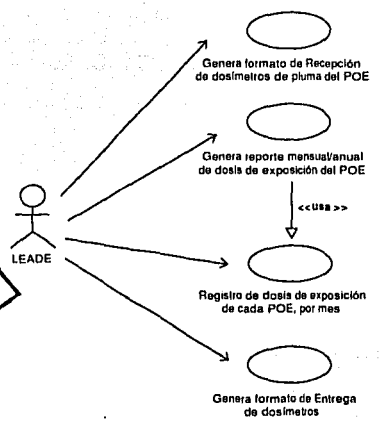


Figura 1.6. Diagrama de casos de uso de Dosimetría de Pluma

Diagrama de casos de uso del Inventario de Fuentes Selladas

Actores:

CNSNS. La Comisión Nacional de Seguridad Nuclear y Salvaguardia tiene a su cargo, evaluar la licencia sobre el uso de fuentes radiactivas.

LEADE. Elabora los documentos para solicitar la licencia de uso de fuentes radiactivas.

Casos de uso:

Entrega licencia a CNSNS. LEADE presenta la licencia a la CNSNS con el inventario de fuentes que se tienen en el Laboratorio.

Evalúa licencia. La CNSNS debe corroborar que los documentos para solicitar la licencia cumplan con los requisitos necesarios para su aprobación.

Requerir fuente adicional. Caso de uso que realiza LEADE donde es el encargado de entregar una lista de las fuentes que se necesitarán a futuro.

Pedir cotización. Una vez que se tiene la lista de fuentes que se requerirán a futuro, LEADE efectúa este caso de uso para poder comprar las fuentes que pasarán a ser parte del inventario de fuentes selladas.

Revisión de fuente. Antes de que se pueda inventariar la fuente, LEADE debe verificar que la fuente esté en buen estado para proceder a almacenarla

Verifica licencia vigente. LEADE realiza este caso de uso, en caso de que la licencia no esté vigente, se elabora un documento para requerir la renovación de la licencia con el inventario de fuentes selladas.

Renovar licencia. Una vez elaborada la licencia por LEADE se manda a la CNSNS para la actualización.

Formato de licencia. LEADE utiliza este caso de uso para elaborar la licencia.

Lista de fuentes a futuro. LEADE crea una lista de fuentes para que sean soportadas por la licencia a futuro.

La figura (1.7) presenta el diagrama correspondiente a estos casos de uso.

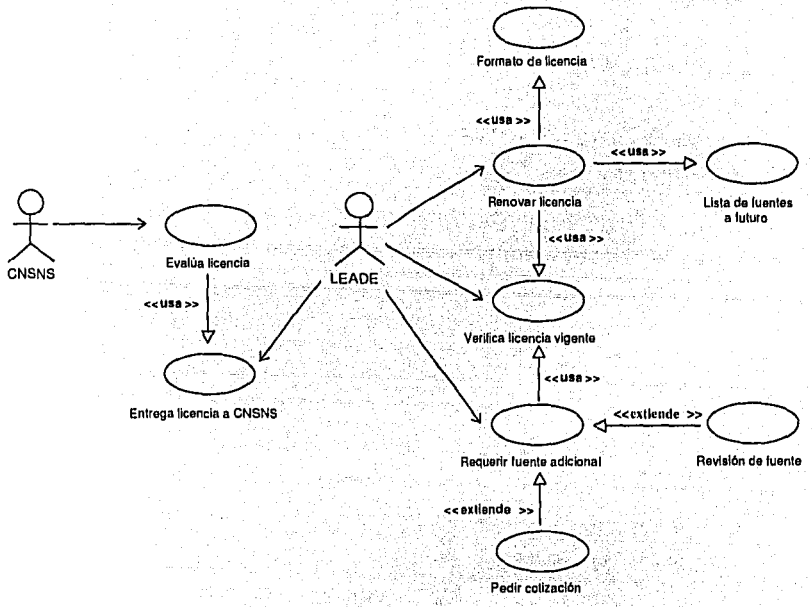


Figura 1.7. Diagrama de casos de uso de Inventario de Fuentes Selladas

Capítulo 2

Análisis y Diseño

En este capítulo se presenta el análisis del sistema encontrando las clases del sistema y los paquetes donde se agrupan las clases. Cabe mencionar que en el análisis se pueden hacer uso de herramientas como tarjetas CRC y diagramas de interacción, estos últimos se dividen en dos: diagramas de secuencia y diagramas de colaboración.

Las tarjetas CRC se utilizan cuando no es claro la obtención de las clases del sistema, las responsabilidades de cada clase y las colaboraciones para llevar a cabo su responsabilidad. También se desechan las que inicialmente fueron clases tentativas y no se les encontró alguna responsabilidad.

Los diagramas de interacción describen la manera en que colaboran grupos de objetos y su comportamiento. Habitualmente, un diagrama de interacción capta el comportamiento de un solo caso de uso; el diagrama muestra cierto número de objetos y los mensajes que se pasan entre estos objetos dentro del caso de uso. Los diagramas de interacción se pueden usar cuando se desea visualizar el comportamiento de varios objetos de un caso de uso. Son eficientes para mostrar la colaboración entre los objetos, sin embargo no definen con presición dicho comportamiento.

Otros diagramas usados en la etapa de análisis son los diagramas de estado, es una técnica usada para describir el comportamiento de un sistema o un objeto en particular. Describen todos los estados posibles que puede

tener un objeto y la manera en que cambia, como resultado de los eventos que llegan a él. En la mayor parte de las técnicas OO, los diagramas de estados se dibujan para una sola clase, mostrando el estado de un solo objeto durante todo su ciclo de vida.

Los diagramas anteriores así como las tarjetas CRC pueden ser o no utilizados para el análisis. Esto depende de la complejidad del sistema y de la experiencia del profesional que desarrolle el sistema.

Finalmente se presenta el diseño del sistema donde se crearon tanto el diagrama de paquetes como el diagrama de clases.

2.1. Encontrando clases

Para encontrar las clases que son parte del sistema se deben tomar en cuenta los siguientes puntos. Algunas serán clases obvias, otras no tanto. La meta es lograr obtener una lista de clases candidatas donde algunas clases de la lista serán eliminadas y otras no. Durante el proceso de búsqueda de clases es común que se entienda en algunas situaciones, cómo es la búsqueda de objetos, ya que la frontera entre estos dos es muy delgada. A continuación se enuncian los puntos para encontrar las clases:

- Modelar objetos físicos.
- Modelar entidades conceptuales.
- Si más de una palabra aplica a un concepto, elegir la más significativa, es decir, una sola palabra por cada concepto.
- Ser cuidadoso con el uso de adjetivos.
- Ser cuidadoso con la voz pasiva.
- Modelar las categorías de clases.
- Modelar interfaces.

- Modelar los valores de los atributos, no los atributos.

Las clases que se encontraron en el sistema son:

- `Proyecto`. Un proyecto se crea cuando se va a realizar un estudio en la refinería.
- `Estudio`. Un estudio es el que se realizará en la refinería
- `POE`. Un POE es la persona que estará efectuando el estudio o colaborando para su elaboración.
- `Guia`. Una guía contiene información relacionada con las fechas en que cambia el formato de la guía para el proyecto que la contenga.
- `Programa`. Un programa de actividad contiene el número de facturas en que se cobra el proyecto.
- `Informe`. Un informe engloba los datos importantes de los métodos que se utilizaron en un proyecto, los modelos en caso de que se hallan requerido y los resultados que se obtuvieron.
- `Fuente`. La fuente se utiliza cuando se hacen los estudios. Hay dos tipos de fuentes: selladas y abiertas.
- `Equipo`. Un equipo esta contenido en una planta y es donde se hacen los estudios.
- `Planta`. Una planta esta contenida en una refinería y es donde se hacen los estudios.
- `Refineria`. Una refinería es donde se hacen los estudios.
- `Modelo`. Hay dos modelos y se puede usar uno o ambos en los estudios.
- `Solicitud`. La solicitud la presenta el cliente a LEADE indicando la refinería que presenta el problema.

- `DosimetroPluma`. El dosímetro es el que usa el POE todo el tiempo ya que éste mide la dosis de exposición que tiene el POE cuando maneja las fuentes radiactivas.
- `DosimetroTLD`. Al igual que el dosímetro anterior lo usa el POE.

Las clases listadas conforman el conjunto de *clases entidad* del sistema, es decir las clases principales que modelan la lógica del negocio, sin embargo es hasta el proceso de modelaje y a criterio de cada analista, el definir *clases asociación* y *composición* que se detallan en el diagrama de clases.

2.2. Tarjetas CRC

CRC es la abreviación de Clases-Responsabilidades-Colaboraciones. Las tarjetas CRC son una herramienta CASE donde cada tarjeta CRC representa un objeto que interviene en el sistema.

Una responsabilidad es una descripción de alto nivel del propósito de una clase. La idea es tratar de eliminar la descripción de pedazos de datos y procesos, y en cambio, captar el propósito de la clase en unas cuantas frases. En la colaboración, cada responsabilidad indica cuáles son las otras clases con las que se tiene que trabajar para cumplirlas. Esto proporciona cierta idea sobre los vínculos entre las clases, siempre a alto nivel.

Las tarjetas son especialmente eficaces cuando se está en medio de un caso de uso para ver cómo se va a implementar por medio de las clases. Se usan cuando hay confusión en encontrar las clases e identificar clases apelmazadas y carentes de definiciones claras.

Para el análisis del sistema de información no hubo necesidad de utilizar extensamente esta herramienta, puesto que la obtención de las clases no fue complicado. La figura (2.8) muestra la estructura de las tarjetas CRC.

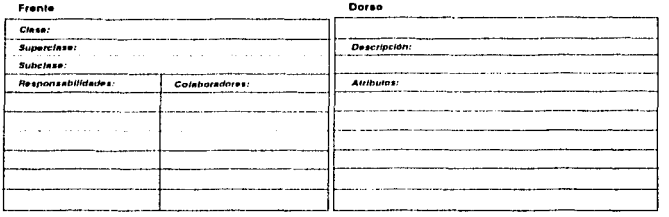


Figura 2.8. Tarjeta CRC

2.3. Diagrama de secuencias

Para describir el comportamiento dinámico del sistema, los dos diagramas de interacción de UML pueden ser utilizados. Para este trabajo se presenta un ejemplo de un diagrama de secuencias.

En un diagrama de secuencias, un objeto se representa como un rectángulo en la parte superior de una línea vertical punteada. La línea vertical se llama línea de vida del objeto y representa la vida del objeto durante la interacción. Los mensajes se representan mediante una flecha entre las líneas de la vida de dos objetos y el orden en que se dan estos mensajes transcurre de arriba hacia abajo. Cada mensaje es etiquetado por lo menos con el nombre del mensaje y se representan por una línea dirigida (flecha); en los mensajes que un objeto se envía a sí mismo se le llama *autodelegación*, la flecha de mensaje se dirige a la misma línea de vida del objeto. En el mensaje también se pueden incluir argumentos o alguna información de control, la información de control consta de dos partes:

1. Existen condiciones que indican cuándo se envía un mensaje de un objeto a otro, por ejemplo [`necesitaAlta`]. El mensaje se envía solo si la condición es verdadera.

2. Un marcador de iteración, que muestra que un mensaje se envía a varios objetos receptores, como sucedería cuando se itera sobre una colección. La base de la iteración se puede mostrar entre corchetes, por ejemplo *[`para cada objeto de tipo 2`].

En la figura (2.9) se puede apreciar el diagrama de secuencias con su elementos.

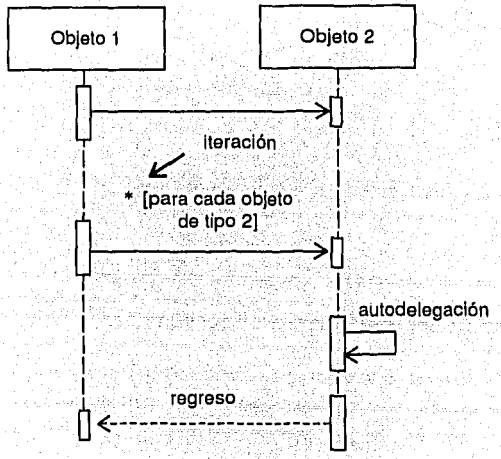


Figura 2.9. Símbolos de un diagrama de secuencias en notación UML

En la figura (2.10) se muestra un diagrama de secuencias del sistema. Como se puede ver, los objetos que colaboran para agregar uno o varios POE's a un proyecto son; `Proyecto`, `ExtensionPOE` (colección que contiene los objetos POE), `POE`.

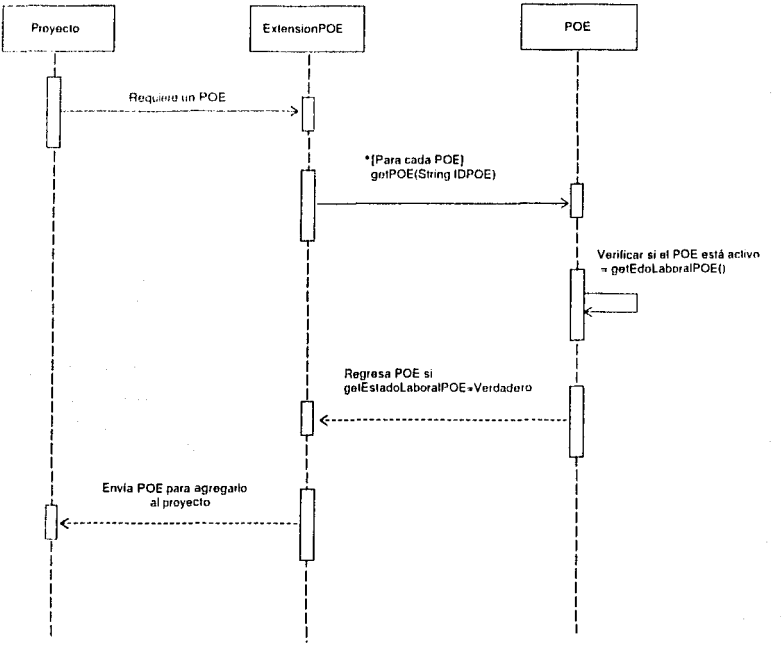


Figura 2.10. Diagrama de secuencias para agregar un POE a un proyecto

2.4. Diagrama de estados

Los diagramas de estado resultan adecuados para describir el comportamiento de un objeto a través de diferentes casos de uso, sin embargo, no resultan del todo adecuados para describir el comportamiento que incluye a una serie de objetos colaborando entre sí. Por lo tanto, resulta útil combinar los diagramas de estado con otras técnicas como los diagramas de interacción.

No es necesario elaborar los diagramas de estado para todos los objetos del sistema, se construyen sólo para aquellos que tengan un comportamiento no trivial, de tal forma que el diagrama de estados ayude a entender dicho comportamiento.

En la figura (2.11) se presentan los elementos que componen a un diagrama de estado.

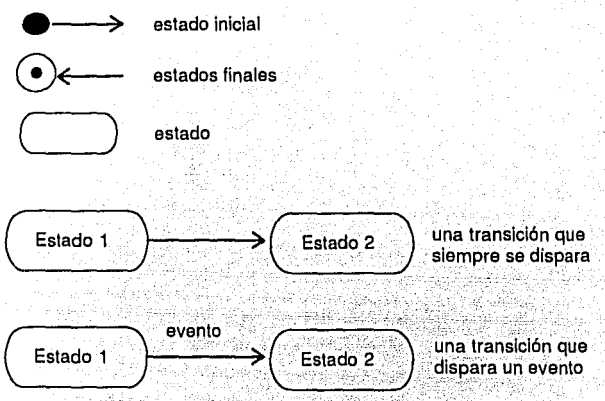


Figura 2.11. Símbolos de un diagrama de estados en notación UML

El *evento* es una ocurrencia que puede causar la transición de un estado a otro del objeto. Esto ocurre por:

- condición que toma el valor de verdadero o falso
- recepción de la señal de otro objeto dentro del modelo,
- recepción de un mensaje,
- paso de cierto periodo de tiempo,
- después de entrar al estado o de cierta hora y fecha particular.

La *transición* es una relación entre dos estados donde un objeto en un estado determinado pasa a otro. El *estado* especifica la respuesta del objeto a los eventos de entrada, también indica un periodo de tiempo en la vida del objeto durante el cual está esperando alguna operación. Solo puede haber un estado inicial, pero puede haber más de un estado final.

La figura (2.12) muestra los estados que presenta el dosímetro de pluma dentro del sistema.

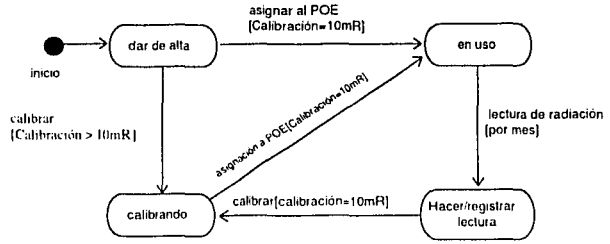


Figura 2.12. Diagrama de estados para el dosímetro de pluma

2.5. Paquetes

En general, un paquete es un mecanismo de agrupación, por el cual todos los tipos de modelos pueden ser enlazados. En el UML, un paquete esta definido como un mecanismo de agrupación, el cual no tiene ninguna semántica. Por lo tanto, los paquetes sólo tienen significado durante el trabajo de modelaje, pero no necesariamente en el sistema ejecutable. Un paquete es dueño de sus elementos, y los elementos de un modelo no pueden pertenecer a más de un paquete.

En el UML los paquetes se representan como lo muestra la figura (2.13). Aquí podemos ver que el nombre del paquete es *información de personas.*

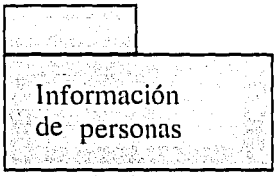


Figura 2.13. Notación de UML para un paquete

Un paquete puede tener distintos niveles de acceso y el UML define tres niveles: `private` (privado), `protected` (protegido), `public` (público). Estos niveles de accesibilidad serán explicados más adelante con detalle, ya que aplican también a clases, métodos y atributos. Si el paquete no especifíca su nivel de acceso, por defecto será público.

Los paquetes que se crearon para el sistema se muestran en la figura (2.14).

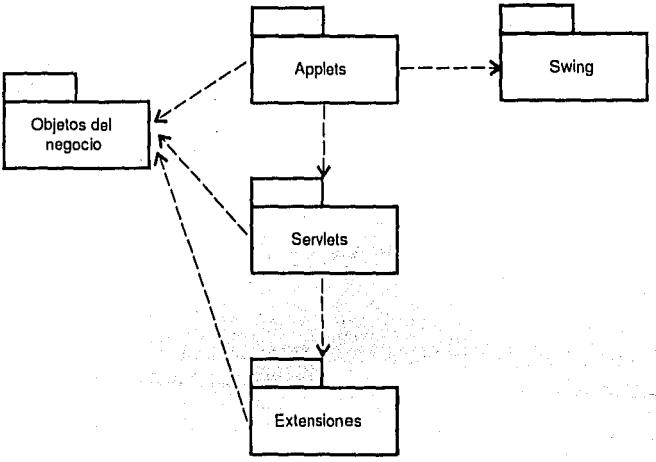


Figura 2.14. Dependencias de los paquetes del sistema

`applets`. Este paquete contiene todos los applets de la aplicación. Los applets representan la interfaz de usuario del sistema.

`servlets`. Este paquete contiene los servlets que se encargan de dar respuesta a las peticiones de los clientes, que pueden ser applets y/o páginas electrónicas.

`swing`. Este paquete contiene los componentes swing que se utilizaron para la construcción de los applets.

`objetos del negocio`. Este paquete contiene el diagrama de clases que describe la lógica del negocio.

`extensiones de la base de datos`. Este paquete contiene las extensiones que se encargan de manejar los objetos persistentes de la base de datos.

2.6. Diagrama de clases

La fase de diseño (los modelos UML resultantes) expande y detalla los modelos de análisis tomando en cuenta todas las implicaciones y restricciones técnicas, el propósito del diseño es especificar una solución que trabaje y que pueda ser fácilmente convertida en código fuente y que permita construir una arquitectura simple y fácilmente extensible.

El diagrama de clases describe los tipos de objetos que hay en el sistema y los diversos tipos de relaciones estáticas que existen entre ellos. Un diagrama de clases esta compuesto por *clases* y *relaciones*.

Clase

Es la unidad básica que encapsula toda la información de un objeto. A través de ella podemos modelar el entorno en estudio (una casa, un auto, etc.).

Una clase en UML se representa como se indica en la figura (2.15).

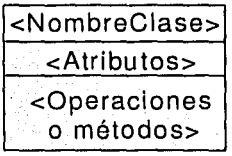


Figura 2.15. Representación de una clase en UML

Los atributos o características de una *clase* modelan el estado de los objetos que son creados a partir de ésta y los métodos u operaciones de una clase son la forma en como los objetos interactúan con su entorno.

Al proceso de crear un objeto a partir de una clase se le conoce como instanciación.

Los niveles de acceso más comunes para paquetes, clases, atributos y métodos son los siguientes: (ver figura (2.16))

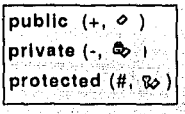


Figura 2.16 Niveles de acceso

- *public* (público). Para el caso de los atributos indica que estos pueden ser accesados por cualquier objeto dentro del sistema. En el caso de métodos, indica que pueden ser ejecutados por cualquier objeto. Las clases públicas indican que se pueden crear objetos a partir de éstas desde cualquier parte del sistema, en el mismo sentido que las clases el nivel de acceso public se aplica a los paquetes.

- *private* (privado). Para el caso de los atributos sólo serán accesibles por el objeto que lo contiene. En el caso de métodos indica que sólo

pueden ser ejecutados por el objeto que lo posee. Una clase privada solo tiene sentido cuando ésta reside dentro de otra para que puedan ser creados objetos a partir de la clase que lo contiene (como en el caso de composición fuerte). Un paquete debe ser privado sólo si es parte de otro.

- *protected* (protegido). Para el caso de los atributos no serán accesibles desde fuera de la clase, pero sí podrán ser accesados por objetos creados a partir de las subclases que se deriven. En el caso de JAVA un atributo protegido puede ser accesado por todos los objetos que pertenecen al mismo paquete donde se encuentra la clase que lo define. Para los métodos y las clases, el acceso protegido es en el mismo sentido que los atributos. Los paquetes protegidos tienen sentido cuando se crean paquetes de paquetes.

El chequeo de los niveles de acceso se lleva a cabo durante el tiempo de compilación.

2.6.1. Relaciones entre clases

Los diagramas de clases muestran como las clases se relacionan entre sí y por lo tanto como interactúan los objetos. Las relaciones más significativas son: la *herencia*, la*composición* y la *asociaciación*.

Herencia

Indica que una subclase hereda los métodos y atributos especificados por una super-clase, por ende la subclase, además de poseer sus propios atributos y métodos, poseerá las características y atributos de la super-clase. Debe mencionarse que existe la herencia privada, sin embargo esta ligada al reuso de código y se aparta del concepto de especialización-generalización, por lo que en este trabajo al referirse a la herencia, debe entenderse como pública o ligada a la especialización-generalización.

La figura (2.17) muestra cómo se representa la herencia.

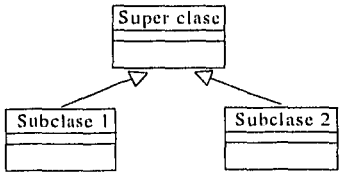


Figura 2.17. La herencia en UML

Composición

Para modelar objetos complejos, no bastan los tipos de datos básicos que proveen los lenguajes como: enteros, reales, secuencias de caracteres, etc., por lo tanto, se necesita definir objetos que sean componentes o partes de dichos objetos complejos. Si un objeto A necesita de un objeto B para llevar a cabo sus tareas, entonces se dice que el objeto B es componente o parte de A, o que el objeto A contiene al objeto B; la organización ODMG nombra al objeto B como atributo-objeto-valuado.

La composición es una relación entre objetos que tiene dos propiedades importantes: la transitividad y la antisimetría. La transitividad indica que si un objeto A es componente del objeto B y el objeto B es componente del objeto C, entonces el objeto A es componente del objeto C. La antisimetría, define que si un objeto A es componente del objeto B, entonces el objeto B no puede ser componente del objeto A.

Cuando se requiere hacer composición de objetos que son instancias de clases definidas por el programador de la aplicación, hay dos posibilidades:

Composición fuerte. Es un tipo de relación que se define de forma estática, en donde el tiempo de vida del objeto componente esta condicionado por el tiempo de vida del que lo incluye. Este tipo de relación es llamada simplemente **composición o composición fuerte.**

Composición débil. Es un tipo de relación dinámica, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye. Este tipo de relación es comúnmente llamada **agregación o composición débil.**

Para ejemplificar la composición fuerte y débil supongamos que tenemos un objeto de tipo `triángulo` que necesita de tres objetos de tipo `punto` para construirse, por lo tanto se dice que los objetos de tipo `punto` son componentes o parte del objeto de tipo `triángulo`. Si al borrar o destruir el objeto de tipo `triángulo`, ya no son necesarios los objetos de tipo `punto`, por lo tanto también son destruidos, entonces se dice que existe una relación de composición fuerte entre el objeto `triángulo` y los objetos `punto`. Ahora si un objeto de tipo `punto` es compartido por dos objetos, uno de tipo `triángulo` y el otro de tipo `cuadrado`, entonces al desaparecer el objeto de tipo `triángulo` no puede destruirse el objeto de tipo `punto` que es compartido, puesto que es necesitado por el objeto de tipo `cuadrado`, entonces se dice que existe una composición débil o agregación entre los objetos `triángulo-punto` y `cuadrado-punto`.

La figura (2.18) muestra cómo se representan la composición en sus dos tipos fuerte y débil.

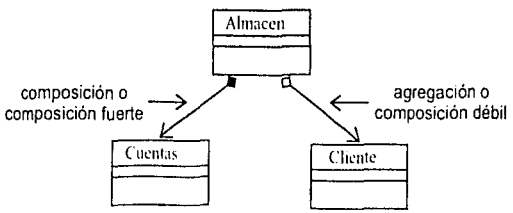


Figura 2.18. Composición

La figura (2.18) muestra que un objeto de tipo `Almacen` necesita de un objeto de tipo `Cuentas` para poder construirse, y que cuando el objeto de tipo `Almacen` sea destruido o borrado, también se destruirá el objeto de tipo `Cuentas` con el que se relaciona. El objeto de tipo `Almacen` también necesita de un objeto de tipo `Cliente` para construirse, pero que al borrarse el objeto de tipo `Almacen` no se destruirá el objeto de tipo `Cliente`. No debe confundirse las flechas de la figura (2.18) con la herencia, nótese que son dos tipos de flecha distinta, el tipo de flecha utilizado en la figura (2.18) denota

navegabilidad es decir que a partir del objeto `Almacen` se puede acceder al objeto `Cuentas` o al objeto `Cliente`.

Asociación

La relación entre clases conocida como asociación, permite relacionar objetos que colaboran entre sí, pero que no dependen para construirse uno del otro. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro. En la figura (2.19) se ve como un objeto de tipo `Cliente` puede tener cero o muchos objetos de tipo `OrdenCompra`, en cambio un objeto de tipo `OrdenCompra` solo puede tener asociado un objeto de tipo `Cliente`.

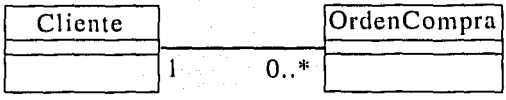


Figura 2.19. Relación de asociación

En la figura (2.19) se introduce el concepto de cardinalidad, al decir que un objeto de tipo `Cliente` se relaciona con cero o muchos objetos de tipo `OrdenCompra`; la cardinalidad especifica la cantidad de objetos que participarán en la relación dada. Nótese también que la flecha de navegabilidad no esta denotada, esto quiere decir que la relación es bidireccional, por lo tanto a partir del objeto de tipo `Cliente` se puede llegar al objeto de tipo `OrdenCompra` por medio de una referencia y viceversa.

La cardinalidad solo es aplicable a las relaciones de tipo composición y asociación.

En UML, la cardinalidad de las relaciones indica el grado y nivel de dependencia, se anotan en cada extremo de la relación y éstas pueden ser:

1. *uno a muchos*: 1..* ó (1..n)
2. *0 a muchos*: 0..* ó (0..n)
3. *0 ó 1*: 0..1

4. *número fijo*: n

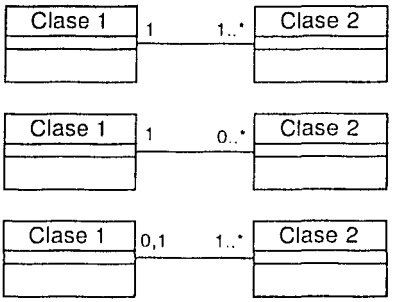


Figura 2.20. Cardinalidad entre clases

Dependencia de instanciación

La dependencia de instanciación es una relación de tipo composición con restricción en el tiempo de creación de los objetos relacionados y especifica que si un objeto A es componente de un objeto B, el objeto B es el encargado de instanciar o construir el objeto A. Debe observarse que la composición indica que se requiere de un objeto pero no cuando este objeto componente debe ser creado, los tipos de composición fuerte y débil imponen restricciones al tiempo de vida de los objetos componente y la dependencia de instanciación impone restricciones en el tiempo de creación.

Por ejemplo en la figura (2.21), una aplicación gráfica que instancía una ventana (la creación del objeto Ventana esta condicionado a la instanciación proveniente desde el objeto Aplicación).

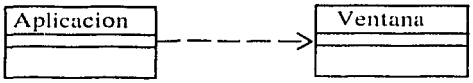


Figura 2.21. Relación de dependencia

Es difícil encontrar una relación de dependencia sin que esta sea también una relación de composición, ya que si un objeto A instancia un objeto B es porque lo requiere, es decir es su componente, si el objeto A deja de existir, es poco probable que se requiera del objeto B. En la figura (2.21), si la aplicación requiere crear una ventana para mostrarse, después de cerrar la aplicación no tiene sentido que siga existiendo la ventana.

La figura (2.22) muestra el diagrama de clases del sistema que se desarrolló para LEADE y por lo tanto la interacción entre los objetos; asimismo denota también el modelo de datos al tratarse de una base de datos orientada a objetos. Las siguientes cuatro páginas a la figura (2.22) son parte de ésta y muestran los atributos y métodos de cada una de las clases, que no se pueden mostrar en el diagrama de clases por falta de espacio.

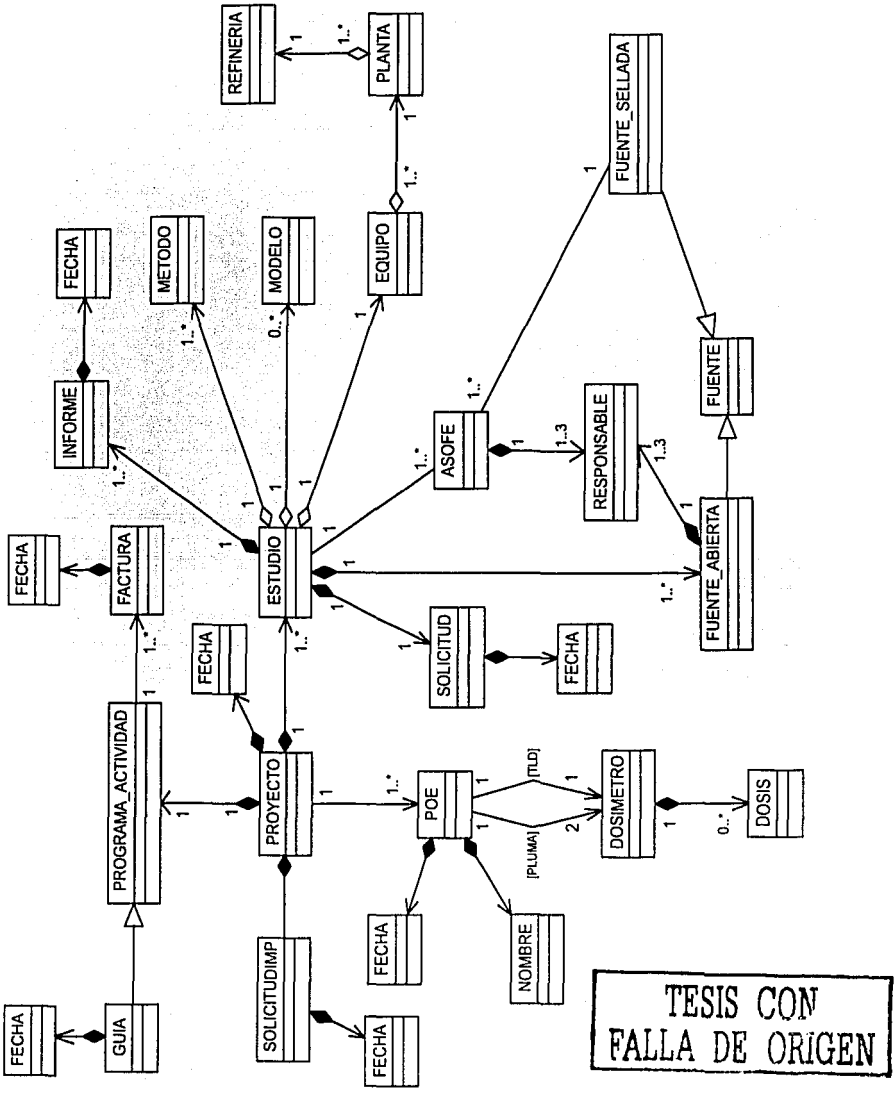


Figura 2.22. Diagrama de clases del sistema LEADE

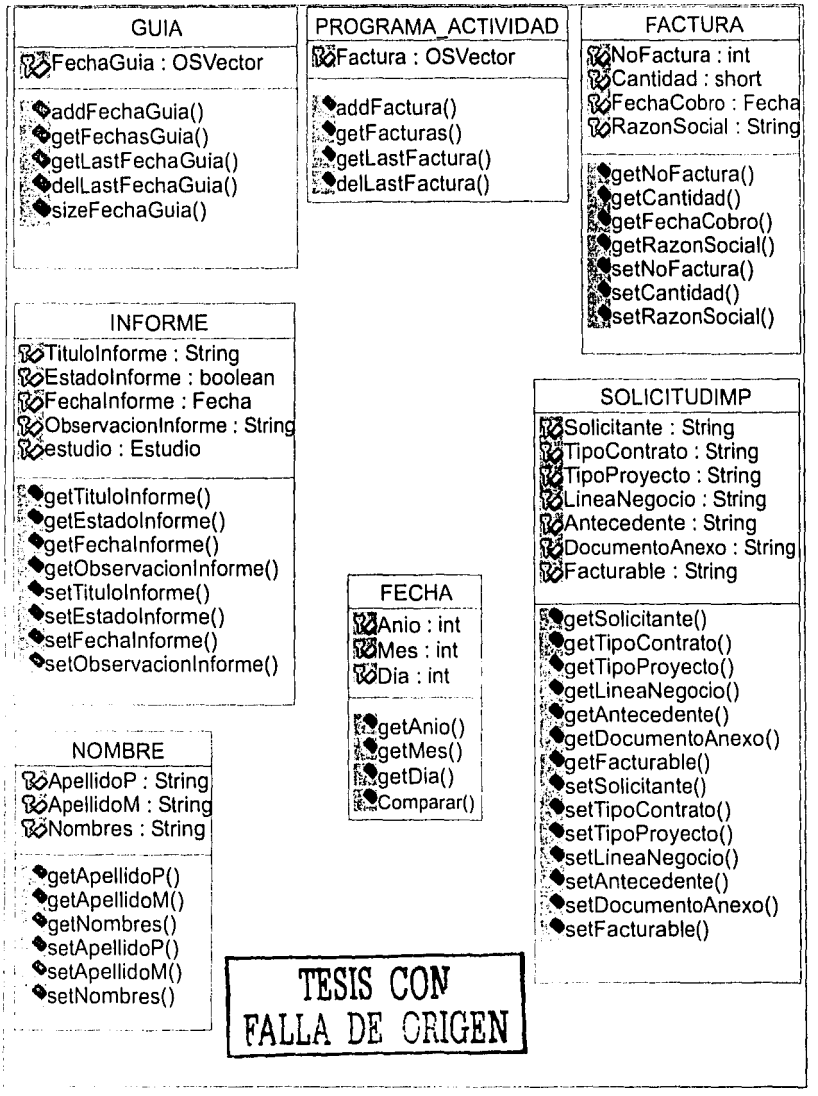


Figura 2.22. (Continuación)

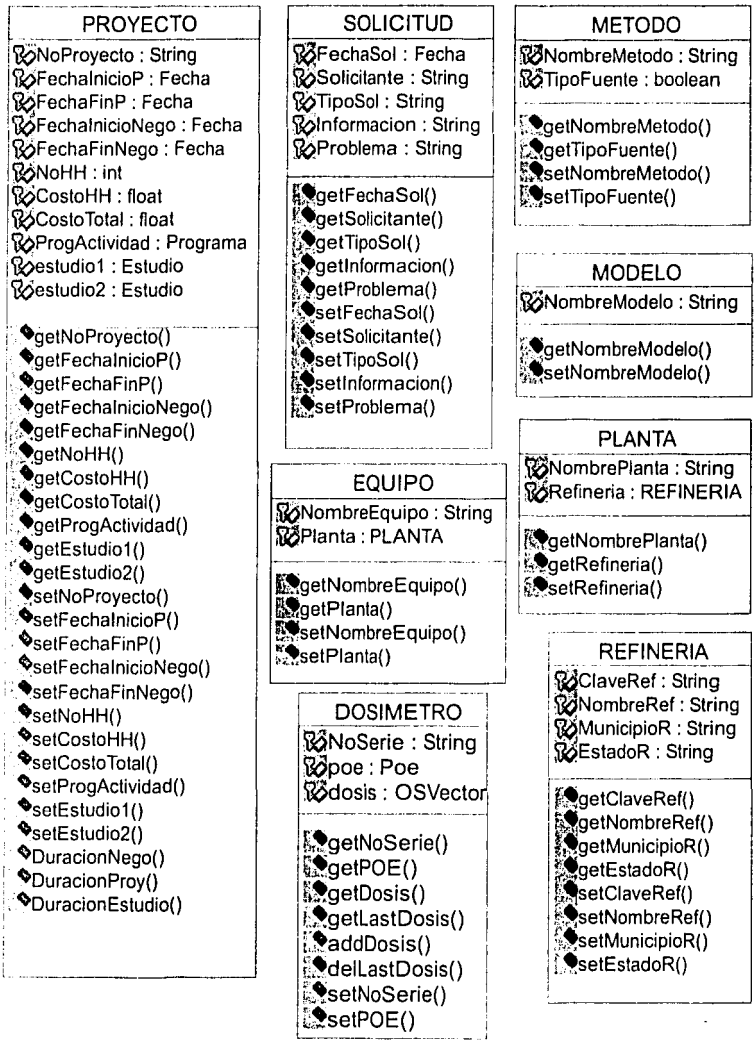


Figura 2.22. (Continuación)

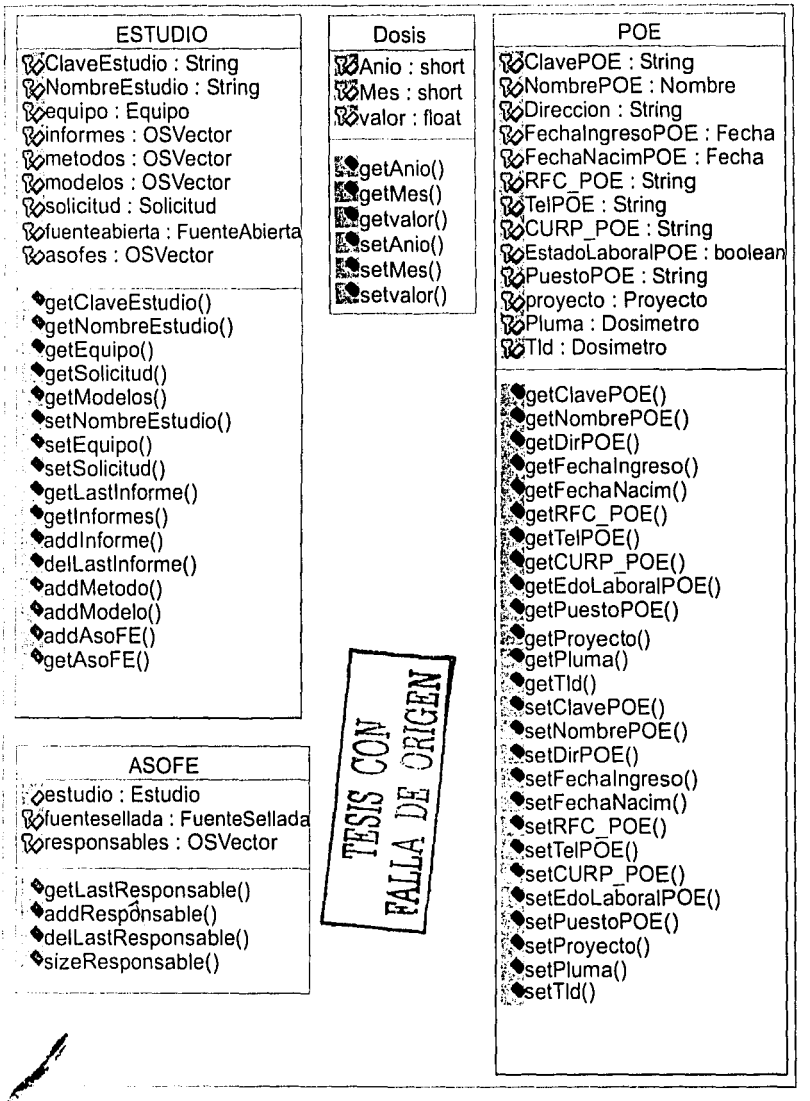


Figura 2.22. (Continuación)

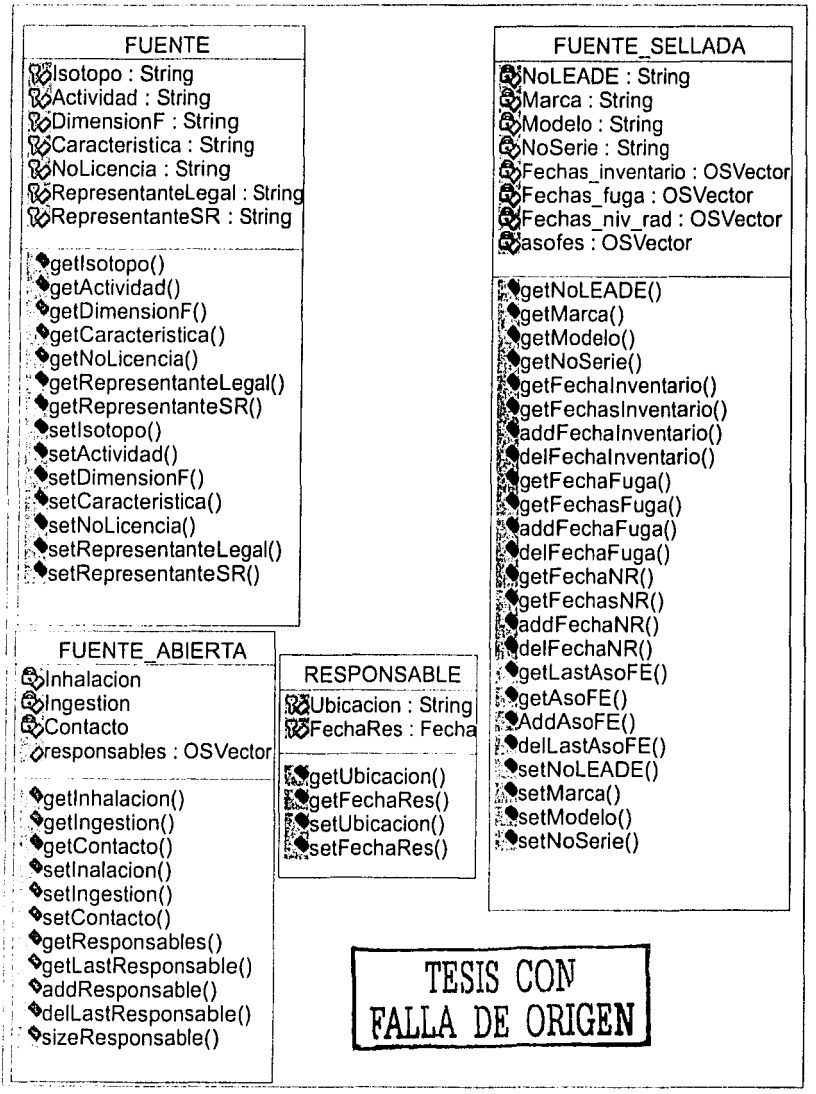


Figura 2.22. (Continuación)

Capítulo 3

Bases de Datos Orientadas a Objetos

En este capítulo se abordan las Bases de Datos Orientadas a Objetos (BDOO) y la arquitectura del Sistema Manejador de Bases de Datos Orientado a Objetos (SMBDOO). Para comprender las BDOO es necesario introducir las definiciones y características del paradigma orientado a objetos.

Las bases de datos relacionales utilizan el modelo Entidad-Relación en donde los datos se organizan en tuplas y las tuplas se organizan en tablas. Las bases de datos orientadas a objetos utilizan el Modelo de Datos Orientado a Objetos (MDOO). Se explica el MDOO y las diferencias con el modelo Entidad-Relación.

3.1. Programación Orientada a Objetos

La programación orientada a objetos (POO) es un paradigma que facilita la creación de software de calidad por sus factores que potencían el mantenimiento, la extensión y la reutilización.

La POO trata de aproximarse al modo de expresarse del hombre, alejándose del lenguaje máquina. Esto es posible gracias a la forma racional con la

que se manejan las abstracciones que representan las entidades del dominio del problema, y a propiedades como la jerarquía y el encapsulamiento.

El elemento básico de este paradigma no es la función (elemento básico de la programación estructurada), sino un ente denominado *objeto*.

Un sistema orientado a objetos (SOO) es donde todos los datos se crean como instancias de una clase, la cual, encapsula propiedades y operaciones. Aquí todas las estructuras se basan en la creación de objetos.

Objeto

Es cualquier cosa real o abstracta en la cual conjuntamos datos y métodos que controlan dichos datos. Se puede decir que un objeto es un modelo de una parte de la realidad, generado dinámicamente en tiempo de ejecución dentro de un espacio o región de memoria, como una instancia de una clase en particular. El *tiempo de vida* o duración de un objeto en un programa siempre está limitada por el tiempo. La mayoría de los objetos sólo existen durante una parte de la ejecución del programa. Los objetos son creados mediante un mecanismo denominado *instanciación*, y cuando dejan de existir se dice que son *destruidos*. Los objetos se comunican entre sí por medio de mensajes. Un mensaje del objeto A al objeto B se usa para activar la ejecución de un método del objeto B. Un objeto posee como características inherentes: identidad, estado y comportamiento, como un modelo del mundo real.

A continuación se definen las propiedades más importantes de un objeto.

Identidad. Cada objeto en el sistema es distinto de cualquier otro, por medio de un identificador único, el cual debe ser generado por el sistema. No debe variar durante el ciclo de vida del objeto, y no es visible al programador o usuario final.

Estado. Todo objeto posee un *estado*, definido por sus atributos o un subconjunto de éstos, con él se definen las propiedades del objeto en un momento determinado de su existencia.

Comportamiento. Todo objeto presenta una interfaz, definida por sus métodos, por medio de la cual los objetos que componen los programas

pueden interactuar con él. Sólo los métodos del objeto *deben* modificar las variables de éste. A los métodos se les suele llamar también funciones miembro.

Clase

Una clase se puede obtener a partir de una agrupación de objetos basada en características comunes. En un desarrollo orientado a objetos, una clase define lo que cada uno de sus objetos puede hacer. La clase también se puede definir como un conjunto de cosas físicas y/o abstractas que tienen el mismo comportamiento y características.

Una clase es un tipo de dato definido por el usuario, define un *patrón* a partir del cual se crean los objetos. El *patrón* contiene una *descripción* general que es compartida por uno o varios objetos. La *descripción* incluye la definición de los datos y las operaciones asociadas a los objetos de la clase. Cuando se crea un objeto (instanciación) se ha de especificar de qué clase es el objeto, para que el compilador comprenda sus características.

Por medio de las clases se representa el estado de los objetos mediante variables denominadas atributos. Cuando se instancia un objeto se crea en memoria un espacio para los atributos del objeto.

Los métodos son funciones que representan el comportamiento de los objetos. Dichos métodos deben modificar los valores de los atributos del objeto, y representar las capacidades del objeto (en muchos textos se les denomina *servicios*).

Características de la Programación Orientada a Objetos

Encapsulación. El encapsulamiento permite a los objetos elegir qué información es mostrada y qué información debe ser ocultada al resto de los objetos. Es decir, la encapsulación es un mecanismo que permite ocultar al exterior, aquello que no contribuye directamente a las características esenciales del objeto. El encapsulamiento evita la comprensión de la implementación del objeto, proporcionando una interfaz, por lo que se reduce la complejidad de su utilización.

Herencia. La herencia expresa relaciones de tipo generalización/especialización, es decir, que es la relación donde una clase adquiere características y comportamiento de otra clase (herencia simple) o varias clases (herencia múltiple).

Conforme se utiliza dentro de una aplicación, la relación de herencia genera una jerarquización que se representa como un conjunto de categorías de abstracciones. Esto da lugar a los denominados árboles de herencia. Se considera que una arquitectura orientada a objetos bien estructurada esta formada de estos árboles.

Mediante la herencia una clase hija puede tomar determinadas propiedades de una clase padre. Así se simplifican los diseños y se evita la duplicación de código al no tener que volver a codificar métodos ya implementados.

Polimorfismo. Polimorfismo quiere decir un *objeto* y *muchas formas.* Esta propiedad permite que un objeto presente diferentes comportamientos en función del contexto en que se encuentre. El polimorfismo es un concepto de la teoría de tipos, en el que la declaración de un nombre determinado puede denotar instancias de varias clases diferentes, siempre y cuando se encuentren relacionadas mediante una superclase común. Cualquier objeto que se denote por tal nombre puede responder a un cierto conjunto de requerimientos y operaciones de diferentes maneras.

La sobrecarga (overload) de funciones se expresa como la designación de un solo nombre que puede hacer referencia a diferentes funciones en un mismo entorno. Esto significa que un nombre puede especificar varias operaciones que comparten características semejantes en su comportamiento, pero que en realidad su implementación es diferente.

En el momento en que el nombre es usado, la función correcta es seleccionada mediante comparación de tipos de los argumentos actuales con los tipos de los argumentos formales.

Serialización. La serialización permite a los objetos ser convertidos en una secuencia de bytes. La secuencia de bytes puede ser almacenada en un archivo, una base de datos, enviarla a una máquina remota, etc.

3.2. Modelo de Datos Orientado a Objetos

Un modelo de datos, es una organización lógica de objetos (entidades) del mundo real, con sus restricciones y relaciones entre ellos. Un lenguaje de base de datos es una sintaxis concreta para un modelo de datos. Un sistema de base de datos implementa un modelo de datos.

La esencia del MDOO se basa en los conceptos del paradigma orientado a objetos. El MDOO se compone de:

1. Objeto e identificador de objeto,
2. Atributos y métodos,
3. Clase, y
4. La jerarquía de clases.

1. *Objeto e Identificador de Objeto (OID).* Cualquier entidad del mundo real, es modelado como un objeto, asociado con un OID único, usado para precisar la recuperación de un objeto.

El sistema asigna un OID, por lo que dos objetos no pueden ser creados con el mismo IDO. Es importante mencionar que este concepto no es equivalente a una *clave primaria*, ya que dicha clave está compuesta de valores ingresados por el usuario, mientras que un OID está asignado por el sistema. Al quitar un objeto del sistema, quita un OID del sistema y por definición el sistema nunca asignará a otro objeto este mismo OID durante una sesión.

2. *Atributos y métodos.* Cada objeto tiene un estado (conjunto de valores para los atributos del objeto) y un comportamiento (conjunto de métodos - código del programa - el cual opera con el estado del objeto). El estado y el comportamiento encapsulados en un objeto, son accesados o invocados desde afuera del objeto solo a través del paso de mensajes explícitos.

La terminología orientada a objetos usualmente se refiere a los atributos como variables de instancia; cuyo dominio puede ser de cualquier tipo, ya sean definidos por el usuario o primitivos.

Los objetos también usan métodos, por medio de los cuales permiten que los datos dentro del objeto sean accedidos.

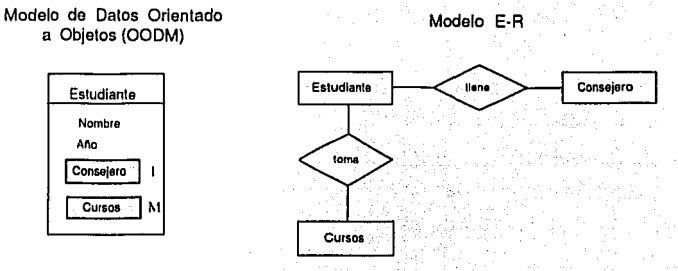
3. *Clase.* Ya definido anteriormente

4. *La jerarquía de clases.* En la realidad, es difícil concebir un objeto aislado, siempre es posible encontrar o definir algún tipo de relación entre objetos. En general, un conjunto de objetos organizados bajo ciertas reglas forman una jerarquía y, mediante su identificación dentro de un diseño, el proceso de comprensión de un problema se facilita. Es posible distinguir en la POO algunas relaciones útiles entre clases de objetos. Considerar cada una o su combinación permiten realizar de cierta forma una jerarquización. Tales relaciones son principalmente: herencia, agregación, asociación y uso.

Características del MDOO

Un MDOO debe soportar como mínimo las siguientes características:

1. El MDOO debe ser capaz de proveer representación de objetos complejos.
2. El MDOO debe ser extensible, es decir, debe proveer soporte para definir nuevos tipos de datos y métodos que sean capaces de operar con este nuevo tipo de dato.
3. El MDOO debe soportar encapsulación o abstracción de información. Debe ser capaz de ocultar cómo los datos se representan dentro del objeto y cómo se implementa cada método de otros objetos y otras entidades fuera de sí mismo.
4. El MDOO debe soportar herencia. Los objetos existirán en una relación de jerarquía.
5. El MDOO debe soportar OID's.



3.23. MDOO vs Modelo ER

En la figura (3.23) se compara el modelo MDOO con el modelo E-R. Ambos modelos están representando un `Estudiante` quien **tiene** un `Consejero`

y toma `Cursos`. El MDOO muestra la clase `Estudiante`, la cual define las características que debe tener un objeto de tipo `Estudiante`, y define los atributos `Nombre` y `Año`, una relación con un objeto de tipo `Consejero` y una colección de objetos de tipo `Curso`.

Por su parte, el modelo E-R define tres tablas de entidades: `Estudiante`, `Consejero` y `Cursos`. Además el modelo E-R debe tener dos tablas de relaciones para reunir la información de manera que tenga sentido. Como se puede ver, la técnica del modelo E-R necesita de dos entidades más para proveer las relaciones que el MDOO sugiere *implícitamente*.

Entidades y tablas. La idea de clase del MDOO, se asemeja a la idea de entidad del modelo ER. Las clases del MDOO son más potentes que el concepto de conjunto de entidades o tablas del modelo ER. Una clase no solo describe la estructura de datos, sino que también describe el comportamiento de los objetos de esa clase; características como métodos, los cuales permiten la descripción del comportamiento de los objetos, dan a una BDOO capacidades completas para Tipos de Datos Abstractos (TDA) permitiendo un incremento en la semántica del objeto que se está modelando. El TDA permite el soporte de encapsulamiento y herencia.

Acceso a datos. Tradicionalmente, el acceso a datos puede verse como un modelo de almacenamiento de dos niveles. Se necesita un vínculo entre la memoria principal de la computadora y el dispositivo de almacenamiento secundario, para acceder a los datos contenidos en la base de datos. El usuario necesita los datos para analizarlos y usarlos. En la figura (3.24) se puede ver que para obtener los datos a través de un manejador de bases de datos relacional, el usuario realiza una consulta mediante lenguaje SQL, éste accede a los datos en un medio de almacenamiento secundario, posteriormente los transforma y controla a tipos de datos compatibles al lenguaje destino, ya que SQL es utilizado por lo general como un lenguaje incrustado.

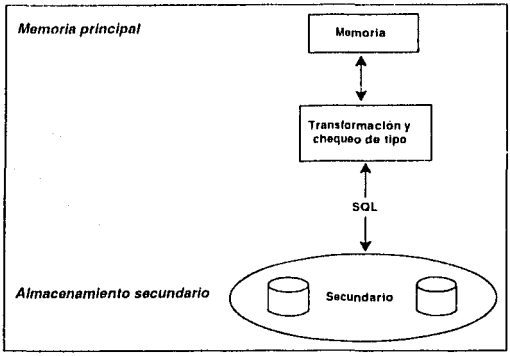
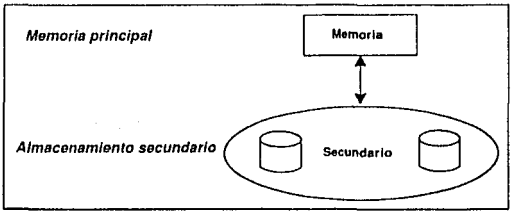


Figura 3.24. Modelo de memoria ER

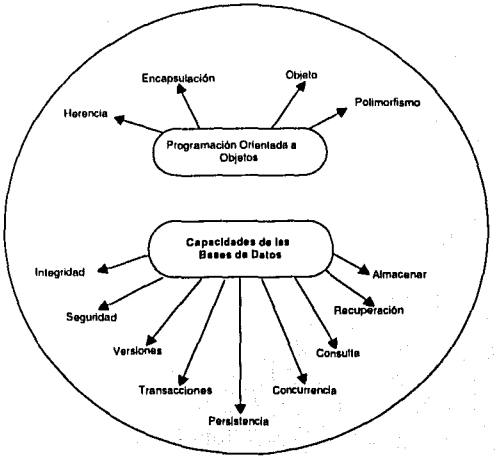
Las BDOO (ver Figura (3.25)) eliminan la vista de almacenamiento en dos niveles y brindan una vista de un solo nivel. Una BDOO *no necesita* SQL o una fase de transformación/control de tipos para traer los datos desde el almacenamiento secundario a la memoria. El usuario entonces instancia un objeto. Este objeto es el vínculo desde la memoria a la información en el almacenamiento secundario. El usuario interactúa con el objeto el cual está en memoria para recibir los datos desde el almacenamiento secundario.



3.25. Modelo de memoria de BDOO

Una BDOO es una base de datos formada por objetos y manejada por un SMBDOO. En la figura (3.26) se muestran las características de la programación orientada a objetos (POO) y las BD, cuando se integran ambas

características, el resultado es una BDOO. Las BDOO están diseñadas para trabajar con lenguajes de POO como JAVA y C++.



3.26. POO y Bases de Datos

Ventajas de las BDOO sobre el modelo tradicional. Entre las ventajas más ilustrativas de las BDOOs esta la flexibilidad y el soporte para el manejo de tipos de datos complejos.

- *Extensibilidad.* Una aplicación no está limitada a los tipos de datos predefinidos.
- *Herencia.* Permite la creación de objetos más complejos a partir de otros más simples (reusabilidad).
- *Verificación de tipos.* No se tiene el riesgo de tener datos que no correspondan a un dominio por su verificación estricta de tipos.
- *Desempeño.* Dado que se utilizan las estructuras de almacenamiento y acceso más adecuadas esto resulta más eficiente que el simple uso de archivos secuenciales con índices.

- *Mejora en la productividad de desarrolladores.* Esto es una consecuencia del soporte de los conceptos de orientación a objetos (menor cantidad de errores, menor tiempo de desarrollo, etc.).

- *Seguridad, recuperación, concurrencia a nivel objeto.* Esto permite que la operación sea más fácil dado que la unidad de trabajo es el objeto y no el archivo.

- *Integridad:* Las verificaciones de la integridad son más fáciles.

Desventajas de las BDOO. La inmadurez del mercado de BDOO constituye una posible fuente de problemas. La segunda desventaja es la falta de estándares en la industria orientada a objetos. Sin embargo, el *Object Management Group* (OMG), es una organización internacional de proveedores de sistemas de información y usuarios que se dedicada a promover estándares para el desarrollo de aplicaciones y sistemas orientados a objeto en ambientes de cómputo distribuidos.

3.3. Sistema Manejador de Bases de Datos Orientado a Objetos

Un SMBDOO es un SMBD con un modelo de datos lógico orientado a objetos (MOO). Un SMBD proporciona las siguientes facilidades:

- Generalidad de aplicación. Un SMBD es capáz de manejar datos de diferentes áreas de la aplicación.

- Acceso a datos eficientemente. Los datos son almacenados de tal manera que pueda haber acceso a ellos tan rápido como sea posible.

- Seguridad. Hay mecanismos de privacidad asegurándose contra el acceso a datos no autorizados o al mal uso. Algunos usuarios podrán modificar datos y otros solo consultarlos.

- Consistencia de datos. Cualquier cambio que se haga a los datos será organizado de tal manera que la base de datos siempre mantenga la consistencia.
- Tolerancia a fallas. Si el hardware o el software fallara mientras la base de datos está corriendo, nuncá harán a la base de datos inconsistente, a lo mucho pocos de los cambios recientes se perderán si el sistema sufre fallas.
- Control de concurrencia. Varios usuarios pueden accesar a la base de datos al mismo tiempo sin interferirse.
- Vistas. Proveer diferentes o más representaciones a diferentes tipos de usuario. Así, una base de datos podrá ser accesada usando una vista detallada para el personal que trabaja intensivamente con un conjunto de datos, mientras un manejador puede ver una vista menos detallada.
- Interfaz de usuario. Se proveerá una interfaz de usuario apropiada para permitir a la base de datos ser diseñada y usada por una amplia variedad de usuarios.
- Integridad. La facilidad para imponer restricciones de integridad que limitan los valores que la base de datos puede tomar, manteniendo el sentido de la información almacenada.
- Distribución. Es posible para los SMBD manejar datos que son distribuidos a través de más de una computadora en una red local o incluso a través de un número de sitios unidos en redes extensas.

Los SMBDOO deben soportar toda la funcionalidad de un SMBD más:

- Interfaz de al menos un lenguaje de programación orientado a objetos.
- Tipos definidos por el usuario.
- Identidad de los objetos.

- Redes de objetos.
- Optimización de referencia de ubicación.

Pros y Contras de los SMBDOO

Pros

- Resuelven muchos de los problemas que los sistemas tradicionales no pueden. La cantidad de información que puede modelarse con un SMBDOO se incrementa.
- Brindan objetos complejos que permiten fácilmente integración de multimedia, CAD, y otras bases de datos especializadas.
- Tienen capacidades de modelado por medio de la extensibilidad. Con un SMBDOO uno puede agregar más capacidades de modelado, permitiendo así modelar sistemas más complejos.
- Una ventaja importante de los SMBDOO sobre los SMBDR es que no tienen impedancia de correspondencia. La *impedancia de correspondencia* es cuando se necesitan mapear objetos usados en la aplicación a tablas almacenadas en una base de datos relacional (ver Figura XXVII*a*). En un SMBDOO los objetos se almacenan directamente sin ningún mapeo a las diferentes estructuras de datos. La figura XXVII*b* presenta cómo un SMBDOO no usa impedancia de correspondencia y los SMBD Relacionales si la usan.

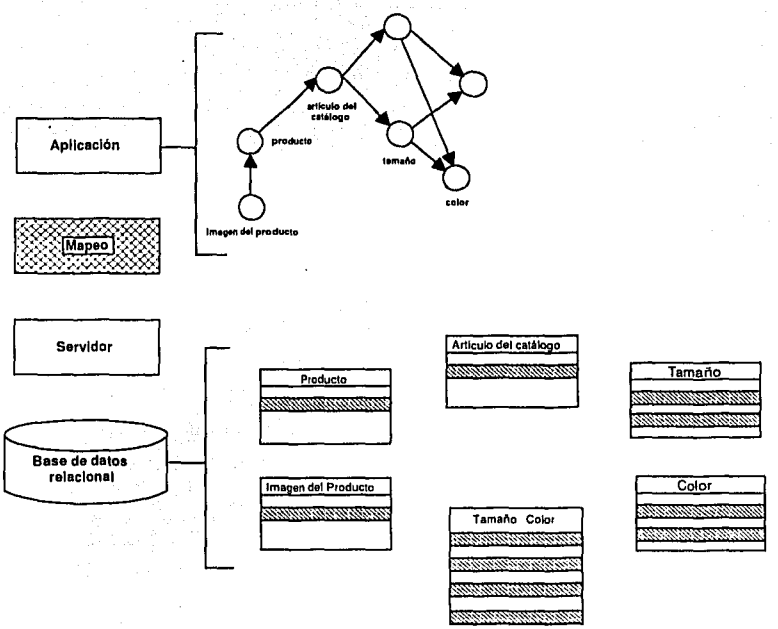


Figura 3.27(a). Impedancia de correspondencia en un SMDBR

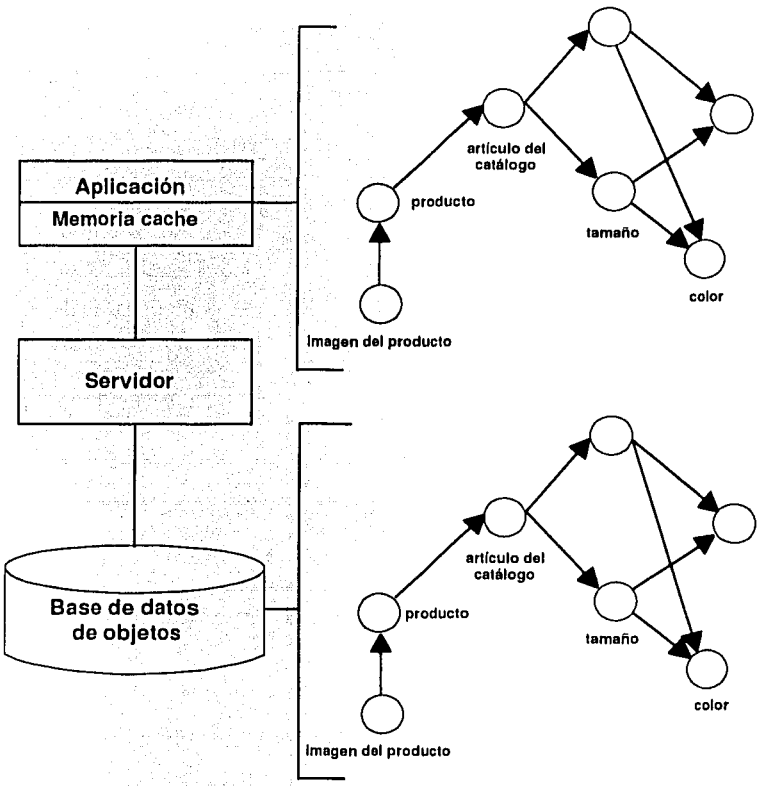


Figura 3.27(b). Almacenamiento en un SMBDOO

Contras

- La migración de sistemas[2].
- La falta de estandarización de modelos orientados a objetos.
- Es una tecnología inmadura, todavía se desarrollan nuevas técnicas.
- EL modelo OO requiere más conocimiento del analista que los basados en registros.

Características de un SMBDOO

El Manifiesto[3] de 1990 de Kioto Japan, es el primer intento de describir una norma en la cual deberían basarse los SMBDOO. Este documento describe las características que un sistema debe tener para calificar como un SMBDOO. Estas características se separan en tres grupos. (Ver figura (3.38))

I) *Obligatorias.* Son las que el sistema debe satisfacer para ser llamado un SMBDOO. Un SMBDOO debe satisfacer dos *criterios obligatorios*:

a) Debe tener un SMBD

b) Debe ser un sistema OO

El *primer criterio* se traduce en cinco características:

1. Persistencia.
2. Manejo de almacenamiento secundario.
3. Concurrencia

[2]Si se tiene una base de datos relacional y se quiere migrar a una orientada a objetos, es difícil o imposible, ya que el Modelo ER no es tan descriptivo como el MDOO

[3]Escrito por M. Actkinson, F. Bancilhon, D. DeWitt, K. Dittrich y otros, para la primer conferencia internacional de bases de datos deductivas y orientadas a objetos

4. Recuperación
5. Facilidad de consultas.

El *segundo criterio* se traduce en ocho características:

1. Objetos complejos.
2. Identidad de objetos
3. Encapsulación
4. Tipos o clases
5. Herencias
6. Derogación combinada con ligado tardío (overriding and late binding)
7. Extensibilidad.
8. Completación funcional.

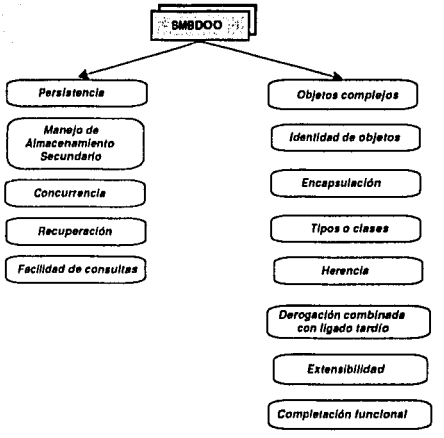


Figura 3.28. Características de un SMBDOO

Enseguida se describirán las cinco características del primer criterio que debe satisfacer un SMBDOO.

La *persistencia* por su importancia y extensión se aborda más adelante.

El *manejo de almacenamiento secundario*, es usualmente soportado a través de un conjunto de mecanismos, que son: manejo de índices, agrupación de datos, selección del camino de acceso y optimización de consultas.

La *concurrencia* permite que múltiples usuarios interactúen con el sistema al mismo tiempo, el sistema debe asegurar que múltiples usuarios puedan trabajar simultáneamente en la base de datos. El sistema debe por lo tanto soportar la atomicidad de una secuencia de operaciones y la compartición controlada.

La *recuperación* se refiere a fallas que pueda haber en el hardware o software, con lo que respecta al software el sistema debe recuperar el estado de los datos y con respecto al hardware que generalmente se refiere a fallas del suministro eléctrico el sistema debe conservar la consistencia.

La *facilidad de consultas ad hoc* permite solicitar un resumen de toda o parte de la base de datos, por ejemplo, consultar todas las personas que son mayores de 30 años. Un SMBDOO también maneja las consultas navegacionales, es decir, trabaja empezando desde una parte particular de los datos moviéndose a los datos que se requieren. Por ejemplo, se tiene una referencia a Juan, se quiere consultar a cerca de la compañía donde trabaja Juan.

Enseguida se describen algunas características del segundo criterio que debe satisfacer un SMBDOO.

Los SMBDOO manejan tipos de *objetos* como son: `integers`, `floats`, `caracter`, etc. También manejan tipos de *objetos complejos* como son: `Tuples`, `Sets`, `Bags`, `Lists`, etc.

La *derogación y ligado tardío* son técnicas para lograr el polimorfismo basado en la herencia. La *derogación* se refiere a la posibilidad de redefinir métodos en la jerarquía de clases, la derogación permite a una subclase

redefinir métodos que hereda de su superclase. Los métodos derogados que aparecen en las subclases, tienen el mismo nombre, la misma lista de parámetros y el mismo tipo de regreso que en la superclase. En los modificadores de acceso, si el método de la superclase es público el método derogado debe serlo también; si el método de la clase es protegido, el método derogado puede ser protegido o público; si el método de la superclase es privado, éste no es heredado y por tanto no es asunto de la derogación. El *ligado tardío* significa demorar la asociación entre el nombre de un método y su implementación en tiempo de ejecución (run-time).

Extensibilidad. El sistema de base de datos viene con un conjunto de tipos predefinidos. Estos tipos pueden ser usados por los programadores para escribir sus aplicaciones. Este tipo de conjuntos debe ser extensible en el siguiente sentido: hay una manera de definir tipos nuevos y no hay distinción en usar entre los tipos definidos por el sistema y los definidos por el usuario. Por supuesto que puede haber una fuerte diferencia en la manera en que los tipos definidos por el sistema y el usuario son soportados por el sistema, pero esto debe ser transparente para la aplicación del programador.

La *completación funcional* desde el punto de vista de un lenguaje de programación significa que se puede expresar cualquier función, usando el Lenguaje de Manipulación de Datos (LMD).

II) *Opcionales.* Las que pueden ser añadidas para hacer el sistema mejor, las cuales no son obligatorias. Estas son:

   a) Herencia múltiple
   b) Chequeo de tipos
   c) Distribución de inferencia
   d) Diseño de transacciones, y
   e) Versiones

A continuación se describen estas características opcionales.

Los SMBDOO pueden o no utilizar *herencia múltiple.*

En el *chequeo e inferencia de tipos*, el grado de chequeo de tipos que el sistema lleva a cabo en tiempo de compilación se deja abierto para que el compilador decida. La situación óptima es donde un programa el cuál fue aceptado por el compilador no pueda producir cualquier tipo de error en tiempo de ejecución. La cantidad de inferencia de tipos se deja abierto también al diseñador del sistema, la situación ideal es donde los tipos base tienen que ser declarados y el sistema deduce los tipos temporales.

La *distribución* de inferencia es una característica ortogonal para los SOO. De esta manera, el sistema de base de datos puede ser distribuido o no.

El *diseño de transacciones.* En la mayoría de las aplicaciones, el modelo de transacciones clásico de un SMBDOO a veces no es satisfactorio, ya que las transacciones tienden a ser muy grandes y el criterio de serialización no es adecuado, de este modo, muchos SMBDOOs soportan diseño personalizado de transacciones (transacciones grandes o transacciones anidadas) y serialización.

Las *versiones* se refieren a que los SMBDOO soportan múltiples versiones de objetos, como se presenta en la figura (3.29).

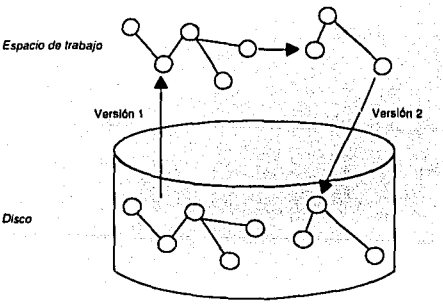


Figura 3.29. Ejemplo de versiones

III) *Abiertas.* Estas características comprenden los siguientes puntos:

- La base de datos puede ser distribuida.
- La representación del sistema está definida por un conjunto de tipos atómicos y constructores. Aunque se den un conjunto mínimo de estos tipos y constructores (tipos elementales del lenguaje de programación, y constructores de conjuntos, tuplas y listas) disponibles para describir la representación de los objetos, se pueden extender.
- Uniformidad que debe presentar el sistema, por ejemplo el diseño de la estética de las aplicaciones debe ser similar.

3.4. Persistencia

La *persistencia* es una característica importante del primer criterio en un SMBDOO, ya que es un mecanismo que debe soportar el movimiento de clases y objetos entre la memoria y el almacenamiento secundario. Las principales cuestiones que surgen en un sistema persistente son:

- Distinguir los datos que van a ser almacenados de los que no.
- Acceso a los datos eficientemente.
- Manejo del código de la aplicación.
- Borrado de los objetos.

El término persistencia tiene dos significados:

- *Con respecto al dato.* El periodo de tiempo durante el cual un dato existe.
- *Con respecto a la información del sistema.* La capacidad del sistema para almacenar sus datos o para prolongar la persistencia de sus datos más allá de la sesión en que fueron creados.

Los lenguajes de programación están enfocados al manejo de *datos temporales* y las bases de datos al manejo de *datos persistentes*. Un *sistema persistente* hace perdurar sus datos después de la ejecución del programa en que fueron creados. El término persistencia ortogonal es el más usado para describir sistemas persistentes atendiendo a dos características, que son:

a) Se almacena el dato con la misma estructura que tenía en memoria.

b) No importa el valor del dato para que sea persistente o temporal.

Existen tres tipos de persistencia:

1. Persistencia de sesión
2. Persistencia de archivos
3. Persistencia ortogonal.

1. *Persistencia de sesión.* En la figura (3.30) se ve cómo el área de trabajo, la cual es el espacio de trabajo de la aplicación es copiado a disco. Permite almacenar el espacio de trabajo de la aplicación al finalizar la sesión, donde la aplicación y los datos quedan juntos.

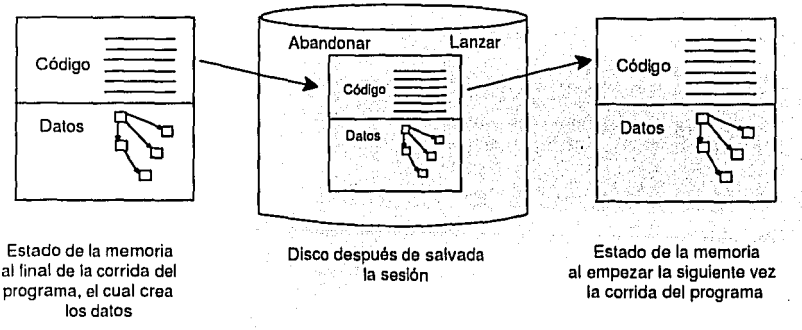


Figura 3.30. Persistencia de sesión

El usuario no tiene control sobre que es o no almacenado, todo en el grupo de trabajo es guardado sin contemplar si los datos son valiosos o relevantes. Los datos almacenados no se pueden compartir entre los usuarios.

2. *Persistencia de archivos.* La aplicación puede escribir datos a uno o más archivos durante su ejecución, al finalizar el programa o cuando una operación de salvar de la aplicación es invocada, los datos son transformados dentro de una estructura conveniente para ser almacenados en archivos y escritos en disco. La estructura de los datos en memoria es diferente a la que mantienen en disco.

La figura (3.31) ilustra la persistencia de archivos, donde los datos son separados del código y antes de que el programa pueda empezar, los datos deben ser leídos explícitamente; por lo que existe el inconveniente de que el programa debe tener funcionalidad para manejo de archivos. Otro problema es que los datos en el archivo están abiertos al acceso y pueden ser corrompidos.

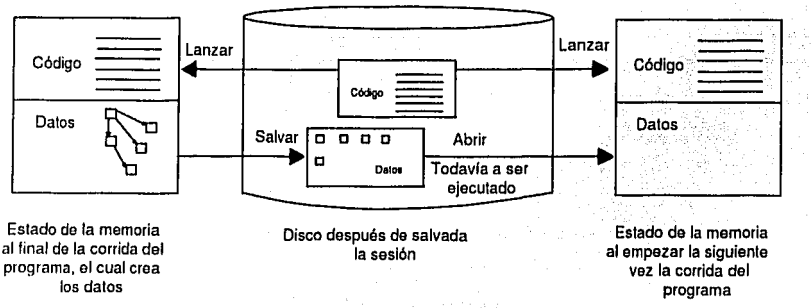


Figura 3.31. Persistencia de archivos

3. *Persistencia ortogonal.* La persistencia ortogonal presenta dos aspectos. Primero, la operación de `commit` almacena los datos de la misma forma en disco como estaban en memoria y pueden ser manipulados por el mismo código donde quiera que éste resida. Segundo, los datos en memoria son almacenados sin transformación.

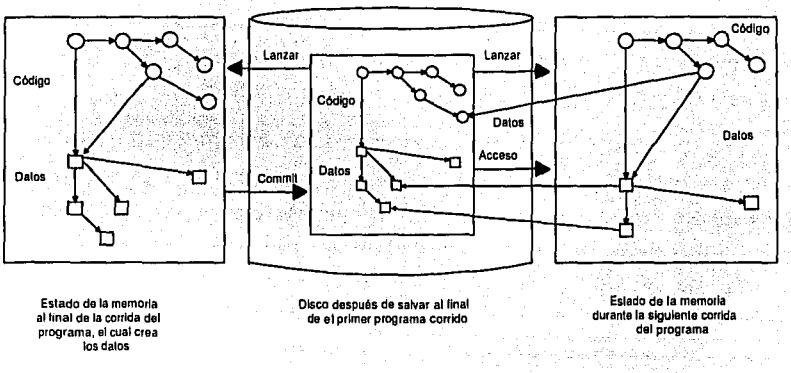


Figura 3.32. Persistencia ortogonal

En la figura (3.32) se muestra cómo una mezcla de código y datos pueden coexistir en memoria y en disco. La carga de módulos de datos y código se realiza tan pronto como el proceso los requiera. En la figura se ve que solo algunos de los módulos del código y datos han sido recuperados, dos de los datos y uno de los módulos del código no fueron traídos a la memoria, sin embargo, hay punteros a la localización en el disco donde están haciendo referencia a otros datos y código, por lo que pueden ser buscados cuando se requiera de ellos. Este modelo es más conveniente cuando se trabaja con BDOO. El modelo lógico es el tipo de sistema del lenguaje de programación, en los lenguajes orientados a objetos el tipo de sistema es la estructura de clases. Por lo tanto, el modelo de datos lógico de un SMBDOO es también la estructura de clases y la base de datos es una jerarquía de clases.

Identificando persistencia.

Se debe determinar en el sistema qué datos serán preservados y cuáles no, por lo que dos puntos se deben tomar en cuenta. Primero, la eficiencia del almacenamiento de los datos. Segundo, la cantidad de esfuerzo que el programador tiene que hacer para asegurar que los datos sean almacenados. En algunos sistemas concernientes a la eficiencia del almacenamiento, el programador puede indicar no sólo cuáles datos son almacenados sino cómo y dónde serán almacenados los datos. Existen varias formas en que el programador puede asignar persistencia a un objeto, y son:

1. Persistencia inferida
   - Persistencia por alcance
   - Raíz de persistencia
2. Persistencia explícita
   - Clases persistentes
   - Clases sombra persistentes
   - Clase raíz persistente
   - Persistencia declarada en la creación del objeto
   - Persistencia por almacenamiento explícito
   - Sistema provisto de raíces persistentes
   - Objetos raíz nombrados

La *persistencia por alcance* significa que, si un objeto es persistente, entonces cualquier otro objeto o literal al cual éste se refiera debe ser persistente también.

En la *raíz de persistencia* los objetos son persistentes explícitamente y asegura que cualquier objeto que pueda ser seguido por cualquier ruta de referencias desde una raíz de persistencia éste también es persistente. Los demás objetos que no puedan ser alcanzados desde la raíz de persistencia son *temporales*. Un tipo de raíz de persistencia usado comúnmente es el alcance de una clase.

La figura (3.33) muestra cómo trabaja la persistencia por alcance. El inciso (a), muestra algunos datos que han sido persistentes y una estructura

de datos temporales la cuál tiene una cabeza H y cuatro componentes. Los datos persistentes mostrados inician desde una raíz la cuál contiene una colección de elementos (mostrada como una línea vertical), uno de los cuales tiene tres componentes, incluyendo un entiquetado C. El inciso (b), muestra el efecto de traer C a memoria y hacer una referencia a H. El inciso (c), muestra lo que ocurre al final de la corrida del programa. C es transferido de regreso a el disco seguido por H, ya que es una referencia desde C a H. Además, los otros componentes de H también son movidos a disco, como se muestra en el inciso (d).

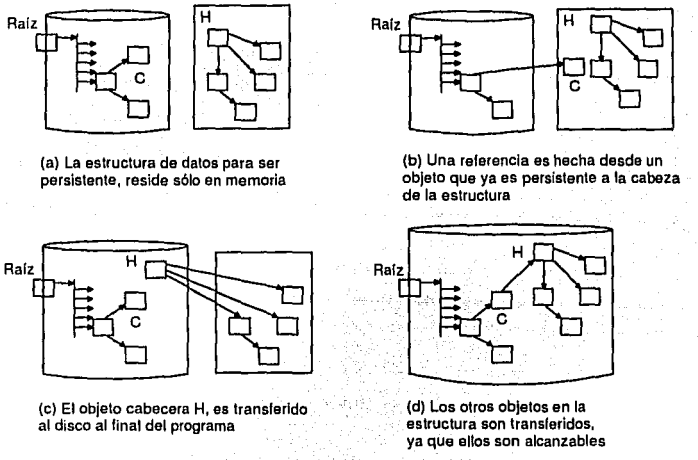


Figura 3.33. La persistencia por alcance

Los *beneficios* de la persistencia inferida son:

- Mantiene las restricciones de integridad referencial.
- Se reduce el esfuerzo del programador para indicar la persistencia.

Los *problemas* son:

- Si un objeto referenciado no es salvado. Por ejemplo, si una persona es almacenada junto con una referencia a la compañía para la cuál él o ella trabaja, pero el objeto compañía no es almacenado, entonces cualquier intento de acceso al nombre del empleado del objeto persona tendrá problemas en tiempo de ejecución.
- Puede causar que información no deseada sea almacenada.

La *persistencia explícita* cuenta con varias formas, y son:

- *Clases persistentes.* Cuando una clase es creada, esta es declarada clase persistente o no. Cualquier objeto creado para ser miembro de esta clase es automáticamente almacenado.
- *Clases sombra persistentes.* Por cada clase, se crea una versión persistente automáticamente. Sólo miembros de las versiones persistentes de las clases tienen persistencia automática.
- *Clase raíz persistente.* Algunos sistemas usan la jerarquía de clases para determinar cuáles clases son persistentes. Estos sistemas proporcionan una clase raíz la cual cuenta con todos los mecanismos para soportar la persistencia. Cualquier clase declarada como subclase de ésta clase también es persistente, ya que ésta heredará los mecanismos de persistencia, entonces cualquier instancia de la clase persistirá.
- *Persistencia declarada en la creación del objeto.* En lugar de declarar la persistencia en la clase, algunos sistemas permiten a cualquier objeto ser persistente, pero forzan al programador a declarar si el objeto al momento de la creación es persistente o no.
- *Persistencia por almacenamiento explícito.* Es similar a la persistencia de archivos en lenguajes tradicionales esto permite al programador instruir al sistema para almacenar objetos particulares. Se agregará al lenguaje un comando *store*, el cual incluirá cómo almacenar el objeto.

- *Sistema provisto de raíces persistentes.* Otro esquema es proporcionar ciertos objetos del sistema, para ser raíces de persistencia. Cualquiera de los objetos que se hacen componentes de este objeto raíz persistirán automáticamente.

- *Objetos raíz nombrados.* Facilidad para nombrar objetos específicos como raíces de persistencia, la cual puede darse en dos formas:

  - Una raíz específica para una base de datos
  - La raíz nombrada puede ser parte del esquema

Almacenamiento de código y datos

Un punto importante es cómo los datos y código de la aplicación están relacionados cuando son almacenados. Hay cuatro formas diferentes del almacenamiento, y son:

a) *Un lenguaje de programación con persistencia de archivos.* Separa el código de los datos, pero hay dificultad de mantener los archivos de datos organizados coherentemente, ya que tiene una estructura abierta y flexible y no facilita la integridad pérdida o corrupción de los datos.

b) *Un sistema de base de datos tradicional.* Son más organizados con respecto a los datos manteniendo su consistencia y el código queda fuera de su control, es decir fuera de la base de datos.

c) *Un lenguaje con persistencia ortogonal.* Toma al código como otro tipo de dato, así puede ser almacenado en la base de datos, recuperado, pasado como dato a otro programa. Sin embargo, no mantienen estructura para la organización de el código y los datos.

d) *Un SMBDOO.* El código y los datos no son tratados de la misma forma, sin embargo ambos son almacenados juntos en una estructura de jerarquía de clases.

En la figura (3.34) se muestran estas formas de almacenamiento, donde el código son los óvalos y los datos son los rectángulos. En el inciso (a), un *sistema de archivos* es un ambiente desprotegido pero flexible en el cual el código y los datos pueden ser mezclados libremente. En el inciso (b), un SMBD tradicional controla parte del sistema de archivos, el cual usa para almacenar y proteger los datos. El código de la aplicación permanece fuera de el control de el SMBD. En el inciso (c), un lenguaje con persistencia ortogonal permite al código ser protegido, los datos y el código puedan ser mezclados libremente. En el inciso (d), el SMBDOO proporciona una estructura de clases en la cual, la mezcla de código y datos es controlada.

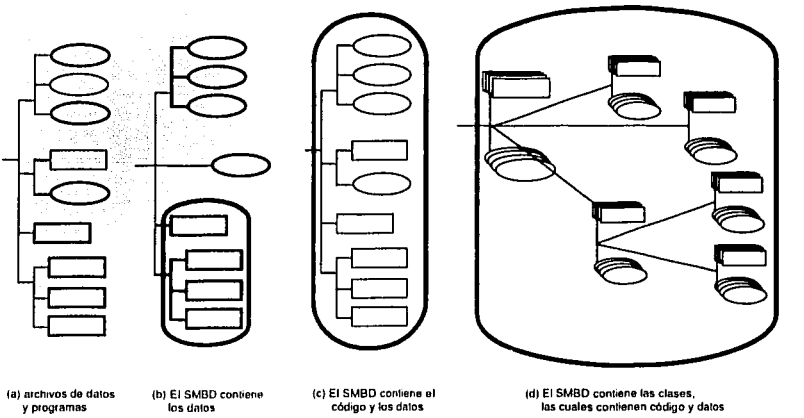


Figura 3.34. Almacenamiento de código y datos

Eliminando datos

Un SMBD debe proporcionar un servicio que permita eliminar los datos inútiles, ya sea de forma explícita o automática. Los sistemas de base de datos tradicionales proporcionan la eliminación explícita, sin embargo pueden violar la integridad de la base de datos. La BDOO es una compleja red de objetos relacionados entre sí y de diferentes tamaños.

Hay dos métodos de eliminación, y son:

a) *Eliminación explícita.* Algunos sistemas permiten la eliminación explícita de objetos y de clases, la forma para decirle al sistema cuáles datos borrar es la misma que para almacenamiento explícito de datos. Los lenguajes POO tienen mecanismos que impiden la violación de las restricciones de integridad.

b) *Eliminación automática.* Otra alternativa que los sistemas proporcionan es el uso de algunas formas de recolección de basura automática. El recolector de basura es usado por los SMBDOO, los cuales usan la persistencia por alcance. La figura (3.35) muestra lo que es basura y no lo es. El recolector de basura consta de dos fases a veces llamadas *mark-and-sweep*, y son:

   - *Identificación* de la basura
   - *Liberación* del área usada por objetos no deseados

La fase de *identificación* inicia con un recorrido recursivo por los datos, partiendo de las raíces de persistencia y marcando los objetos que alcanza, cualquier objeto no marcado es basura. La fase de *liberación* construye una lista de espacios libres o compacta las áreas de almacenamiento para crear espacios libres más grandes.

Otra técnica de eliminación automática consiste en el conteo de referencias, donde cada objeto debe mantener un contador de referencias a él y cada vez que se hace una nueva referencia, el contador se incrementa y si la referencia es removida el contador se decrementará. Los objetos con contador en cero son eliminados. La técnica es costosa en espacio y tiempo, por ejemplo en la figura (3.35) se presenta una estructura de objetos en forma de ciclo que no es alcanzada por una raíz de persistencia y si el contador de referencias no es cero, no podrá ser detectado como basura y el espacio que la estructura ocupa no será liberado. El conteo de referencia puede usarse como un recurso temporal. La recolección de basura es necesaria para liberar espacio de memoria y espacio de almacenamiento.

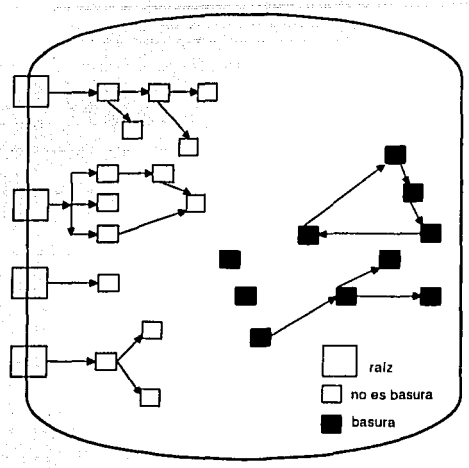


Figura 3.35. Recolección de objetos basura

3.5. Arquitectura de un SMBDOO

El Grupo ODMG (Object Data Management Group) creado en 1991 es un consorcio formado por vendedores de bases de datos dirigido por un pequeño y enfocado grupo de expertos. Su meta es producir un estándar para de SMBDOO.

Los principales componentes de la arquitectura ODMG para un SMBDOO son:

1. Modelo de objetos
2. Lenguaje de especificación de objetos
3. Lenguaje de consulta de objetos
4. Lenguaje de ligado (language binding)

3.5.1. Modelo de objetos

El modelo de objetos especifica los tipos de semánticas que pueden ser definidos explícitamente en un SMBDOO. Entre otras cosas, las semánticas del modelo de objetos determinan las características de los objetos, cómo pueden ser relacionados con los otros, y cómo pueden ser nombrados e identificados. El modelo de objetos ODMG permite que tanto los diseños como las implementaciones, sean portables entre los sistemas que lo soportan. Las construcciones que especifica el Modelo de Objetos soportadas por un SMBD son:

- Los componentes básicos de una base de datos orientada a objetos son *objetos* y *literales*. Un objeto es una instancia de una entidad de interés del mundo real y tiene un identificador único (OID). Una literal es un valor específico como `Adriana` o `20`, una literal no tiene que ser necesariamente un solo valor, puede ser una estructura o un conjunto de valores relacionados que se guardan bajo un solo nombre. Los literales no tienen identificador único.

- Los objetos y literales pueden ser categorizados por sus tipos. Cada tipo tiene un dominio específico (por ejemplo, el conjunto de propiedades) compartido por todos los objetos y literales de ese tipo. Cuando un tipo tiene comportamientos, todos los objetos de ese tipo comparten los mismos comportamientos. Un tipo puede ser una clase, una interfaz o una literal (por ejemplo, `integer`.)

- El estado de un objeto se define por los valores de sus propiedades (atributos y relaciones) que tienen con otros objetos. El estado actual de un objeto viene dado por los valores actuales de sus propiedades, los cuales pueden cambiar con el tiempo.

- Los objetos pertenecen a una jerarquía de objetos y las literales son similares a los tipos de un lenguaje.

- Una base de datos permite compartir objetos a múltiples usuarios y aplicaciones. Una base de datos se construye a partir de su esquema en Lenguaje de Definición de Objetos (LDO) y está poblada por instancias de los tipos allí definidos.

La figura (3.36) muestra una representación jerárquica de tipos y componentes del modelo de objetos.

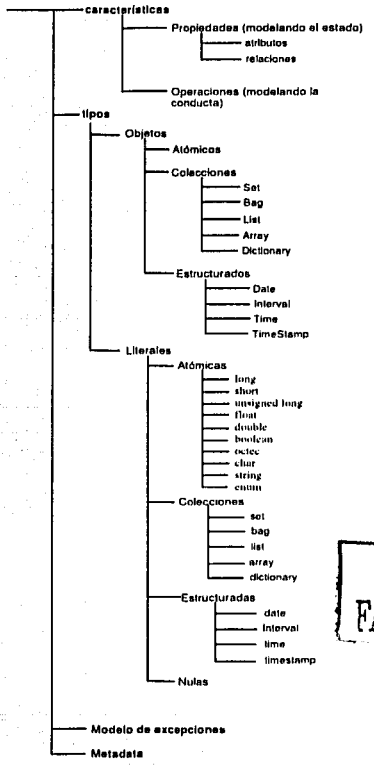


Figura 3.36. Modelo de objetos

En seguida se explica cada uno de los componentes del modelo de objetos.

Tipos. La definición de un tipo consta de dos aspectos:

1. *Especificación externa o interfaz.* Describe la apariencia externa del tipo, las propiedades que tiene, las operaciones disponibles y los parámetros de las operaciones. Éstos son aspectos de los tipos visibles a los usuarios.

2. Una o más *implementaciones.* La implementación de tipos define los aspectos internos de los objetos. Consta de la estructura de datos que implementa las propiedades y el código de las operaciones. La implementación consta de dos partes:

   - Representación. La representación es una estructura de datos dependiente de un lenguaje de programación que contiene las propiedades del tipo. Las especificaciones de la implementación vienen de una conexión con un lenguaje (language binding). Cada lenguaje de ligado define un mapeo para la implementación de tipos literales.
   - Métodos. Los detalles de las operaciones de un tipo se especifican mediante un conjunto de métodos, los cuales se escribirán en el mismo lenguaje de programación utilizado para expresar la representación del tipo.

La separación entre especificación e implementación es la forma que tiene el Modelo de Objetos de reflejar la encapsulación. El LDO es usado para precisar las especificaciones externas de los tipos en el modelo de objetos de la aplicación.

- Por cada propiedad en el estado el lenguaje de ligado define una variable de instancia de tipo apropiado.
- y por cada una de las operaciones en la conducta define un método.

Objetos. Los tipos de objetos se descomponen en:

1. atómicos,
2. colecciones, y
3. tipos estructurados.

Objetos atómicos. Un tipo de objeto atómico es definido por el usuario. No existen objetos atómicos pre-construidos en el modelo ODMG.

Objetos colecciones. Las instancias de las colecciones de objetos pueden ser de tipo atómico, colección o literal. Además, todos los elementos de la colección deben ser del mismo tipo. Los objetos colección identificados por el estándar ODMG son:

- `Set<tipo>`, es un grupo desordenado de objetos del mismo tipo que no permite duplicados.
- `Bag<tipo>` es un grupo desordenado de objetos del mismo tipo que acepta duplicados.
- `List<tipo>`, es un grupo ordenado de objetos del mismo tipo que acepta duplicados.
- `Array<tipo>`, es un grupo ordenado de objetos del mismo tipo que se pueden acceder por su posición. Su tamaño es dinámico y los elementos se pueden insertar y borrar de cualquier posición.
- `Dictionary<tipo,valor>`, es una secuencia desordenada de pares (llave,valor) sin llaves duplicadas. Cada llave es una instancia de: *struct Association{any key; any value;}*. La iteración sobre un diccionario resulta en la iteración sobre una secuencia de asociaciones.

Objetos estructurados. Los objetos estructurados definidos en el modelo de objetos ODMG son:

- `Date`, es una fecha del calendario (día, mes y año)

- `Interval`, representan una duración de tiempo y se usan para realizar algunas operaciones sobre objetos *Time* y *Timestamp*
- `Time`, denota tiempo específicamente en un tipo horario, es decir es una hora (hora, minutos y segundos)
- `Timestamp`, consta de un objeto Time y un Date encapsulados.

Estos tipos de objetos estructurados están definidos en la especificación ANSI-SQL.

Literales. Los literales no tienen identificadores y no pueden aparecer como objetos, sino que están embebidos en objetos; por lo que no pueden referenciarse de modo individual. Los tipos literales que ODMG soporta son:

1. atómicas,
2. colección,
3. estructuradas, y
4. nulas

Literales atómicas. El modelo de objetos soporta los siguientes tipos:

- `long`
- `short`
- `unsigned long`
- `unsigned short`
- `float`
- `double`
- `boolean`
- `octet`

- `char (character)`
- `string`
- `enum (enumeration)`

Colección de literales. Los elementos de estas colecciones pueden ser literales u objetos. Existe el tipo Table equivalente a una colección de estructuras. El modelo de objetos soporta los siguientes tipos:

- set<tipo>
- bag<tipo>
- list<tipo>
- array<tipo>
- dictionary<tipo>

Literales estructuradas. Ésta es una estructura simple formada por un número fijo de elementos con nombre y que puede ser literal u objeto. Los tipos de estructuras soportados por el modelo de objetos son:

- date
- interval
- time
- timestamp

Literales nulas. Para cada tipo de literal existe otro tipo de literal que soporta un valor nulo, como float o set<> que tienen su `nullable_float` y `nullable_set` respectivamente. Los tipos concretos son directamente instanciables, pero las colecciones no. La semántica de `null` es la misma que soporta SQL-92.

Propiedades (Modelando el estado). El modelo de objetos ODMG define dos tipos de propiedades: *atributos* y *relaciones*.

1. *atributos*. Un atributo se define del tipo de un objeto. Un atributo no es un objeto de *primera clase*, por lo tanto no tiene identificador, pero toma como valor un literal o el identificador de un objeto. No es posible definir atributos de atributos o relaciones entre atributos.

2. *relaciones*. Las relaciones se definen entre tipos. El modelo actual sólo soporta relaciones binarias con cardinalidad **1:1**, **1:n** y **n:m** Una relación no tiene nombre y tampoco es un objeto de *primera clase*, pero define caminos transversales en la interfaz de cada dirección. La integridad de las relaciones las mantiene automáticamente el SMBDOO.

Un atributo también puede ser object-valued (tener como valor un objeto). Esto permite que un objeto referencíe a otro sin una ruta transversal ni la integridad referencial. El atributo objeto-valuado se conoce también como composición.

Operaciones (Modelando la conducta). La conducta de un tipo está especificada como el conjunto de firmas de sus operaciones. No existen las operaciones independientes de un tipo, ni las operaciones definidas para más de un tipo.

Tipos diferentes pueden tener operaciones con el mismo nombre, dando origen al *operation dispatching*. Esto es, verificar qué objeto está haciendo referencia a ese método.

Modelo de excepciones. Las operaciones pueden originar excepciones y éstas pueden comunicar resultados. En ODMG, las excepciones son objetos con interfaz que permite relacionarlas con otras excepciones en una jerarquía generalización-especialización.

Metadata. Es la información descriptiva a cerca de los objetos persistentes que definen el esquema de un SMBDOO. El metadata es utilizado por el SMBDOO para definir la estructura del almacenamiento de los objetos.

El metadata esta almacenado en el contenedor del esquema del Lenguaje de definición de objetos.

3.5.2. Lenguajes de Especificación de Objetos

Los dos lenguajes de especificación son:

1. *Lenguaje de Definición de objetos-LDO (ODL-Object Definition Language).* El LDO debe ser empleado para definir, no sólo los tipos, los atributos de los tipos, los dominios de los atributos, y las restricciones sobre los atributos o tipos; sino que debe admitir también métodos, datos compuestos, herencia, etc.

LDO es el equivalente del LDD (Lenguaje de Definición de Datos) de los SMBD tradicionales, está pensado para definir tipos de objetos que pueden ser implementados en una variedad de lenguajes de programación. Por lo tanto, LDO no está atado a la sintaxis de un lenguaje de programación. Esta portabilidad es necesaria para que una aplicación sea capaz de correr con mínimas modificaciones en una variedad de SMBDOOs.

2. *Formato de Intercambio de Objetos-FIO (OIF - Object Interchange Format).* El FIO es un lenguaje de especificación usado para descargar y cargar el estado actual de un SMBDOO o de un archivo o un conjunto de archivos. El FIO puede usarse para el intercambio de objetos persistentes entre SMBDOOs, datos fuente, documentación proporcionada, y suites de manejo de prueba.

Los siguientes puntos han guiado al desarrollo de el FIO:

- El FIO debe soportar todos los estados en un SMBDOO de acuerdo al Modelo de Objetos de ODMG y la definición del LDO.
- El FIO no debe ser un lenguaje de programación completo.
- El FIO no necesita otras palabras que el tipo, atributo, e identificadores de relaciones provistos con la definición del LDO y un SMBDOO.

Estados de un SMBDOO. Los estados de los objetos contenidos en un SMBDOO son:

- Identificadores de objetos
- Ligado de tipos
- Valores de los atributos
- Enlaces a otros objetos

Cada uno de estos puntos son especificados dentro del FIO.

Estructura básica. Un archivo de FIO contiene las definiciones de los objetos. Cada definición de objeto especifica el tipo, valores de los atributos, y las relaciones a otros objetos para el objeto definido.

3.5.3. Lenguaje de Consulta de Objetos

Es un lenguaje declarativo para consulta y actualización de objetos. El OQL proporciona un mecanismo por el cual el usuario solicita un subconjunto de datos de la base de datos, tal solicitud es una consulta *ad hoc*.

Las características principales de OQL son:

- Descansa sobre el modelo de objetos de ODMG
- Es muy parecido a SQL92
- No es computacionalmente completo
- OQL puede ser invocado desde lenguajes de programación para los que se ha definido una liga de ODMG (language binding)
- Es declarativo y por lo tanto optimizable
- Es un superconjunto de la sintaxis de consulta *select* de SQL

- Su semántica formal puede definirse fácilmente
- Es puramente para escribir consultas
- Está diseñado para dos tipos de uso: en un intérprete de OQL y embebido en un programa.

Algunos lenguajes de consulta de objetos son:

- Object SQL
- ReLoop

3.5.4. Lenguaje de enlaze (Language Binding)

El mecanismo primario para añadir la persistencia a un lenguaje de programación es llamado lenguaje de enlace, el cual trae mecanismos de soporte para persistencia dentro del ambiente en tiempo de ejecución del lenguaje.

Es el ligado entre el Modelo de objetos de ODMG (ODL y OIF) y un lenguaje de programación, ya sea Smalltalk, C++, Java. Para este trabajo el lenguaje elegido fue Java.

Capítulo 4

El entorno de desarrollo de JAVA

En este capítulo se aborda el entorno de la plataforma JAVA. Se especifican las partes que conforman el JDK, utilizado en la construcción del sistema, así como la tecnología de `applets` y `servlets`.

Se utiliza TOMCAT como el contenedor de `servlets` y Object Store como el manejador de la base de datos orientada a objetos, ambos se integran de manera eficiente y transparente con la plataforma JAVA pero no pertenecen a SUN Microsystems. TOMCAT es parte del proyecto `jakarta` y Object Store es un producto de Object Design Inc.

4.1. La plataforma JAVA

JAVA es una tecnología de SUN Microsystems originalmente llamado *Oak* que surgió en 1991 bajo la dirección de James Gosling y Bill Joy, quienes pertenecían a una subsidiaria de SUN conocida como FirstPerson Inc. Oak nació para programar pequeños dispositivos electrodomésticos, como los asistentes digitales PAD (Personal Digital Assistants). Ninguno de estos pro-

ductos tuvo éxito comercial. Gosling y Joy se quedaron con una tecnología robusta, eficiente, orientada a objetos, independiente de la arquitectura, pero en esos momentos sin utilidad práctica.

Debido a que existen diferentes tipos de CPUs, se requería una herramienta independiente del tipo, por lo que desarrollaron un código neutro que no dependía del tipo de electrodoméstico, el cual, se ejecutaba sobre una máquina virtual denominada "Java Virtual Machine" (JVM). Es la JVM la que interpreta el código neutro a un código particular de la CPU utilizada. De ahí surgió el principal lema del lenguaje *write once, run everywhere - escríbalo una vez, ejecútelo en todos lados.* Debe mencionarse que el adjetivo *virtual* que se le da a la máquina de JAVA se debe a que se construye por software, si esta máquina es construida por hardware como en algunos productos de SUN, entonces se le nombrará simplemente máquina de JAVA.

No pasó mucho tiempo para que SUN se diera cuenta de que esas características cubrían a la perfección las necesidades de las aplicaciones en internet.

Como plataforma de desarrollo para computadoras de propósito general, JAVA se introdujo a finales de 1995. La plataforma JAVA se divide en tres tecnologías:

- Lenguaje JAVA
- Máquina de JAVA (generalmente virtual)
- Conjunto de APIs

Las características que ofrece JAVA son:

Simple

JAVA tiene la funcionalidad de proporcionar un lenguaje potente y aunque C y C++ son lenguajes más difundidos, adolecen de falta de seguridad. El lenguaje C permite operar ampliamente con punteros, y tales operaciones son muy propensas a errores ya que son totalmente controladas por el

programador, esto es eficiente con fines de velocidad pero no de seguridad; C fue originalmente un lenguaje creado para escribir sistemas operativos y no aplicaciones, sin embargo el comercio convirtió al lenguaje C muy popular para aplicaciones de propósito general. El lenguaje C++ originalmente llamado por su creador como C con clases, se considera un superconjunto de C añadiendo las tecnologías de un lenguaje orientado a objetos, pero al preservar las características de C como los punteros y agregar nuevos conceptos como referencias, lo convierte en un lenguaje híbrido que permite una gran flexibilidad, pero también una gran corrupción en el manejo de los datos en memoria. Podemos ver la implicación de mantener compatibilidades y diferentes soportes de manejo de memoria con los sistemas operativos. Por ejemplo el sistema operativo UNIX, trabaja con un solo tipo de aplicaciones y es uno de los sistemas operativos más seguros y menos propensos a errores, en cambio el sistema operativo Windows 9x, que tiene compatibilidad con aplicaciones de tipo DOS, de 16-bits y de 32-bits, lo convierte en un sistema operativo bastante inestable.

JAVA tiene gran soporte de seguridad en redes interconstruido, para que se conserve la integridad de los datos y así evitar acciones mal intencionadas, como tratar de borrar archivos o introducir virus. JAVA obliga al programador a tratar toda parte del código que pueda generar errores externos, como leer información de un floppy o de una red.

JAVA elimina de su lenguaje muchas características de otros lenguajes como C++ y añade características novedosas, como el reciclador de memoria dinámica (garbage collector). El reciclador se encarga de liberar memoria y se ejecuta en un *hilo* de baja prioridad. Cuando entra en acción el recolector de basura, permite liberar bloques de memoria grandes, lo que reduce la fragmentación de la memoria.

JAVA reduce en un 50 % de los errores más comunes de programación en lenguajes como C y C++. Entre ellos están:

- No utiliza aritmética de punteros

- Sólo existe un único modelo de referencias
- Todo código en JAVA es un objeto es decir es puro
- Tiene un soporte robusto para crear aplicaciones hiladas
- No maneja macros
- No hay necesidad de liberar memoria (`free`)

Orientado a Objetos

JAVA implementa la tecnología básica de C++ con algunas mejoras y elimina algunas para mantener el objetivo de la simplicidad del lenguaje. JAVA trabaja solo con objetos, por lo que soporta características propias del paradigma orientado a objetos: encapsulación, herencia, polimorfismo.

Las plantillas de objetos son llamadas *clases* y sus copias, *instancias*. Estas instancias necesitan ser construidas y destruidas en espacio de memoria.

Distribuido

JAVA proporciona APIs con extensas capacidades de interconexión por medio de TCP/IP. Existen librerías de rutinas para acceder e interactuar con protocolos como `http` y `ftp`. Esto permite a los programadores acceder a la información a través de una red con tanta facilidad como a los archivos locales.

Robusto

JAVA realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución. La comprobación de tipos en JAVA ayuda a detectar errores en el ciclo de desarrollo. JAVA obliga a la declaración explícita de métodos, reduciendo de esta manera las posibilidades de error. Maneja la memoria para que el programador no se preocupe de la liberación o corrupción de la memoria. Implementa los `arrays` auténticos, en vez de listas enlazadas de punteros, con comprobación de límites, para evitar

la posibilidad de sobreescribir o corromper memoria, resultado de punteros que señalan a zonas equivocadas.

Para asegurar el funcionamiento de la aplicación, realiza una verificación de los byte-codes, que son el resultado de la compilación de un programa JAVA. El código generado por el compilador es ejecutado por el intérprete JAVA (JVM); no es código máquina entendible por alguna plataforma general, pero ya ha pasado todas las fases del compilador: análisis de instrucciones, orden de operadores, y generación de la pila para la ejecución de órdenes.

JAVA proporciona:

- Comprobación de punteros
- Comprobación de límites de `arrays`
- Excepciones
- Verificación del código generado por el compilador(byte-codes)

Arquitectura neutral

Para establecer la plataforma JAVA como parte integral de una red, el compilador JAVA genera código a un archivo objeto de formato independiente de la arquitectura de la máquina en que se ejecutará (ver figura(4.37)). Cualquier máquina que tenga la JVM o sistema de ejecución (run-time) puede ejecutar el código objeto, sin importar la máquina en que ha sido generado. En la actualidad, existen sistemas run-time para Solaris 2.x, SunOs 4.1.x, Windows 95, Windows NT, Linux, Irix, Aix, Mac, Apple y posiblemente haya otros grupos de desarrollo trabajando en la migración (porting) a otras plataformas.

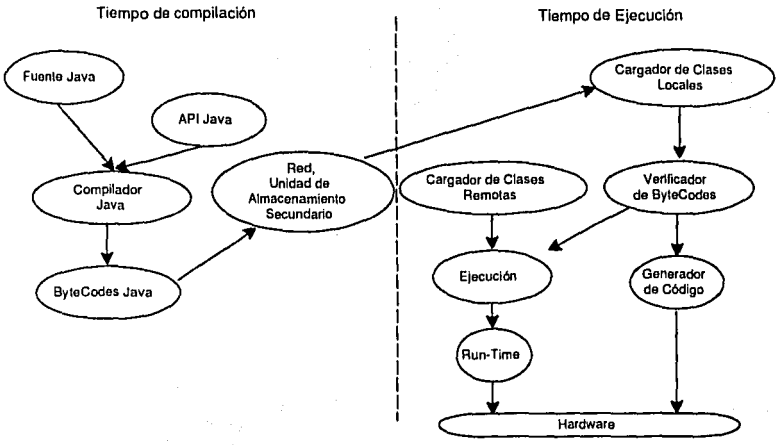


Figura 4.37. Entorno de un programa JAVA

Seguro

La plataforma JAVA, elimina características como los punteros o el *casting* implícito que proporcionan los compiladores de C y C++, con la finalidad de prevenir el acceso ilegal a la memoria. Esto se hace con el objetivo de que el lenguaje de programación pueda ser seguro, y no tenga acceso a los recursos del sistema de manera incontrolada.

El código generado por el compilador de JAVA pasa muchas pruebas (tests) antes de ejecutarse en una máquina de JAVA. El código se pasa a través de un verificador de byte-codes que comprueba el formato de los fragmentos de código para detectar fragmentos de código ilegales[4]

Si los byte-codes pasan la verificación sin generar ningún mensaje de error, entonces sabemos que:

[4]Código que falsea punteros, viola derechos de acceso sobre objetos o intenta cambiar el tipo o clase de un objeto.

- El código no produce desbordamiento de operandos en la pila.
- El tipo de los parámetros de todos los códigos de operación son conocidos y son ejecutables.
- No ha ocurrido ninguna conversión ilegal de datos, tal como convertir enteros a punteros.
- El acceso a los campos de un objeto es controlado por medio de los identificadores de acceso: `public`, `private`, `protected`.
- No hay ningún intento de violar las reglas de acceso y seguridad establecidas.

JAVA proporciona herramientas para la construcción de interfaces de usuario, a través de un sistema abstracto de ventanas, de forma que las ventanas puedan ser implementadas en entornos Unix, Pc o Mac.

Compilado

El lenguaje JAVA es *totalmente compilado*, existen compiladores como el que proporciona SUN que genera código para una máquina que interpreta y ejecuta directamente el código objeto. Compilar y enlazar (linkar) un programa, normalmente, consume más recursos que sólo compilarlo, por lo que los desarrolladores que trabajan con JAVA pasarán más tiempo desarrollando y menos esperando por tiempo de ordenador. Sin embargo, debe mencionarse que existen compiladores nativos para JAVA que generan código para una plataforma específica como lo son `Jove` para Windows o `Guava` para Linux.

Debe mencionarse que no existen lenguajes interpretados o compilados más bien existen compiladores o interpretes para un lenguaje específico. Por ejemplo, es incorrecto decir que *BASIC es un lenguaje interpretado*, ya que BASIC es solo un lenguaje; pueden existir interpretes para el lenguaje BASIC como GW-BASIC o compiladores como el QBASIC.

Multihilado

Al ser multihilado JAVA permite varias actividades simultáneas en un programa. Los hilos (a veces llamados, procesos ligeros), son básicamente pequeños procesos o piezas independientes de un gran proceso. Al estar los hilos interconstruidos en la plataforma, son más fáciles de usar y más robustos que sus homólogos en C o C++.

El lenguaje soporta la concurrencia a través de hilos. Se puede dividir una aplicación en varios flujos de control independientes, cada uno de los cuales lleva a cabo sus funciones de manera concurrente.

Dinámico

JAVA se beneficia de la tecnología orientada a objetos y de componentes, por lo que JAVA no intenta conectar todos los módulos que comprenden una aplicación hasta el tiempo de ejecución. Las librerías nuevas o actualizadas no paralizarán las aplicaciones actuales (siempre que se mantenga el API anterior).

4.2. JDK, SDK, J2SE

"Java Development Kit (JDK)", "Standard Development Kit" (SDK), y "Java 2 Standard Edition" (J2SE) son nombres para el mismo componente que incluyen: el API (Aplication Programming Interface) de JAVA, el JRE (JVM), compilador de JAVA y otras funcionalidades definidas por SUN. Ver figura (4.38).

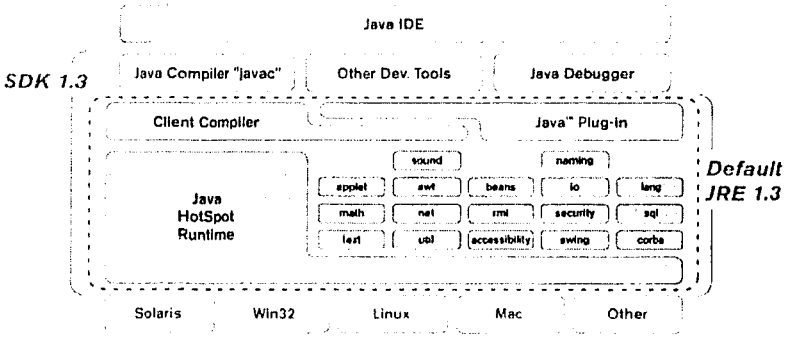


Figura 4.38. Plataforma JAVA

Ambiente de desarrollo integrado

Como se puede observar en la figura (4.38), JAVA no ofrece un ambiente de trabajo para proyectos complejos. Los ambientes de desarrollo integrado (Integrated Development Environment IDE) ofrecen un ambiente gráfico en los que se tiene acceso a mayor número de herramientas no ofrecidas en los JDK's. Algunos IDE's son:

- *Forte* de SUN
- *JBuilder* de Borland
- *Visual Cafe* de Symantec
- *Visual Age* de IBM
- *JDeveloper* de Oracle
- *Eclipse* Open-Source

El IDE que se utilizó para realizar la interfaz gráfica del sistema fue **JBuilder** versión 7 Personal, haciendo uso de los componentes `swing` para la elaboración de los `applets`.

Fundamentos del entorno JAVA

La construcción de programas JAVA, normalmente pasan por cinco fases antes de ejecutarse (ver figura 4.39). Estás son: editar, compilar, cargar, verificar y ejecutar.

La fase 1 consiste en editar un archivo que contiene el código fuente. Después el programa se almacena en un dispositivo de almacenamiento, como un disco. Los nombres de los archivos que contiene código en JAVA deben tener la extensión `.java`.

En la fase 2, el programador emite el comando `javac` para compilar el código fuente. El compilador de JAVA traduce el código fuente (de alto nivel) a códigos de bytes (de bajo nivel), que es el código que entiende la JVM.

Si el programa compila correctamente, se producirá un archivo con extensión `.class`. Este archivo contiene los códigos de bytes que serán interpretados durante la fase de ejecución.

La fase 3 se llama carga. Antes de que un programa pueda ejecutarse, es necesario colocarlo en memoria. Esto se realiza mediante un cargador de clases que toma el archivo `.class` que contiene los códigos de bytes y lo transfiere a la memoria. El archivo `.class` puede cargarse de un disco del sistema propio o a través de una red.

El comando, `java Ejemplo` invoca la JVM para el programa `Ejemplo` lo que implica que el cargador de clases lea la información empleada en el programa `Ejemplo`. El cargador de clases también se ejecuta cuando un `applet` de JAVA se ejecuta dentro de un navegador WEB, como Navigator de Netscape, Internet Explorer de Microsoft o HotJava de SUN.

En la fase 4, antes de que la JVM pueda ejecutar los códigos de bytes, éstos pasan por un proceso de verificación. Esto asegura que los códigos de bytes son válidos y que no violan las restricciones de seguridad. La JVM debe hacer cumplir reglas de seguridad ya que los programas que llegan por medio de la red pueden causar daños a los archivos y el sistema del usuario.

En la fase 5, La JVM utiliza el conjunto de recursos de la plataforma donde se hospeda, para poder ejecutar el código de bytes o código objeto.

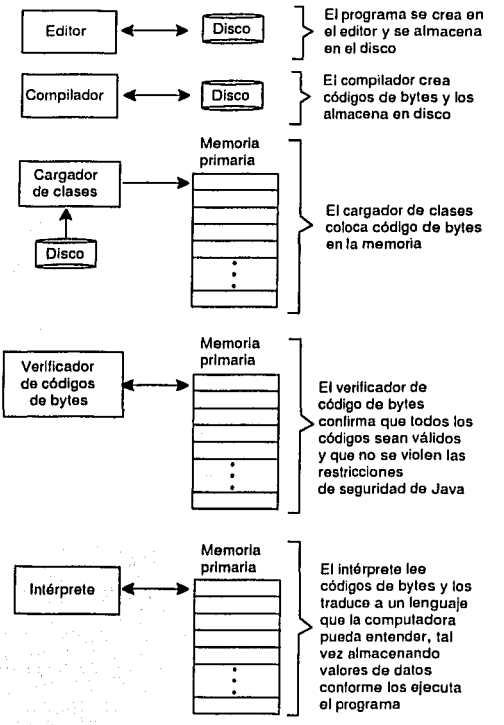


Figura 4.39. Compilación de un programa JAVA

La Java Virtual Machine (JVM). Como se dijo con anterioridad, la existencia de diferentes procesadores en la industria de la computación, llevó a los ingenieros de SUN a construir una plataforma que no dependiera del tipo de procesador utilizado. Por ello se desarrolló un código neutro que se ejecutara sobre la JVM. La JVM se encarga de ejecutar el código neutro convirtiéndolo a código particular de la CPU utilizada. De esta manera, se

evita tener que realizar un programa diferente para cada CPU o plataforma (Hardware + SO). Ya que en una gran variedad de plataformas o sistemas operativos (Windows, Linux, etc) existe la JVM, garantiza que toda aplicación escrita en JAVA puro (sin llamadas a métodos nativos) logre ser ejecutada en dichas plataformas. Este componente de JAVA es el ingrediente principal del logro *Write Once, Run Everywhere.*

Debe mencionarse que para acelerar la ejecución de los bytes-code sobre la JVM existen compiladores en tiempo de ejecución que analizan el código y lo optimizan antes de ejecutarlo en la plataforma nativa. También existen compiladores nativos que generan código para una plataforma específica, sin embargo son excesivamente caros y no implementan en su totalidad todos los APIs de SUN.

4.3. Servlets

Un `servlet` es un componente de software, escrito en JAVA, que dinámicamente atiende peticiones por medio de una red de computadoras. Los `servlets` son módulos que extienden la funcionalidad de servidores orientados a petición-respuesta, como por ejemplo un servidor `http`.

Por ejemplo, en la figura (4.40) un `servlet` podría ser el responsable de tomar los datos de un formulario de entrada de pedidos en HTML y aplicarle la lógica de negocios utilizada para actualizar la base de datos de pedidos de una compañía.

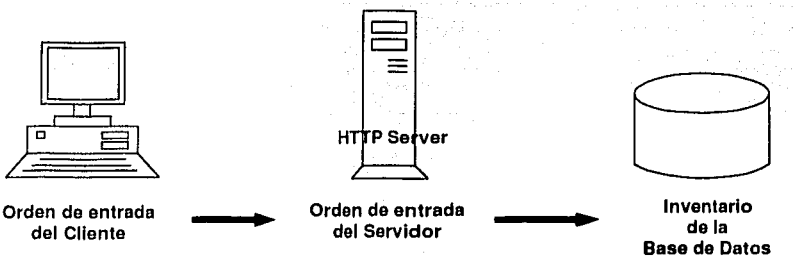


Figura 4.40. Uso de Servlets

Características de los servlets

1. Específicamente los `servlets` son clases JAVA que tienen una interfaz específica y pueden ser invocadas desde un servidor. La funcionalidad que provee el `servlet` no esta restringida a servidores WEB.

2. El `servlet` API es una especificación desarrollada por SUN que define una jerarquía de clases para crear y ejecutar `servlets`.

3. Los `servlets` no tienen interfaz gráfica de usuario.

4. Son independientes de la plataforma en donde se ejecutan. A pesar de estar escritos en JAVA, el servidor puede estar escrito en cualquier lenguaje de programación.

5. Se pueden comunicar distintos `servlets` entre sí .

6. Son muy seguros (hacen uso del Security Manager de JAVA). Por lo que no pueden realizar conexiones directas a través de sockets de red.

7. Son eficientes, dado que múltiples llamadas sobre el mismo servlet no origina la ejecución de distintos procesos, sino que se crean hilos de ejecución una vez que ya existe el proceso del servlet ejecutándose en la máquina.

Entre las ventajas de los servlets destacamos las siguientes:

- *Capacidad de correr en un mismo proceso.* Los `servlets` son capaces de correr en el mismo espacio de direcciones de un proceso del servidor a diferencia de los métodos de programación del lado del servidor como la interfaz CGI, los cuales requieren que los programas corran como procesos disjuntos cuando el cliente solicita el servicio.
- *Compilación.* A diferencia de los lenguajes basados en `scripts` (interpretados por el navegador), los servlets se compilan dentro de códigos de bytes de JAVA. Los servlets en JAVA pueden ejecutarse más rápido que los lenguajes comunes basados en scripts.

Uso de los servlets

- Los `servlets` extienden la funcionalidad de un servidor de archivos como lo es un servidor `http`, por medio de una Interfáz Común de Comunicaciones de red (CGI - Common Getaway Interfaz), los `servlets` permiten la interacción entre el servidor y el cliente de manera dinámica.
- Construyen y regresan dinámicamente un archivo HTML basado en la naturaleza de la solicitud del cliente o archivos binarios.
- Procesan entradas del usuario capturadas por una forma HTML y regresa una respuesta.
- Facilitan la comunicación entre grupos de personas para la publicación de información enviada por muchos clientes.
- Proporcionan autentificación del usuario y otros mecanismos de seguridad.
- Interactúan con los recursos del servidor como pueden ser una base de datos, aplicaciones nativas y archivos de red, para regresar información conveniente al usuario.

- Permiten al servidor comunicarse con un cliente `applet` por medio de un protocolo personalizado.

Ciclo de vida de los servlets.

Cada servlet tiene el mismo ciclo de vida y es el siguiente:

- Un servidor carga e inicializa el `servlet`.
- El `servlet` maneja una o más peticiones del cliente.
- El servidor elimina el `servlet` cuando no es requerido. (Algunos servidores sólo cumplen este paso cuando se desconectan).

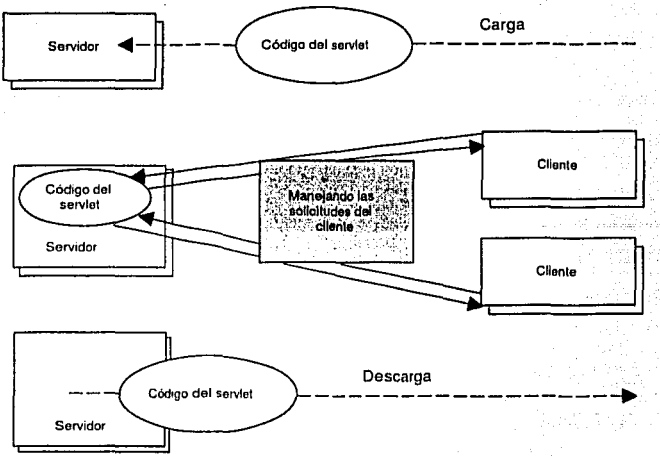


Figura 4.41. Ciclo de vida del Servlet

4.4. Applets

Un `applet` es una pequeña aplicación cliente escrita en JAVA, que se transporta por la red, y se instala automáticamente como parte de un documento WEB.

El `applet` está almacenado en el servidor y se transmite al usuario por medio de la red. La ventaja de los applets con respecto a otros medios audiovisuales es que el usuario puede interactuar con un `applet`.

Un `applet` se ejecuta completamente del lado del cliente. Si es necesario, el `applet` también se puede comunicar con el servidor.

Para que un navegador de WEB pueda ejecutar un `applet` es necesario que posea una JVM. Actualmente Netscape, Internet Explorer y Hotjava cumplen con este requisito.

Uso de un applet

Los `applets` son capaces de realizar operaciones muy complejas a pesar de su reducido tamaño de código. Los `applets` se sirven de los recursos del navegador donde se ejecutan, para realizar las tareas de presentación gráfica o de comunicaciones.

Algunas de las tareas más significativas de un `applet` son:

- Presentar animaciones gráficas en la ventana del navegador.
- Reproducir música y sonido.
- Establecer comunicaciones con el servidor de donde procede el `applet`. En particular, existen funciones para descargar archivos de imágenes y de sonido, etc.
- Crear una interfaz gráfica con los elementos usuales de los entornos de ventanas, como menús desplegables, barras de desplazamiento, áreas de texto, etc.

- Pedir datos al usuario para su proceso. En general, la interacción con el usuario es total, pudiendo gestionarse eventos como pulsaciones de teclas, movimientos del ratón, etc.

Ciclo de vida de un applet

Cuando un `applet` se carga en el `appletviewer`[5], inicia su ciclo de vida de la siguiente manera:

- Se crea una instancia de la clase que controla el `applet`. Es decir, lo primero que lleva a cabo es cargar la referencia que se indica en el tag de HTML `CODE`. Esta es una operación interna del navegador.

- El `applet` se inicializa. El primer método del `applet` que invoca el navegador es el método `init()`, que se encarga de realizar todas las labores de inicialización necesarias para que el `applet` pueda iniciar su ejecución. El método `init()` sólo se invoca en una ocasión, y es justo después de que el navegador a terminado de cargar el `applet`.

- El `applet` comienza a ejecutarse. El navegador invoca el método `start()`, para indicar al `applet` que ya puede comenzar el proceso de ejecución. El navegador invocará `start()` cada vez que desea arrancar el `applet` de nuevo. Por lo tanto, se incluirá aquí código que sea necesario repetir cada vez que se desee volver a arrancar el `applet`. La utilización más común de `start()` es lanzar uno o más hilos de control, de modo que la ejecución del `applet` transcurra en paralelo con la del navegador y la del resto de las aplicaciones del sistema. Una vez que `start()` se ha invocado, este puede invocar opcionalmente a los métodos: `paint()`, `update()` y `repaint()`.

- La detención del `applet` se lleva a cabo por el método `stop()` y ocurre cuando el lector deja la página que contiene el applet que actualmente

[5]el appletviewer es una herramienta proporcionada por SUN para ejecutar applets sin todo el entorno que implica un navegador.

está ejecutándose. Por defecto, cuando el lector de la página la abandona, el applet continúa ejecutándose. Derogando el método `stop`, se puede suspender la ejecución de un applet y restaurarla cuando ésta sea visitada de nuevo.

- El método `destroy()` es llamado cuando el navegador no tiene activo el `applet` y necesita liberar recursos o cuando se cierra el navegador.

La figura (4.42) muestra el ciclo de vida de un `applet`

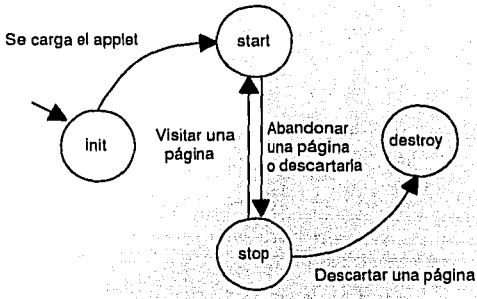


Figura 4.42. Ciclo de vida del Applet

4.5. Swing

JFC (Java Fundation Classes) comprende un grupo de **características** para construir interfaces gráficas de usuario (GUI), estas son (ver figura (4.43)):

- `swing`
- Soporte de aspecto y comportamiento (pluggable look and feel)
- API de accesibilidad
- JAVA 2D API (sólo JDK1.2)

- Soporte de *arrastrar y poner* (drag and drop sólo JDK1.2)
- AWT (Abstract Windows Toolkit)
- Internacionalización

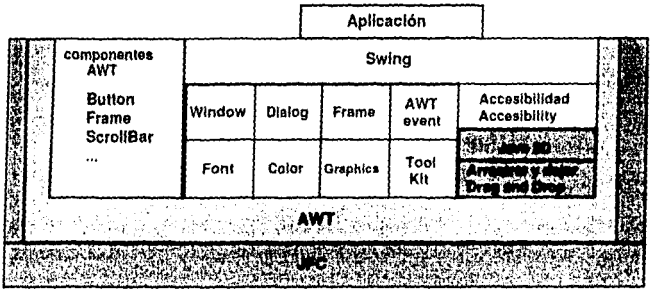


Figura 4.43. JFC para interfaces gráficas

Los paquetes gráficos que contienen las JFC son `swing` y AWT. El paquete que se utilizó para la GUI del sistema fue `swing`.

`Swing` apareció en la versión 1.2 de JAVA. Está conformado por un conjunto de componentes de interfaces de usuario que funcionan en una gran variedad de plataformas JAVA (algunas versiones de JAVA como la Micro Edition no soporta `swing`.

Una diferencia importante entre los componentes `swing` *y AWT*, radica en que los componentes `swing` son ligeros, esto quiere decir que la JVM pide lo mínimo necesario a la plataforma huésped para construirlos, esto es soportado gracias a la arquitectura *modelo-vista-controlador* (MVC) que permite diseñar los componentes en gran medida en la JVM. Por su parte AWT es un componente pesado, ya que está asociado con los componentes gráficos nativos de la plataforma huésped, es decir cuando la JVM requiere de un componente AWT, ésta se lo pide como tal a la plataforma que lo soporta.

Otras ventajas de swing con respecto a AWT

- Amplia variedad de componentes.
- Los *botones* y las *etiquetas* pueden mostrar imágenes además de texto.
- Aspecto modificable (look and feel) que permite personalizar la estética de la interfaz de usuario, utilizando varias vistas que son:
  - Metal. Java Metal look and feel
  - CDE/Motif. Solaris look and feel
  - Windows. Windows look and feel
- Arquitectura Modelo-Vista-Controlador
- Se pueden crear distintos tipos de bordes complejos alrededor de los componentes

El patrón de diseño Modelo-Vista-Controlador (MVC)

Muchos de los componentes **swing** están basados en un patrón de diseño llamado MVC. La arquitectura MVC separa los datos de la aplicación (modelo), de la representación gráfica de los componentes (vista) y la lógica de procesamiento de entrada (controlador). El concepto de este patrón de diseño se basa en tres elementos:

- *Modelo.* Almacena el estado interno en un conjunto de clases, es decir contiene los datos de la aplicación.
- *Vista.* Genera una representación de los datos almacenados en el modelo, es decir muestra la información del modelo.
- *Controlador.* Implementa la lógica para procesar las entradas del usuario, es decir cambia la información del modelo (Delegado).

La figura (4.44) muestra la relación entre los componentes MVC.

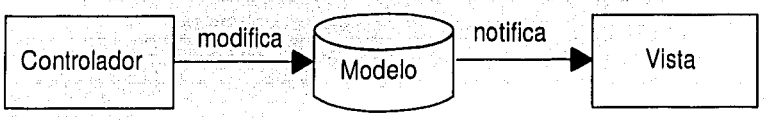


Figura 4.44. Arquitectura MVC

La figura (4.45) muestra un ejemplo de la arquitectura Modelo-Vista-Controlador

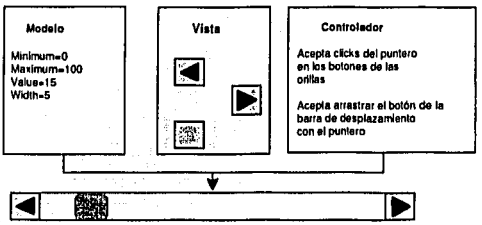
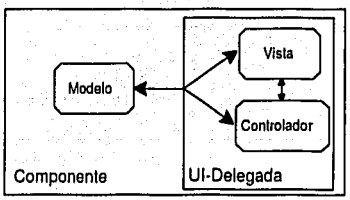


Figura 4.45. Ejemplo MVC

Los componentes swing de JAVA implementan una variación de MVC que combina la vista y el controlador dentro de un objeto, llamado *delegado*. Este objeto proporciona la representación gráfica del modelo y una interfaz para modificar el modelo (ver figura (4.46)).



UI Delegada = Interfaz de usuario Delegada

Figura 4.46. Arquitectura Delegación-Modelo

4.6. CGI

CGI significa *Common Gateway Interface*- en otras palabras, es un método estándar de comunicación entre diferentes procesos de tipo `gateway` (compuerta).

CGI es un conjunto de reglas que definen cómo se realiza la comunicación que emplea un servidor WEB (la computadora que manda una página WEB) para enviar información útil en ambos sentidos, entre el navegador y su propio programa de cómputo. Este mecanismo permite la programación de páginas interactivas.

En el sentido más estricto del término, no es un lenguaje o protocolo, es sólo un conjunto de variables y convenciones nombradas para pasar información en ambos sentidos entre el servidor y el cliente.

Cabe enfatizar que la programación de las secuencias CGI pueden realizarse utilizando diversos lenguajes, como: C, Visual Basic, AppletScript, Perl, etc.

4.7. ObjectStore para JAVA

El SMBDOO utilizado para implementar el presente sistema es ObjectStore PSE Pro for JAVA Release 3.0.

Un SMBDOO se caracteriza por tener un modelo de datos lógico orientado a objetos y por usar un lenguaje de programación orientado a objetos como su principal interfaz. ObjectStore soporta C++ y JAVA.

ObjectStore en su soporte para JAVA incluye tres productos:

- ObjectStore PSE (Personal Storage Edition)
- ObjectStore PSE Pro
- JAVA interface to ObjectStore Development Client

PSE y PSE Pro proporcionan un API que permite:

- Iniciar y finalizar sesiones
- Crear, abrir, cerrar y destruir la base de datos
- Iniciar la operación de `commit`, y abortar transacciones de acceso de datos en la base de datos.
- Leer y escribir las raíces de la base de datos, las cuales proporcionan el inicio de los puntos para navegar a los objetos persistentes.
- Almacenar, recuperar y actualizar los objetos en la base de datos.

PSE Pro es un superconjunto de PSE, provee todas las características de PSE e incluye otras, como son:

- Soporta cientos y miles de objetos y cientos de megas en bytes en una base de datos.
- Proporciona un mejor soporte de la concurrencia.
- Permite múltiples sesiones.
- Reúne la basura de la base de datos.
- Soporta consultas e índices.
- Incluye una recuperación completa de la base de datos de fallas del sistema.
- Incluye utilidades para desplegar información a cerca de los objetos y checar las referencias en la base de datos.

Términos de ObjectStore PSE Pro

Sesión (session). PSE Pro usa la clase abstracta **COM.odi.Session** para representar las sesiones.

La aplicación debe crear una sesión antes de que ésta pueda usar el API de PSE Pro. Después que una sesión es creada, ésta es una sesión activa. Una sesión permanece activa hasta que la aplicación o PSE Pro la finaliza. Una vez que la sesión terminó, no podrá ser utilizada nuevamente. Sin embargo, se puede crear una nueva sesión.

Una sesión crea un contexto en el cual se puede crear una transacción, un acceso a la base de datos y manipular objetos persistentes. Una sesión consta de un conjunto de objetos persistentes y un conjunto de objetos del API de PSE Pro, tales como, una transacción (transaction), base de datos (database).

En una instancia de la JVM:

a) PSE Pro permite múltiples sesiones al mismo tiempo.

b) PSE permite una sesión a la vez.

En cada instancia de la JVM se puede crear una sesión al mismo tiempo. Adicionalmente PSE Pro permite que múltiples sesiones en cada instancia de la JVM se puedan crear.

Capacidad de persistencia (Persistence-capable). El término *capacidad de persistencia* se refiere a la capacidad de un objeto de ser almacenado en la base de datos. Si se puede almacenar un objeto en la base de datos, se dice que es *capaz de ser persistente*. En otras palabras, si se pueden almacenar las instancias de una clase en la base de datos, ésta es *capaz de ser persistente* y las instancias también.

La definición de una clase con *capacidad de persistencia* incluye anotaciones específicas por PSE Pro. Después de compilar una clase, se ejecuta el

`postprocesador`[6] sobre la clase compilada para añadir código a bajo nivel (archivo `.class`) que permite a las clases tener *capacidad de persistencia*. Se puede explícitamente definir una clase como persistente dentro del código fuente o ejecutar el postprocesador para modificar el código objeto de la clase ya compilada. Si se opta por utilizar el postprocesador las clases no heredan la *capacidad de persistencia* por lo que se debe de ejecutar el postprocesador en todo el código objeto de la jerarquía de clases relacionadas. Sin embargo, definir la persistencia en el código fuente de la clase puede conducir a errores en la programación y a complicar el entendimiento del código al insertar fragmentos que no son propios de la aplicación.

Es importante mencionar que no todas las clases o no todos los atributos de una clase pueden ser definidos como persistentes, clases como las abstractas no pueden ser marcadas como persistentes, los métodos o atributos de clase (estáticos) tampoco pueden ser persistentes.

Objeto Persistente. Un objeto persistente es la representación de un objeto que está almacenado en la base de datos.

Después de que una aplicación recupera un objeto de la base de datos, la aplicación trabaja con el objeto persistente en el ambiente JAVA. Un objeto persistente pasa por los siguientes estados:

a) *Sombra* (Hollow). Un objeto persistente sombra tiene la misma estructura que el objeto en la base de datos que éste representa. Un objeto sombra es creado con fines de optimización, con el objetivo de aminorar la carga en memoria, es decir es solo una representación ligera del objeto que se encuentra en la base de datos, por lo que no tiene los valores de los atributos.

[6]Utilidad de PSE Pro para facilitar la creación de clases persistentes.

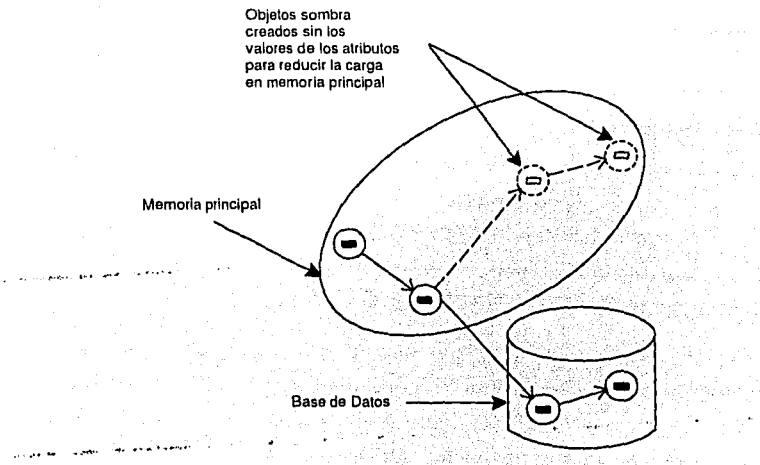


Figura 4.47. Objetos Sombra

b) *Activo* (Active). Un objeto persistente activo es una copia exacta del objeto que éste representa en la base de datos. El contenido de un objeto activo esta disponible para ser leído y/o modificado por la aplicación. Si un objeto activo es modificado por la aplicación, éste ya no es idéntico al objeto en la base de datos y se dice que su estado es *dirty* (sucio). Si un objeto activo conserva su estado es decir no es modificado se dice que su estado es *clean* (limpio). Debe mencionarse que el paso de un objeto del estado *Hollow* al estado *Active* es transparente al programador.

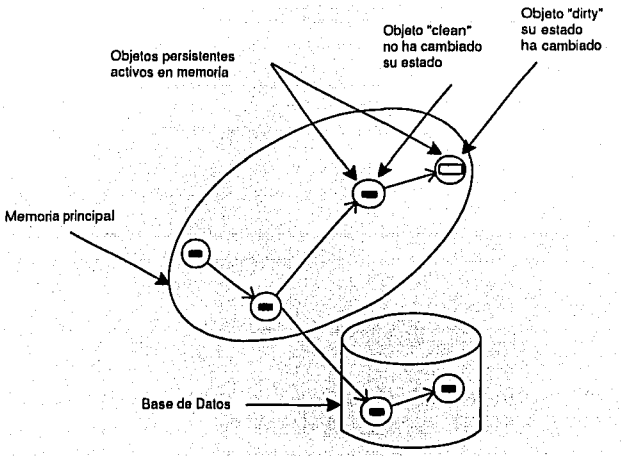


Figura 4.48. Objetos Activos

c) *Duro* (Stale). Un objeto persistente *stale* no puede ser usado ya que no se encuentra en memoria principal, si se requiere ser modificado o utilizado es necesario volver a indicar al sistema que lo pase a objeto sombra-activo. Un objeto llega a ser stale cuando una aplicación llama:

- Transaction.commit(). Las modificaciones a los objetos dentro de la transacción son almacenadas.
- Transaction.abort(). Las modificaciones a los objetos son ignoradas y se restaura su estado original antes de comenzar la transacción.

Si una aplicación trata de leer o actualizar un objeto *stale*, PSE Pro lanza la excepción `ObjectException`. Una aplicación no puede ejecutar ningún método de instancia en un objeto stale.

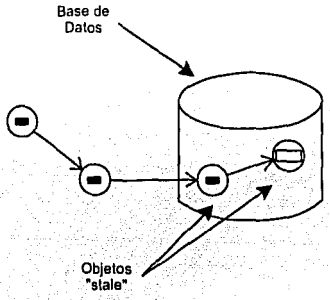


Figura 4.49. Objetos Stale

Persistencia notificada (Persistence-aware). Si los objetos de una clase pueden operar sobre atributos de objetos persistentes, pero estos no son `persistence-capable` entonces se definen como `persistence-aware`.

Objeto transitorio (Transient Object). Un objeto transitorio es un objeto que no se almacena en la base de datos.

Persistencia transitiva (Transitive persistence). Cuando una aplicación realiza una transacción, PSE Pro almacena en la base de datos cualquier objeto transitorio que pueda ser alcanzado transitivamente por algún objeto persistente. Los objetos transitivos que son referenciados por objetos persistentes se convierten en persistentes por transitividad.

Raíces de la Base de Datos. Una raíz de la base de datos proporciona la manera de asociar un *nombre* con un objeto en la base de datos. Las aplicaciones usan las raíces de las bases de datos para localizar uno o más objetos persistentes, para mejorar las consultas o navegar a otros objetos persistentes.

Beneficios de usar PSE Pro

PSE Pro proporciona un API completo para almacenar y compartir objetos JAVA entre los usuarios, servidores y programas. Además de que permite definir clases capaces de ser persistentes (persistence-capable).

También permite leer y modificar rápidamente partes de los datos persistentes. No es necesario leer en todos los objetos persistentes cuando sólo se quiere buscar un subconjunto.

Cuando se accesa a datos persistentes dentro de una transacción, PSE Pro asegura que los resultados no son comprometidos por otros usuarios que comparten los datos. Si algo va mal, o si se determina que no se quieren guardar los cambios, se pude abortar la transacción. En este caso PSE Pro restaura la base de datos a su estado antes del comienzo de la transacción.

PSE Pro contra la Serialización

El tiempo de vida de los objetos persistentes excede el tiempo de vida de la aplicación que lo crea. El manejo de objetos persistentes es un mecanismo para almacenar el estado de los objetos en un lugar no volátil y cuando la aplicación termine, los objetos continúen existiendo. JAVA en su API provee la serialización de objetos *como una forma simple de persistencia de los objetos*. Se puede usar la serialización para almacenar copias de objetos en un archivo o transportar copias de objetos a una aplicación a otra por medio de la red.

Sin embargo, no es óptimo elegir la serialización para aplicaciones que:

- Manejan de diez a cientos de megabytes de objetos persistentes.
- Actualizar objetos frecuentemente.
- Asegurar que los cambios son salvados fiablemente en almacenamiento persistente.

La serialización no proporciona confianza para almacenar objetos, si el sistema o la aplicación dejan de funcionar cuando los objetos se están escribiendo en disco. Para la protección en contra de fallas de la aplicación o el

sistema y asegurar que los objetos persistentes no son destruidos, se debe copiar el archivo persistente antes de salvar cada cambio.

PSE Pro es un SMBDOO escrito completamente en JAVA. PSE Pro tiene un pequeño paquete de 300k comparado con muchos DBMS's, por lo que es fácil ser empaquetado con pequeñas aplicaciones o `applets`. Las tres formas en que PSE Pro supera la serialización son:

- Mejora el rendimiento de números grandes de objetos
- Maneja objetos fiables
- Permite consultas

PSE Pro permite el acceso tanto a un pequeño grupo como a un objeto simple al mismo tiempo, y permite a múltipes aplicaciones leer de la base de datos al mismo tiempo.

Transacciones y recuperación. Una diferencia primaria entre la serialización y PSE Pro es evidente en el área de transacciones y recuperación. Con la serialización, los almacenamientos persistentes no son automáticamente recuperables. En consecuencia, en el evento de una falla de la aplicación o sistema, un archivo sólo puede ser recuperado al comienzo de sesión de la aplicación y sólo si una copia del archivo es hecha antes de que la aplicación comience. En contraste, PSE Pro puede recuperar una falla en la aplicación o caída del sistema, si la falla impide algunos de los cambios en la aplicación sean salvados en el disco, PSE Pro asegura que ninguno de los cambios de la transacción repercutirán en la base de datos. Cuando se reinicia la aplicación, la base de datos es consistente con la manera en que ésta fué almacenada antes de la transacción. PSE Pro soporta transacciones atómicas, es decir todos los cambios son salvados o no son salvados.

Fácil de usar. PSE Pro proporciona una interfaz que facilita el almacenar y recuperar objetos JAVA. Permite definir clases JAVA persistence-capable, así también como clases JAVA persistence-aware.

Habilidad de leer y actualizar un objeto. Mientras que la serialización lee y escribe grafos completos de objetos, PSE Pro proporciona explícito control de granularidad fina sobre el acceso a los objetos para cargarlos. Esto implica que se pueden leer y escribir objetos simples de la base de datos. PSE Pro automáticamente carga los objetos relacionados cuando el código de la aplicación se refiere a ellos.

Concurrencia. PSE Pro soporta acceso de programas simples a la base de datos, esto significa que la base de datos puede ser actualizada por al menos una aplicación al mismo tiempo. Múltiples aplicaciones pueden leer la misma base de datos simultáneamente, pero solo una aplicación puede escribir al mismo tiempo en la base de datos.

Soporte para el Modelo de Objetos

ObjectStore soporta todos los conceptos fundamentales del modelo de objetos:

- Clases
- Atributos
- Métodos
- Asociaciones
- Herencia

Clases Colección. Una librería de clases es un componente importante de cualquier ambiente de desarrollo orientado a objetos y consta de código presumiblemente eficiente y reusable.

Una colección es un objeto que agrupa y puede manejar otros objetos, que pueden ser: objetos que representan colecciones menores y objetos primitivos. PSE Pro soporta colecciones heterogéneas (tipos de objetos diferentes) y colecciones homogéneas (tipos de objetos iguales). PSE Pro incluye

un protocolo para iterar sobre los elementos de una colección. También soporta colecciones persistentes y temporales, las colecciones persistentes solo pueden contener elementos persistentes.

PSE Pro soporta las siguientes clases colección:

- `set`. Es una colección desordenada de objetos que no acepta duplicados. Permite insertar y eliminar objetos e iterar en orden arbitrario sobre ellos.
- `bag`. Es una colección desordenada de objetos que permite duplicados. Permite insertar objetos, eliminarlos e iterar para accesar a cada uno de ellos en orden arbitrario.
- `list`. Es una colección desordenada de objetos que permite duplicados.
- `array`. Es una colección ordenada e indexada de objetos que permite duplicados. Cuando se crea un array sus elementos se inicializan en `null`. Las operaciones básicas de lectura y escritura son hechas en una localización particular del arreglo.
- `dictionary`. Es una colección desordenada de objetos que permite duplicados. El diccionario asocia una clave por cada elemento.

4.8. Tomcat

Tomcat es un contenedor de `servlets` con un entorno JAVA Server Page(JSP), un contenedor de `servlets` es un `shell` de ejecución que maneja e invoca `servlets` por cuenta del usuario. Los contenedores de servlets se dividen en:

1. *Contenedores de Servlets Stand-alone(Independientes).* Estos son una parte integral del servidor WEB. Es el caso de un servidor WEB basado en JAVA, por ejemplo, el contenedor de servlets es parte de JAVAWeb-Server (actualmente sustituido por iPlanet). Sin embargo, la mayoría de los servidores, no están basados en JAVA como Tomcat el cual corre nativamente en la plataforma para mejorar el rendimiento.

2. *Contenedores de servlets dentro del proceso.* Es una combinación de un `plug-in` para el servidor WEB y una implementación del contenedor JAVA. El `plug-in` del servidor WEB abre una Máquina Virtual Java (JVM) dentro del espacio de direcciones del servidor WEB y permite que el contenedor JAVA se ejecute en él. Si una petición requiere ejecutar un `servlet`, el `plug-in` toma el control sobre la petición y lo pasa al contenedor JAVA usando JNI.

3. *Contenedores de Servlets fuera del proceso.* El contenedor `servlet` es una combinación de un plugin para el servidor WEB y una implementación de contenedor JAVA que se ejecuta en una JVM fuera del servidor WEB. El plugin del servidor WEB y el contenedor JAVA se comunican usando algún mecanismo sobre TCP/IP. Si una petición requiere ejecutar un `servlet`, el `plug-in` toma el control sobre la petición y lo pasa al contenedor JAVA (usando IPCs, v.gr. tuberías). El tiempo de respuesta en este tipo de contenedores no es tan bueno como el anterior, pero se obtiene mejor rendimiento en otras características como: escalabilidad, estabilidad, etc.

4.9. Documentos PDF

El formato de documentos portables (PDF) de Acrobat es un estándar para la distribución de documentos electrónicos de alta calidad que preserva las fuentes, el formato, los gráficos y el color de los documentos. Son independientes de la plataforma por lo que pueden ser visualizados en cualquier computadora, son compactos y pueden ser compartidos, navegados e impresos exactamente como se pretende.

Cuando se generan reportes en aplicaciones distribuidas basadas en internet que utilizan un navegador como su principal cliente, existe el inconveniente de que los reportes generados en HTML su impresión y visualización dependen totalmente del navegador, por lo que es posible que diferentes clientes generen distintas impresiones del mismo reporte, éste es un inconveniente si se pretende crear documentos de calidad y formales.

Para generar los reportes del sistema se usó el iTextPDF de Bruno Lowagie. Este API permite crear documentos PDF dinámicamente desde cualquier aplicación JAVA.

Los principales problemas que se presentan al utilizar documentos portables en HTML u otros dependientes de la plataforma para fines de publicación e impresión de alta calidad se presentan en la siguiente tabla:

Problemas comunes	Solución con PDF
Los usuarios no pueden abrir los archivos porque no tienen la aplicación que los creo.	Cualquier usuario en cualquier lugar puede abrir un archivo PDF utilizando software libre como Acrobat Reader.
El formato, las fuentes, y los gráficos se pierden debido a la incompatibilidad del software.	Los documentos PDF siempre se muestran de igual manera como fueron creados.
Los documentos no se imprimen correctamente por las limitaciones de impresión del software.	Los documentos PDF siempre se imprimen correctamente en cualquier dispositivo de impresión.
Los documentos no pueden ser creados para visualizarlos a través de distintos dispositivos, por ejemplo, computadoras de bolsillo, sitios WEB, máquinas imprentas, etc.	Los documentos PDF preservan la integridad visual de un documento hasta en máquinas que ejecutan Palm OS®.
El contenido de los documentos no puede ser usado para otros propósitos como extraer texto o imágenes debido a los problemas de formato.	Los documentos en PDF pueden ser salvados en formato RTF y reusados en otras aplicaciones.

Capítulo 5

Implementación

En este capítulo se describe la arquitectura de tres capas, utilizada para la implementación del sistema, mostrada por los diagramas de implementación que soporta UML.

Los diagramas de implementación se dividen en el diagrama de componentes que muestra la estructura del código y el diagrama de despliegue (también llamados diagramas de emplazamiento) que muestran la estructura del sistema en tiempo de ejecución.

5.1. Arquitectura de tres capas (three-tier)

La arquitectura de tres capas a diferencia de la arquitectura tradicional cliente-servidor, tiene una capa intermedia que juega un papel vital en aplicaciones de tres capas. Ésta maneja las solicitudes de los clientes, los protege de la complejidad envuelta con servidores y bases de datos. El servidor de la capa intermedia puede soportar una variedad de clientes, tales como navegadores WEB, aplicaciones JAVA y dispositivos portátiles. Los clientes manejan la interfaz de usuario, no consultan directamente la base de datos, no ejecutan reglas del negocio complejas, no se conectan a aplicaciones legadas, es

decir, los clientes se alejan de la complejidad de la lógica del negocio para que el servidor de la capa intermedia haga este trabajo transparente para ellos. La figura (5.50) ilustra la arquitectura de tres capas. La capa 1 esta compuesta de múltiples clientes, los cuales solicitan servicios al servidor de la capa intermedia, la capa 2. El servidor de la capa intermedia accesa a los datos desde los sistemas existentes en la capa 3, aplica las reglas del negocio a los datos y regresa los resultados a los clientes en la capa 1.

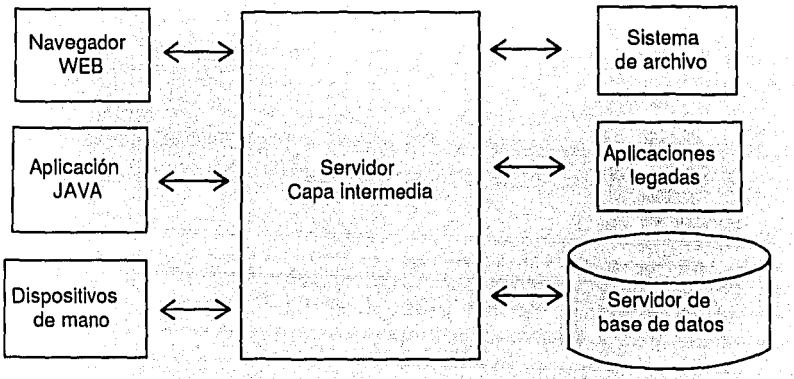


Figura 5.50. Arquitectura de 3 capas

Los servidores de la capa intermedia proporcionan los servicios del sistema al cliente. Por ejemplo, un servidor de la capa intermedia en una aplicación de compra en línea puede proporcionar una variedad de servicios como: buscar en el catálogo, entrada de orden y verificación de crédito.

Los servidores de la capa intermedia también proporcionan servicios a nivel de sistema, como son:

- Acceso remoto a los clientes y servidores
- Manejo de sesiones y transacciones

- Aplicación de la seguridad
- Contenedores de recursos
- Ya que la capa intermedia proporciona estos servicios, los clientes pueden ser ligeros, sencillos y desarrollados rápidamente.

Los `Applets` y `Servlets` de la tecnología `JAVA` pueden usarse en conjunto para el diseño de aplicaciones `WEB` multihiladas. Los `Applets` poseen un mecanismo para construir aplicaciones con interfaces dinámicas, mientras que los `Servlets` proporcionan un manejo eficiente de solicitudes en un servidor `WEB` o servidor de aplicaciones.

En la arquitectura de tres capas para el diseño del sistema, se encapsula la comunicación, con los recursos que dan soporte al sistema (*back-end*) dentro de los `Servlets`, dejando a los `Applets` el manejo de la interfaz de usuario (*front-end*).

Los `Servlets` ayudan a superar las restricciones de seguridad inherentes en los `Applets` y el control de acceso de los `Applets` a los sistemas de información y lógica del negocio. Cuando una solicitud es atendida por un `Servlet`, éste puede buscar información en los recursos (*back-end*) de la base de datos, mejorar los cálculos o hacer lo que sea necesario para obtener información en nombre del `Applet`. Una gran ventaja es que los pares `Applets/Servlets` pueden ser desplegados a través de una gran variedad de servidores `WEB`.

Diseñando a través de `Servlets` ayuda a modularizar el diseño, abstraer la lógica del negocio detrás de la aplicación, y proporcionar escalabilidad. En la figura (5.51) se muestra este tipo de arquitectura.

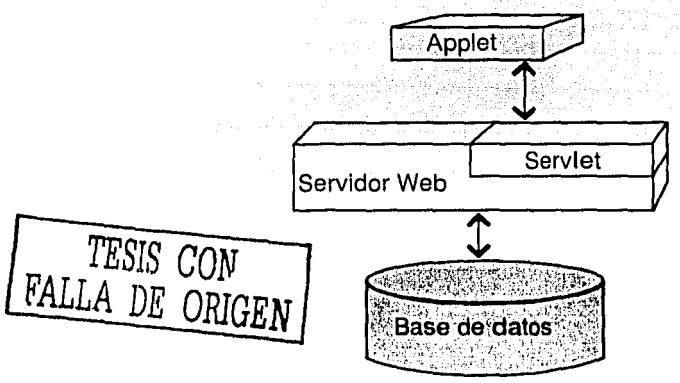


Figura 5.51. Arquitectura de 3 capas utilizando Servlets

5.2. Diagrama de componentes

Un diagrama de componentes muestra la dependencia entre los componentes de software, incluyendo el código fuente y ejecutable de éstos. Para un negocio, los componentes de software son tomados del sentido común que inducen los procedimientos y documentos del negocio. Algunos componentes existen en tiempo de compilación, algunos existen en tiempo de enlace, algunos existen en tiempo de ejecución, y algunos existen más de una vez.

Un diagrama de componentes es un grafo conectado por relaciones de dependencia de un componente a otro. Este tipo de relación de dependencia significa que las clases contenidas en el componente cliente, dependen de clases que son exportadas de componentes servidores.

Las dependencias se muestran con líneas punteadas del componente cliente al componente servidor. En la figura (5.52) se muestra la notación UML que se usa para los diagramas de componentes.

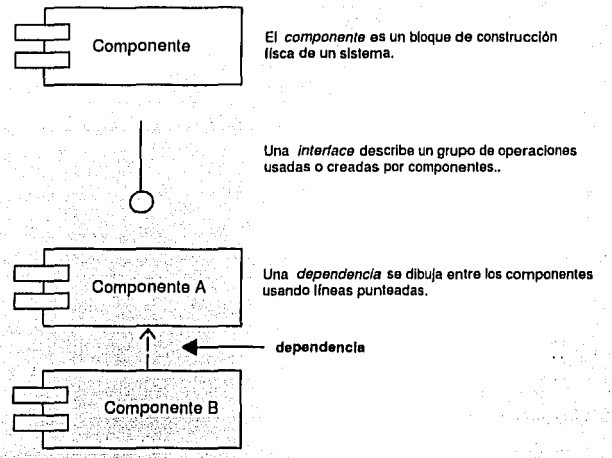


Figura 5.52. Notación para Diagramas de Componentes

Un componente puede tener interfaces. El diagrama puede mostrar esas interfaces representadas con círculos en los componentes. Si un componente presenta una interfaz significa que el componente soporta esta interfaz en particular. En la figura (5.53) se muestra la representación de un componente con interfaces.

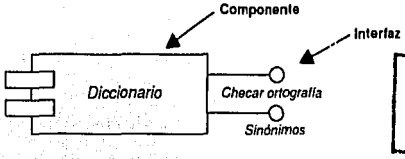


Figura 5.53. Componente con interfaces

La figura (5.54) muestra el diagrama de componentes para capturar refinerías.

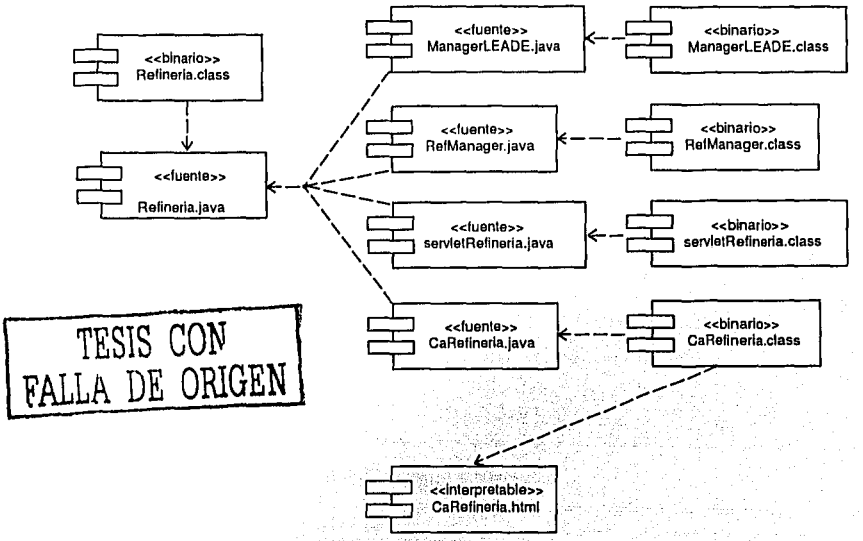


Figura 5.54. Diagrama para capturar Refinería

5.3. Diagramas de despliegue

El diagrama de despliegue muestra las relaciones físicas entre los componentes de software y de hardware en el sistema. El diagrama de despliegue es una buena forma para mostrar cómo se enrutan y se mueven los componentes y los objetos, dentro de un sistema distribuido. Las instancias de los componentes de software representan manifestaciones de unidades de código en tiempo de ejecución. Los componentes pueden contener objetos, ésto indica que el objeto reside en el componente. Un estereotipo puede usarse para indicar la dependencia precisa, si es necesario.

Un diagrama de despliegue es un grafo de nodos conectados por asociaciones de comunicación. Cada nodo representa algún tipo de recurso compu-

tacional, en la mayoría de los casos se trata de una pieza de hardware, como son: computadoras, impresoras, lectores de tarjetas, dispositivos de comunicación, un sensor simple, un mainframe, etc. Los nodos pueden contener instancias de componentes. Ésto indica que el componente vive o corre en el nodo. La figura (5.55) muestra la notación UML del diagrama de despliegue.

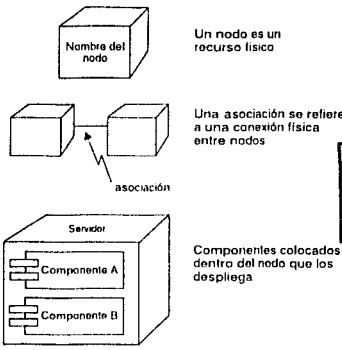


Figura 5.55. Notación para Diagramas de Despliegue

La figura (5.56) muestra un ejemplo de un diagrama de despliegue.

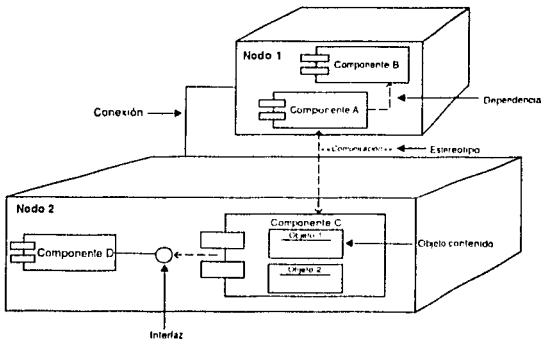


Figura 5.56. Ejemplo de un Diagrama de Despliegue

En la figura (5.57) se ilustra la interacción del sistema entre los componentes de software y hardware. Este diagrama tiene dos nodos que representan al cliente y al servidor WEB con el contenedor de `Servlets`. El nodo cliente debe soportar cualquier navegador que esté debidamente configurado para usar la JVM y los componentes `swing`. El nodo servidor WEB con el contenedor de `Servlets` se ejecuta en una plataforma Win32 o tipo UNIX (Linux, Solaris, SCO, etc.), con el ambiente de ejecución JAVA (Runtime Environment - JRE). Estos nodos se comunican a través del protocolo TCP/IP.

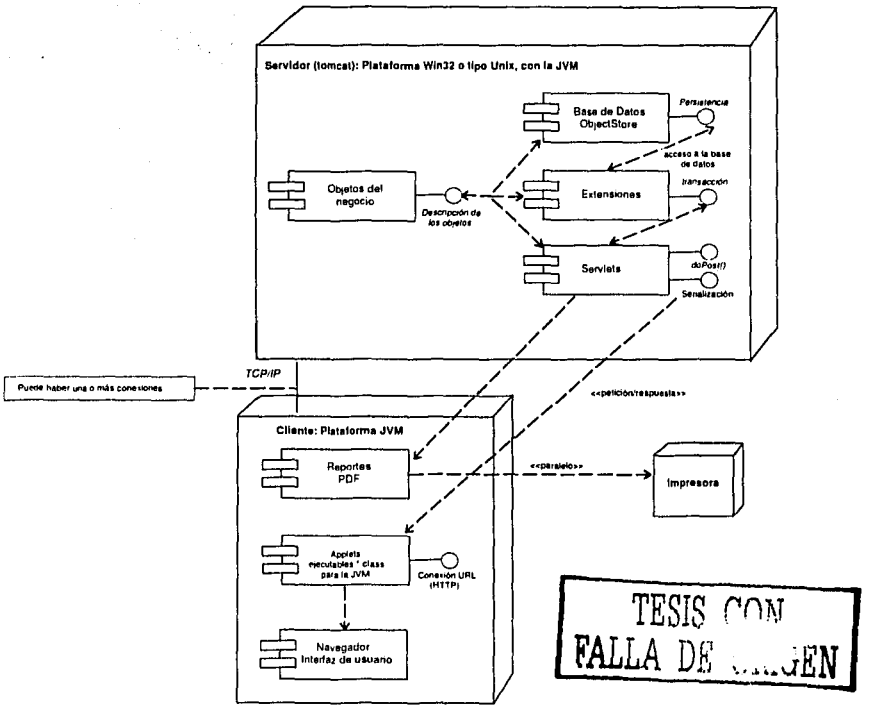


Figura 5.57. Diagrama de despliegue del Sistema

5.4. Implementación de la Arquitectura

La interfaz de usuario (IU) se visualiza a través de un navegador, el cual descarga el código HTML adecuado, este código tiene referencia al código ejecutable de los `applets` (*.class) del sistema. El navegador descarga el código binario del `Applet` en cuestión. El `applet` una vez que ha sido cargado, descarga los archivos (*.class) que sean necesarios para su ejecución.

Es necesario configurar el navegador para que soporte `applets` con código `swing`. SUN proporciona `plug-in's` para distintos navegadores.

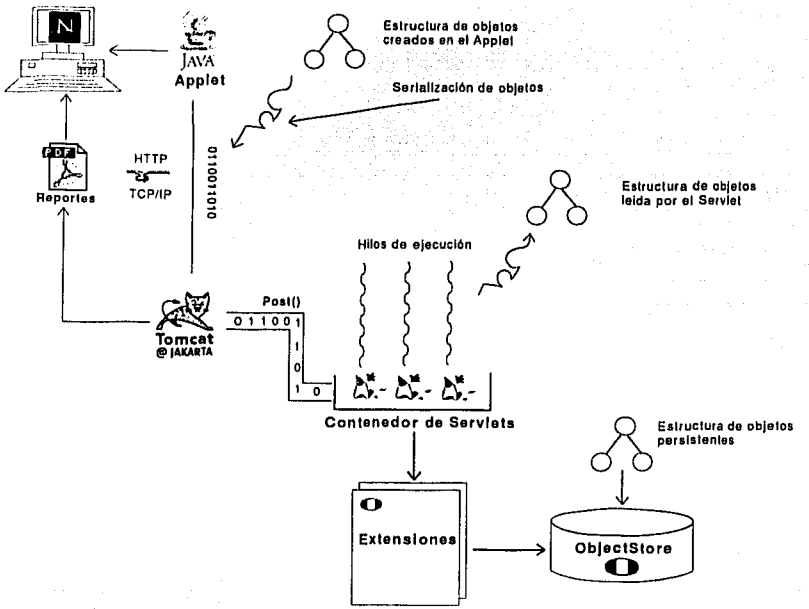


Figura 5.58. Implementación de la Arquitectura del Sistema

En la figura (5.58) los Applets corren dentro del entorno del navegador con una JVM y combinados con código HTML hacen una interfaz más dinámica y poderosa que un archivo HTML plano. Los Applets son descargados desde el servidor TOMCAT y ejecutados en el navegador del cliente (en la máquina del usuario final), son actualizados automáticamente cada vez que el usuario vuelve a visitar el sitio WEB, eliminando lo concerniente a guardar los datos de la aplicación en cada cliente. Además se encargan de verificar restricciones de integridad en los datos, por ejemplo, el formato de las fechas.

Los Servlet se encargan de la gestión entre el cliente y la base de datos, es el encargado de controlar la lógica del negocio, permitiendo que el Applet sea un cliente ligero alejándose de verificar la consistencia de los datos (relaciones entre objetos).

Los servlets no verifican la integridad de los atributos de un objeto, por ejemplo no verifican si el usuario tecleó una cadena en lugar de un entero, esto es una tarea del Applet que se verifica en la máquina del cliente, impidiendo que se complique la lógica de Servlets, como sucede con clientes planos HTML y se cargue de comunicaciones innecesarias al servidor. Se pueden decir que la carga de trabajo es balanceada entre Applets y Servlets.

En base a la información que el usuario proporciona se construye el objeto o los objetos referentes a la información, el Applet por medio de un URL (Uniform Resource Locator) y la interfaz CGI, se comunica al servidor TOMCAT indicándole que el recurso que se requiere es un Servlet, el mecanismo utilizado de la interfaz CGI es el POST ya que comunica el TOMCAT con el contenedor de servlet a través de una tubería[7] permitiendo que el tipo de información enviado sea en cualquier cantidad y binaria. Una vez establecida la comunicación el applet manda al servlet un objeto o conjunto de objetos a través del mecanismo de serialización, el Servlet los recibe y los maneja para almacenarlos en la base de datos. El procedimiento para enviar del Servlet al Applet la información es similar.

[7] Mecanismo de paso de mensajes por el cual dos procesos se pueden comunicar a través de un dispositivo de grabación

Cabe mencionar que el TOMCAT por cada petición, crea una instancia del servlet, esto con el fin de poder atender un conjunto de peticiones simultáneas. Cada `Servlet` corre en su propio hilo ejecución, éste es un mecanismo bastante eficiente en el manejo de recursos y peticiones. El TOMCAT es el encargado de controlar la concurrencia de procesamiento pero el programador debe de controlar la concurrencia de los datos que van a ser almacenados.

La instancia de un `Servlet` se comunica con la extensión adecuada para manipular objetos persistentes. Una extensión es un objeto colección encargado de gestionar la persistencia de un conjunto de objetos. La extensión abre una transacción para accesar a la base de datos y realizar lo solicitado, después la extensión regresará la respuesta al `Servlet`, para que este último regrese el resultado al cliente.

Para imprimir reportes, se solicita el reporte a través de la `IU`, el `Servlet` construye un archivo PDF dinámicamente y este puede ser visualizado e impreso tal como se pretende por medio del plug-in de Acrobat Reader.

Con lo que respecta a la seguridad y validación, ésta puede ser manejada por el TOMCAT o cualquier otro servidor WEB, que permita la validación de usuarios y sesiones. Si se requiere un alto nivel de seguridad se puede adquirir un certificado que trabaje con `https://` o cualquier otro mecanismo similar. Hasta aquí se ve la ventaja de diseñar con modelo de componentes, se puede escoger el nivel de seguridad que equiera.

5.5. Instalación del Servidor

El servidor TOMCAT es nativo para alguna plataforma específica, existe para la gran mayoría de sistemas desde los basados en Windows NT hasta los tipos UNIX como Linux, Solaris, etc.

A continuación se describe como instalarlo y configurarlo para trabajar con ObjectStore, ejecutar el sistema de `Servlets` y despachar `Applets`.

Para la mayoría de los sistemas tipo UNIX tenemos que descomprimir el archivo que contiene el TOMCAT.

```
[@azul /]$ gunzip jakarta-tomcat-4.1.12.tar.gz
```

Para desagrupar el conjunto de archivos tecleamos

```
[@azul /]$ tar -vxf jakarta-tomcat-4.1.12.tar.gz
```

Si el directorio principal de TOMCAT es `tomcat` , dentro del directorio de `webapps` se crea el directorio `LEADE`, que es el directorio base donde colocaremos la aplicación . Dentro del directorio `LEADE` crearemos los directorios `COM, images, jsp, LEADE, Servlets, WEB-INF`.

En el directorio `tomcat/webapps/LEADE/LEADE` se crean los directorios `Applets` y `Clases`, dentro de directorio `Applets` colocaremos los archivos `.class` referentes a los applets así como los archivos HTML; en el directorio `Clases` colocaremos los objetos referentes al negocio que necesitan conocer los `Applets`.

En el directorio `tomcat/webapps/LEADE` se crea la estructura `COM/odi` donde se colocan las clases proporcionadas por ObjectStore que especifican las propiedades de las clases persistentes que serán descargadas por los `Applets` y `servlets`.

En el directorio `tomcat/webapps/LEADE/WEB-INF/classes/LEADE` se deben crear los directorios `Clases` y `Extensiones`, en el directorio de `Clases` se deben colocar los archivos class referentes a los servlets y objetos del negocio, en el directorio de `Extensiones` se deben colocar los archivos class de las extensiones que accesan a los objetos persistentes de la base de datos.

La estructura de directorios que maneja la versión 4.0 de TOMCAT es la siguiente:

Subdirectorio	*Contenido*
`bin`	Contiene los `scripts` para empezar y finalizar el TOMCAT
`common`	Contiene las clases y archivos `jar`, comunes para los servidores TOMCAT
`conf`	Contiene la inforamción de configuración general, tales como las definiciones del servidor
`jasper`	Contiene el motor `JSP` de TOMCAT, llamado código `Jasper` las definiciones del servidor e información del usuario
`lib`	Contiene varios archivos jar que son utilizados por TOMCAT. En UNIX, cualquier archivo de este directorio se añade al classpath de TOMCAT
`logs`	Contiene los archivos log de ejecución generados por TOMCAT
`server`	Contiene las clases y archivos jar que son comunes a todas las aplicaciones `WEB` dentro de un servidor
`src`	Contiene el código fuente del servidor TOMCAT
`webapps`	Contiene las aplicaciones `WEB` que automáticamente serán cargadas cuando empiece TOMCAT
`work`	El directorio de trabajo temporal usado por aplicaciones `WEB`

Los subdirectorios `logs` y `work` son creados al momento en que se inicia TOMCAT. Es decir, si se tiene instalado TOMCAT y no se ha iniciado el servidor, estos directorios no existirán. La figura (5.59) muestra la estructura de directorios de TOMCAT 4.0.

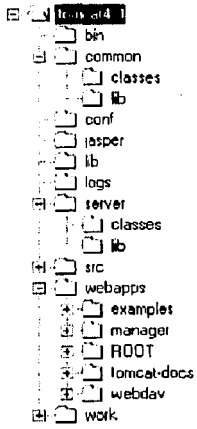


Figura 5.59. Estructura de directorios de TOMCAT

Los scripts más comunes para los usuarios son los siguientes:

script	*Descripción*
`tomcat`	El script principal, configura el entorno apropiado incluyendo el CLASSPATH, TOMCAT_HOME y JAVA_HOME
`startup`	Arranca TOMCAT
`shutdown`	Termina TOMCAT

Otro servidor gratuito para el manejo de las servlets es el `vqServer`. `vqServer` es un producto de `vqSoft` el cual contiene entre otros servicios un contenedor de `Servlets` y un manejo de seguridad con una interfaz amigable. A diferencia de TOMCAT `vqServer` no es nativo para una plataforma en específico con lo cual se ahorra el tener que utilizar un archivo binario al

cambiar de plataforma, `vqServer` es una aplicacin JAVA pura que necesita una JVM para ejecutarse.

La elección de un servidor dependerá de lo económico, del soporte, de la seguridad, de la disponibilidad y del uso.

CONCLUSIÓN

Los sistemas de información surgen de la necesidad de procesar con eficiencia la información, ésto consiste en lograr una mayor velocidad en el procesamiento de datos, consistencia, disminuir los costos, seguridad, veracidad, confiabilidad, ventaja competitiva y ampliar la comunicación, entre otras.

Existen varios modelos para procesar información, siendo el más usado, el desarrollado en 1970 por Codd que creó un modelo de datos relacional, el cual a mostrado ser muy eficiente y sencillo para manejar información hasta nuestros días. Las bases de datos relacionales basadas en el modelo relacional fueron diseñadas para manejar tipos de datos enteros, flotantes y caracter, por este motivo difícilmente pueden almacenar objetos, lo cual implica el almacenar código y datos. Por lo tanto los sistemas de bases de datos relaciones son poco expresivos cuando los datos están organizados en estructuras complejas o se encuentran en una jerarquía de herencia.

Se han diseñado sistemas donde el control y flujo de datos se manejan por medio de un lenguaje orientado a objetos, sin embargo los datos tienen que ser mapeados a una estructura de tablas que manejan las bases de datos relacionales, por consecuencia se requieren dos o más lenguajes de programación para poder construir dichos sistemas.

Una base de datos relacional reduce capacidad que un sistema totalmente orientado a objetos proporciona. Si se construye un sistema siguiendo el modelo orientado a objetos entonces debe tenerse una BDOO que siga la

esencia que la programación orientada a objetos proporciona. El modelo de la lógica del negocio debe ser igual al modelo de datos, esta es una idea nueva y moderna que lleva a una fácil utilización, escalabilidad y comprensión del modelo.

La programación orientada a objetos ha demostrado ser una metodología de programación excelente porque es extensible, reusable, etc. Las ideas de orientación a objetos están influenciando no sólo como se programa, sino como pensamos acerca de otros conceptos tales como bases de datos.

Actualmente, el creciente uso de las metodologías de programación orientada a objetos está promoviendo la aparición de manejadores de BDOO en el mercado, ya que los manejadores de BDOO tienen la flexibilidad tanto en la definición del modelo de datos como en el desempeño tan anhelado por muchos desarrolladores de aplicaciones, lo que es imposible encontrar en los modelos relacionales. Las BDOO como todas las tecnologías computacionales están supeditadas al comercio de las grandes empresas, pero no es de dudarse que a un futuro cercano empiecen a introducirse cada vez más en los sistemas actuales.

Por otro lado considero que lo mejor de la tecnología `JAVA` es su lenguaje, el cual surge de la evolución que ha tenido la tecnología orientada a objetos, los creadores de `JAVA` han desechado todo aquello que puede llevar a deficientes diseños que implican poco entendimiento y reusabilidad como por ejemplo la herencia multiple. El sistema diseñado esta totalmente escrito en `JAVA` y no requiere de ningún otro lenguaje como es el caso del `SQL`, ni tampoco conectores a base de datos como un `ODBC`, mucho menos de motores costosos y que consumen una gran cantidad de recursos como `Oracle` e `Informix`.

Gracias a todo esto, el sistema es portable, escalable, modular, extensible y reutilizable. El sistema puede ser ejecutado colocándolo en un servidor gratuito o comercial sin necesidad de reescribir o recompilar una sola línea de código.

La arquitectura más utilizada en los sistemas actuales es la cliente/servidor (dos capas) y ha mostrado tener los siguientes problemas:

- Los clientes deben mantener la integridad del sistema por lo tanto son bastante complejos
- Por cada cliente que se quiera añadir a interactuar con el sistema es necesario replicar la aplicación. Por ejemplo una aplicación cliente escrita en `Visual Basic` debe ser replicada en cada máquina cliente.
- Es poco escalable
- Es difícil tener clientes heterogéneos, etc.

Por ejemplo, en una arquitectura de dos capas, se pueden usar `Applets` del lado del cliente, presentando el inconveniente de codificar todo el acceso a la información en el código del `Applet`, en este caso los `Applets` se deben comunicar con la base de datos a través de un `ODBC` u otro mecanismo.

La arquitectura de tres capas que se utilizó para el sistema, supera los problemas que la arquitectura de dos capas presenta. La arquitectura de tres capas permite la fácil creación de clientes heterogéneos, es escalable, modular, etc. Esta arquitectura agrega una capa intermedia entre el cliente y la base de datos, esto permite que haya una clara separación entre la interfaz de usuario y la lógica del negocio y se protejan los datos del acceso directo por el cliente. A pesar de que esta arquitectura supone un cliente ligero, debe haber un balance entre lo que hace el cliente y lo que hace la capa intermedia.

Por ejemplo, si el cliente es un `Applet`, éste se encargará de corroborar que los datos sean válidos sin necesidad de mandarlos al servidor, los `Servlets` quitan el trabajo al `Applet` de comprender la lógica del negocio y la conexión con la base de datos a través de las extensiones, la seguridad del sistema se le deja al servidor `WEB`.

El software ha cambiado mucho. Para entender esta evolución, los programadores están dedicándose a las técnicas orientadas a objetos. Desafortunadamente, las bases de datos convencionales no están diseñadas para almacenar objetos y almacenar los datos de un programa orientado a objetos en una base de datos convencional, incrementa la complejidad del sistema.

El creciente uso de las metodologías de programación orientadas a objetos están promoviendo la aparición de manejadores de BDOO en el mercado. Las BDOO están diseñadas para simplificar la programación orientada a objetos y representan la mejor alternativa para la solución de problemas en las áreas donde se desea contar con un concepto de orientación a objetos agregando las ventajas de una base de datos como lo es la persistencia y la facilidad de recuperación de información.

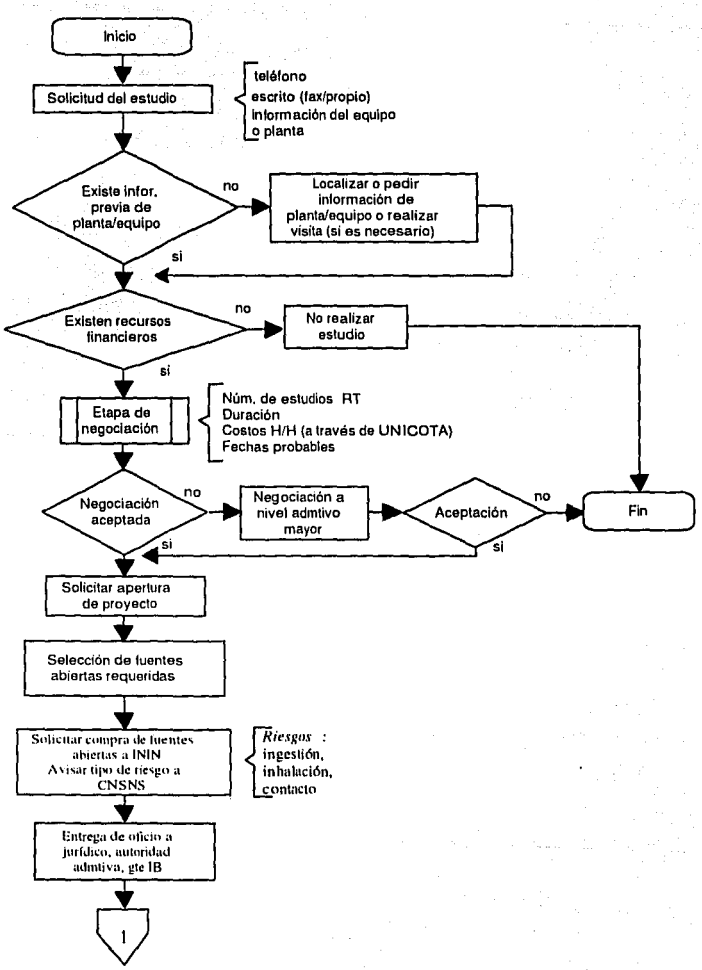
Apéndice A

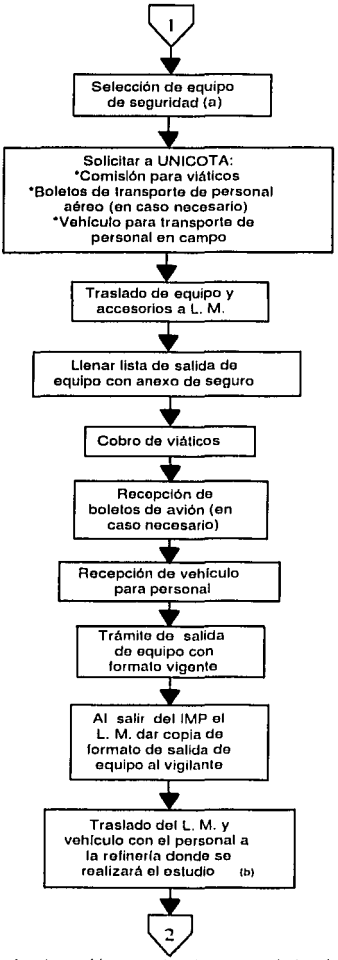
Diagramas de Flujo de Datos

Los diagramas de flujo de datos muestran paso a paso, tanto los procedimientos que LEADE sigue para llevar a cabo las actividades en esta área, como los procedimientos que se utilizan para realizar los estudios en las refinerías.

Los diagramas de flujo de datos son útiles para comprender la lógica del negocio y por lo tanto delimitar los requerimientos del sistema.

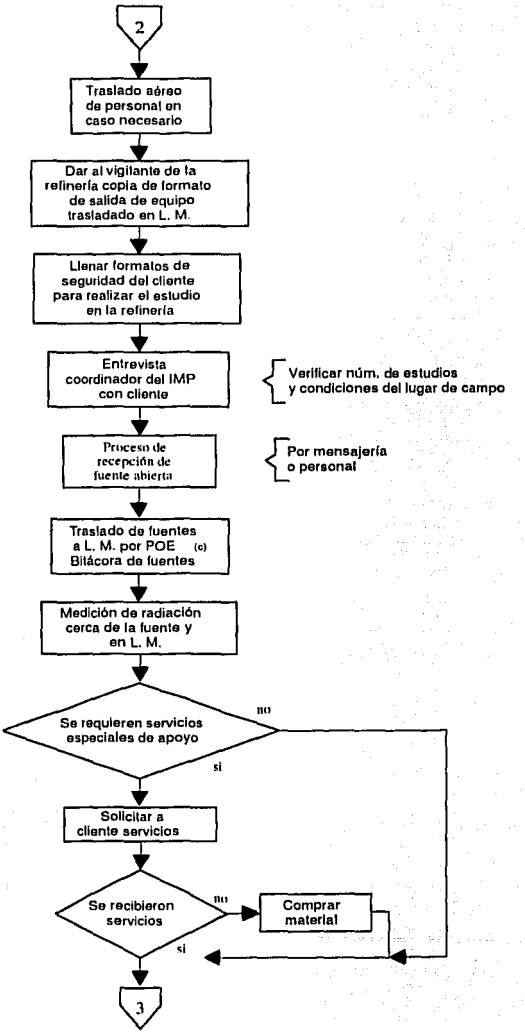
A.1. Diagrama de flujo de datos para el método de radiotrazado



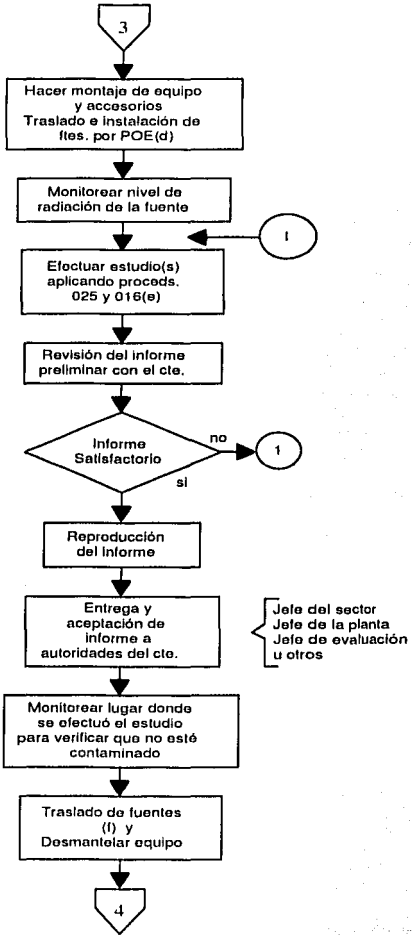


(a) Dosímetros (TLD/pluma), cascos, anteojos, tapa oídos, guantes de carnasa, batas de algodón, overoles, zapatos de seguridad, etc.

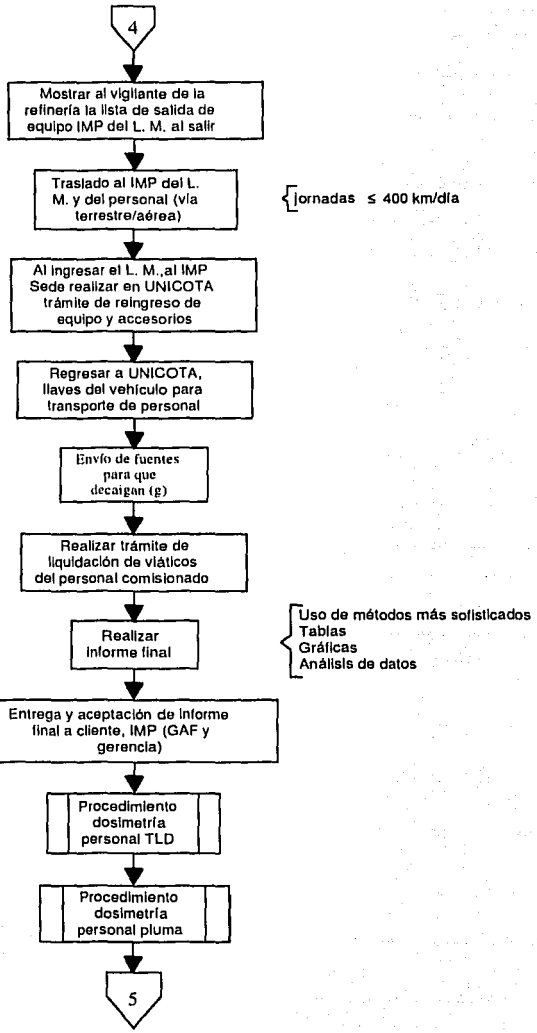
(b) El Laboratorio Movil se trasladará en jornadas $\leq$ 400 km/día



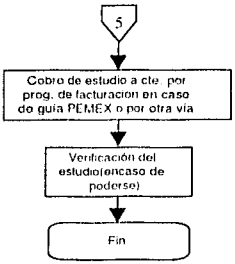
(c) POE.- Personal Ocupacionalmente Expuesto. El personal debe trasladar las fuentes portando equipo de seguridad correspondiente y dosímetros (TLD/pluma).



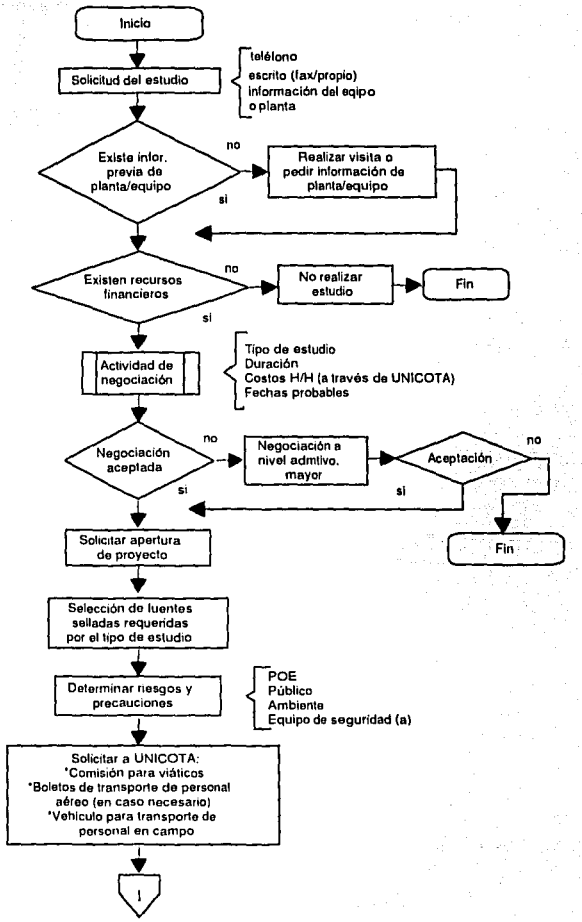
(d) Al trasladar las fuentes el personal deberá portar equipo de seguridad y dosímetros (TLD/pluma)
(e) Procedimiento de protección: LEADE 016 Seguridad radiológica en el Radiotrazado (RT), Procedimiento de trabajo LEADE 025 Radiotrazado (RT)
(f) El traslado se puede realizar desde campo a ININ por mensajería o trasladar las fuentes en el L. M. al IMP para que decaigan en pozo.



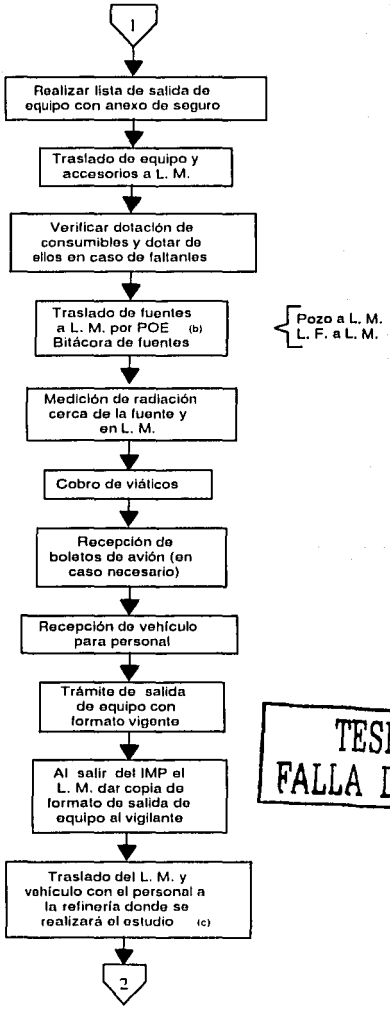
(g) Las fuentes a pozo o a ININ


5
Cobro de estudio a cte. por prog. de facturacion en caso de guia PEMEX o por otra via
Verificación del estudio(encaso de poderse)
Fin

A.2. Diagramas de flujo de datos para los métodos GLE, Balance, Nivel y Tomografía

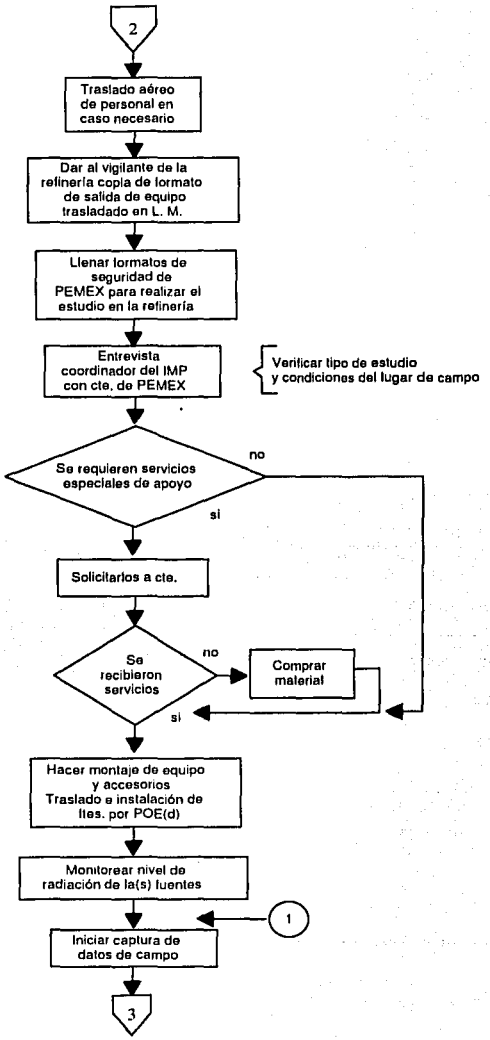


(a) Dosímetros (TLD/pluma), cascos, anteojos, tapa oídos, guantes de carnasa, batas de algodón, overoles, zapatos de seguridad, etc.

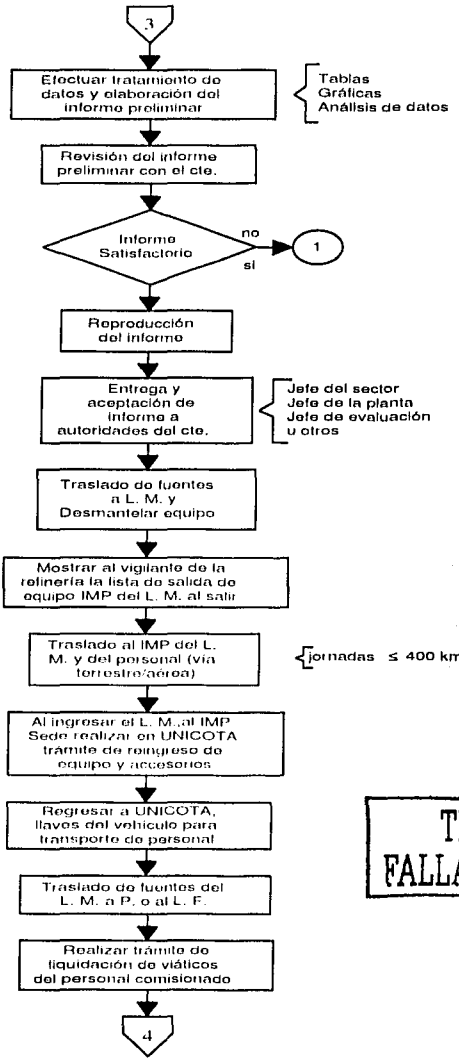


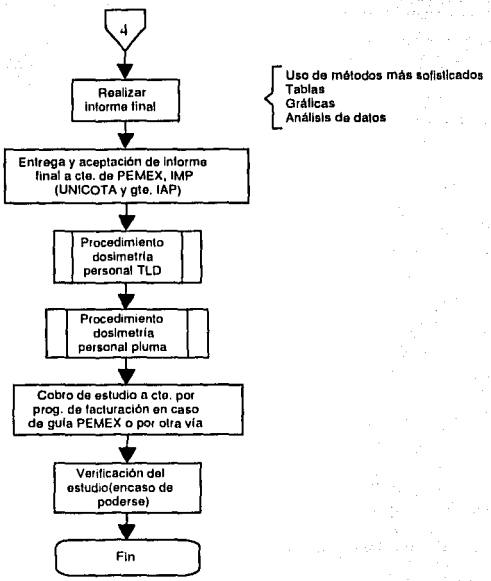
(b) POE.- Personal Ocupacionalmente Expuesto. El personal debe trasladar las fuentes portando equipo de seguridad correspondiente y dosímetros (TLD/pluma).

(c) El Laboratorio Movil se trasladará en jornadas $\leq$ 400 km/día



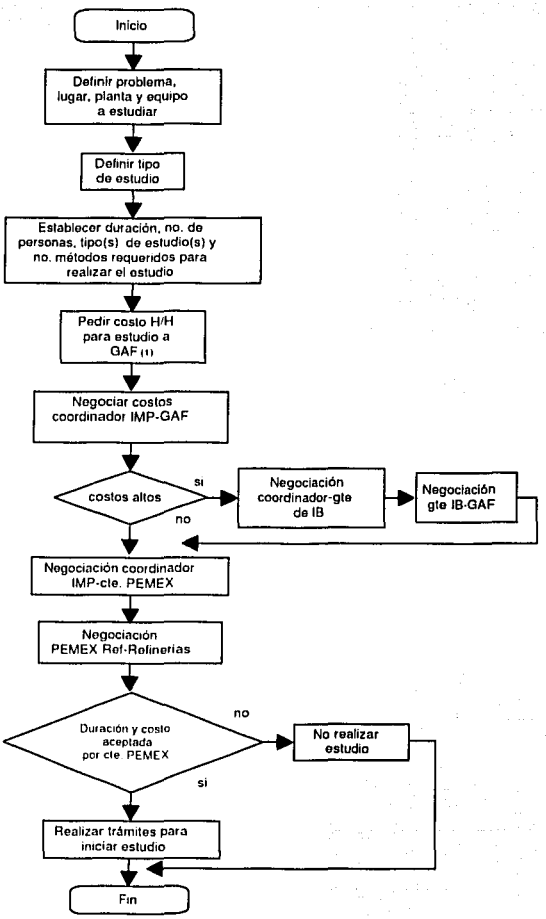
(d) Al trasladar las fuentes el personal deberá portar equipo de seguridad y dosímetros (TLD/pluma)





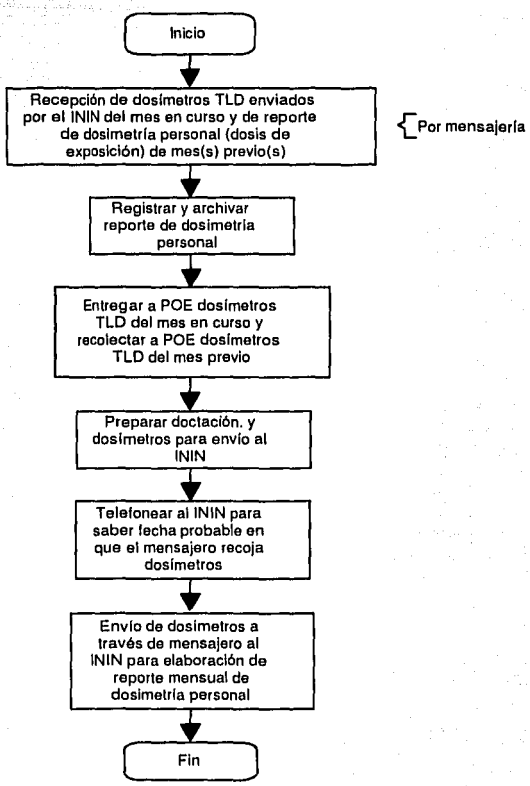
L. M. Laboratorio Movil
L. F. Laboratorio Fijo
P. Pozo
Nota.- En los métodos Balance, GLE, Nivel y Tomografía se deben cumplir las normas del IMP y de la CNSNS

A.3. Diagrama de flujo de datos para la actividad de negociación



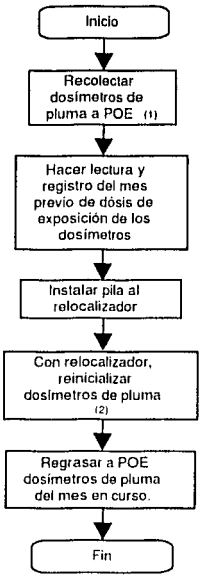
(1) GAF. Gerencia de Administración y finanzas.

A.4. Diagrama de flujo de datos para la manipulación de dosímetros TLD



ININ . Instituto Nacional de Investigaciones Nucleares.

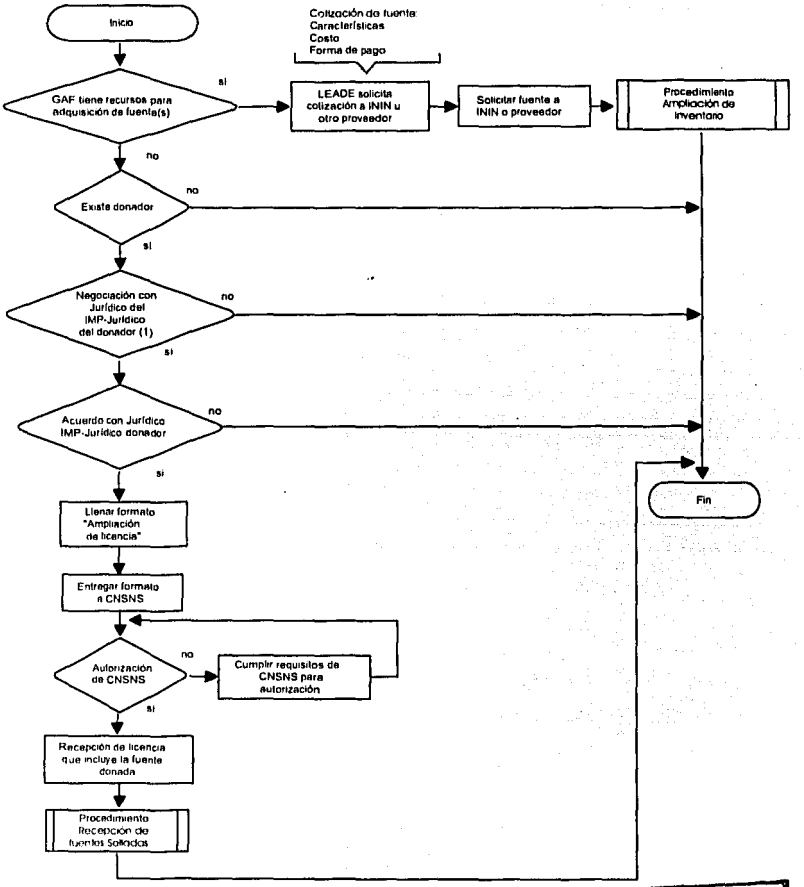
A.5. Diagrama de flujo de datos para la manipulación de dosímetros de PLUMA



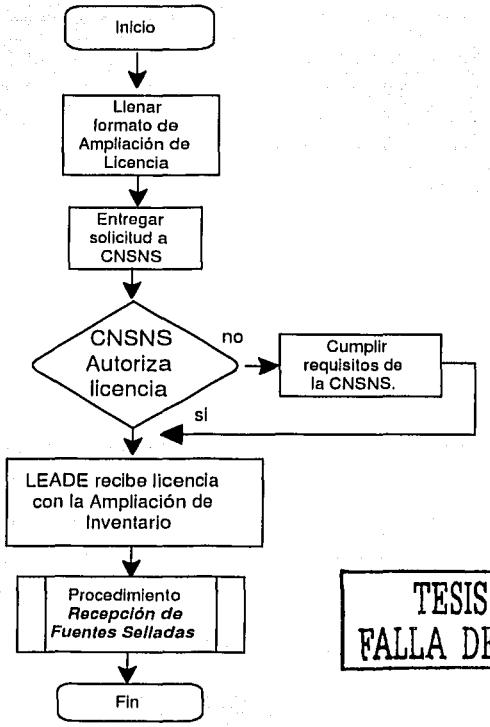
(1) POE. Personal Ocupacionalmente Expuesto.
(2) Los valores en los que deben empezar los dosímetros calibrados son de 10 mR

El relocalizador es un instrumento de metal, eléctrico que consta de una zona para colocar el dosímetro, la cual consta de una protección de ule. También tiene una zona donde se coloca la pila. Contiene una perilla para relocalizar el dosímetro al punto de partida. Con la pila se genera una señal luminosa que permite ver el marcador del dosímetro.
Las medidas del relocalizador son de 11cm de largo, 5cm de altura y 8cm de ancho.

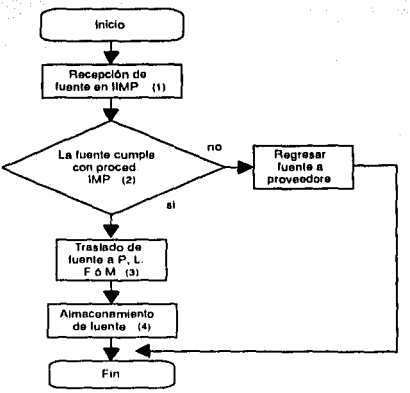
A.6. Diagrama de flujo de datos para cotización de fuentes



A.7. Diagrama de flujo de datos para ampliación de inventario

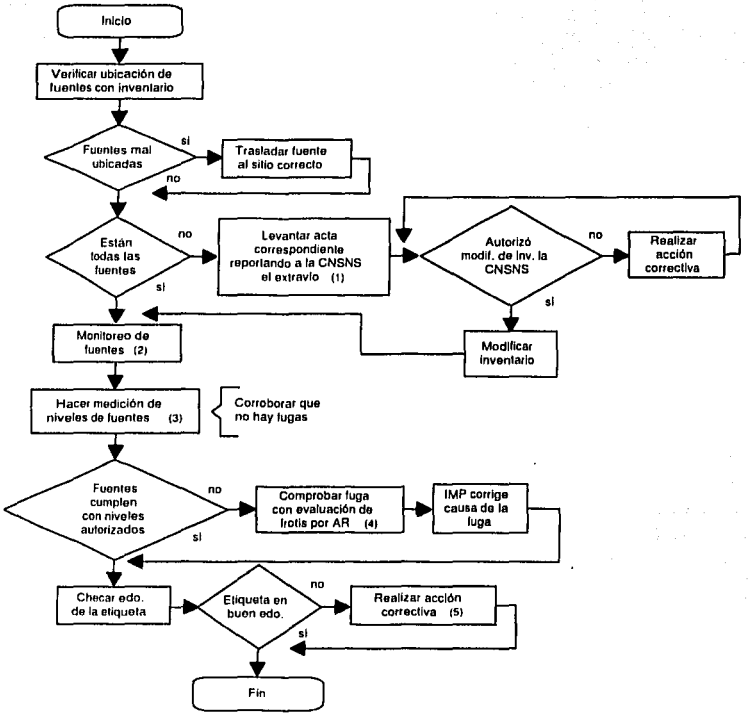


A.8. Diagrama de flujo de datos para recepción de fuentes selladas



(1) El IMP al recibir la fuente debe verificar las condiciones en que viene, checar que el bulto de la fuente no tenga emisiones radiactivas, los datos de la fuente esten correctos y el contenedor de la fuente se encuentre en buen estado.
(2) La fuente debe cumplir con los requerimientos que vienen en el procedimiento "LEADE 001 RECEPCIÓN DE LAS FUENTES SELLADAS DE RADIACIÓN IONIZANTE"
(3) P. - Pozo; L. F. - Laboratorio Fijo, M - Mössbauer.
(4) Para almacenar las fuentes se debe seguir el procedimiento "LEADE 002 ALMACENAMIENTO DE FUENTES SELLADAS"

A.9. Diagrama de flujo de datos para revisión de fuentes selladas



(1) Para llevar a cabo este proceso se utiliza el formato "LEADE /D 015 FORMATO DE REPORTE DE ACCIDENTE O INCIDENTE"
(2) Parar efectuar el monitoreo se utiliza el procedimiento "LEADE 005 PRUEBA DE FUGA A LAS FUENTES SELLADAS (TOMA DE LA MUESTRA)" y los monitroes de radiación ionizante.
(3) La medición de niveles se realiza usando el procedimiento " LEADE 006 LECTURA DE NIVELES"
(4) AR. Asesores en Radiación. Son los proveedores que se encargan de verificar si existe fuga.
(5) Si la etiqueta este legible pero despegada; sólo se pega. En caso de que no esté legible y se encuentre rota, se reemplaza por una etiqueta nueva con los datos de la fuente.
Nota.- El POE (Personal Ocupacionalmente Expuesto) efectúa la revisión de fuentes, quien debe portar el equipo de seguridad y el material necesario.
Equipo de seguridad: Dosímetros TLD y pluma, bata, guantes, etc.
Material: Plumines especiales, pegamento, etiquetas.

Bibliografía

[1] Fowler, Martin; Scott, Kendall: *UML Gota a Gota.* Editorial Addison Wesley Longman. 1997.

[2] Bruce, Eckel: *Thinking in JAVA.* Editorial Prentice Hall, 2000.

[3] Deitel, H. M.; Deitel, P. J. *Cómo programar en JAVA.* Editorial Prentice Hall, 1998.

[4] Cattel, R. G. G.; Barry, Douglas; Mark, Berler; Jeff, Eastman; Jordan, David; Russell, Craig; Schadow, Olaf; Stanienda, Torsen; Velez, Fernando.*The Object Data Standard: ODMG 3.0.* Editorial Morgan Kaufmann Publishers, 2000.

[5] Cooper, Richard: *Object Database. An ODMG Approach.* Editorial International Thomson Computer Press, 1997

[6] Bertino, Elisa; Martino, Lorenzo: *Object-Oriented Database Systems, Concepts and Architecture.* Editorial Addison Wesley, 1993.

[7] Deitel, H. M.; Deitel, P. J.; Santry, S.E: *Advanced JAVA™ 2 Platform - How to Program-.* Prentice Hall, 2002.

[8] Coad, Peter; Yourdon, Edward: *Object-Oriented Analysis.* Editorial Prentice Hall, 1991.

[9] UML, RTF: *OMG Unified Modeling Language Specifications.* Object Managmente Group Inc. 1997-2001

[10] SUN Microsystems: *Guía del desarrollador, The Java 2 Enterprise Edition Developer's Guide v. 1.2.1.* 2000.

[11] Hall, Marty: *Core servlets and JavaServer Pages.* Prentice Hall, 2000.

[12] Gutz, Steven J.: *Up to speed with Swing: user interfaces with Java fundation classes.* Manning, 1998.