15

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

Facultad de Contaduría y Administración

Desarrollo de una Herramienta para el Modelado de Datos Usando Diagramas Entidad - Relación

Tesis Profesional Que Para Obtener
El Título de:
LICENCIADO EN INFORMATICA

P r e s e n t a:

Julieta Alejandra Sánchez Olivo

A s e s o r:

M. EN I. Graciela Bribiesca Correa

México, D. F. 2001

AGRADECIMIENTOS

A la M. en I. Graciela Bribiesca Correa, por su enorme apoyo y confianza.

Al M. en C. Cristóbal Juárez Castellanos, por tu tiempo, impulso y paciencia.

A mis padres, **Carmen e Ignacio**,
por su amor infinito y la oportunidad de ser lo que yo elija.

A mis hermanas, **Sandra y Leticia**,
por su cariño y alegría.

A mi mejor amiga, **Angélica**,
por creer en mí, por tu afecto, lealtad, confianza y apoyo inagotables.

A mis amigos, **Gaby, Isa, Christian, Claudia, Laura, Caty y Samuel**,
Porque de alguna manera han tocado mi vida, haciéndola mucho mejor.

CONTENIDO

INTRODUCCIÓN

ANTECEDENTES

El desarrollo de software es una actividad altamente riesgosa debido a la complejidad que implican los sistemas de información que las organizaciones requieren en la actualidad. Uno de los grandes problemas de la industria del desarrollo de software es la falta de aplicación de metodologías, la cual produce problemas tales como retrasos en la entrega de los sistemas, insatisfacción del usuario final y altos costos del mantenimiento. La aplicación de metodologías fomenta el desarrollo sistemático y controlado de los sistemas, reduciendo significativamente los márgenes de error.

Como complemento a la aplicación de metodologías de diseño, en los últimos años han surgido diversas herramientas automatizadas mejor conocidas como ***herramientas CASE*** (*"Computer-Aided Software Engineering"*). Estas herramientas han evolucionado hasta nuestros días proporcionando ahora ambientes gráficos, un enfoque orientado a los datos, generación automática de código y esquemas, así como ***ingeniería inversa***.

Una herramienta CASE incluye típicamente un módulo o herramienta de modelado de datos, mediante la cual el desarrollador puede crear esquemas de datos. Estos esquemas deben representarse de acuerdo con una metodología o modelo para el diseño conceptual, lógico y físico de bases de datos. Uno de esos modelos es el modelo entidad – relación.

Una herramienta para el modelado de bases de datos que utiliza diagramas entidad – relación generalmente permite crear diagramas partiendo de cero, así como modificar un diagrama existente para cambiar su formato u organización, agregar entidades o interrelaciones, cambiar sus nombres, eliminarlas. Así mismo debe permitir al usuario tener una visión del modelo a diferentes niveles (conceptual, lógico y físico). A nivel lógico, la herramienta traduce del modelo entidad – relación al modelo relacional y posteriormente al nivel físico, proporcionando incluso una interfaz con algunos sistema manejadores de bases de datos comerciales.

Actualmente existen en el mercado diversas herramientas para el diseño de bases de datos. Una de ellas es ERWin, producida por Logic Works Inc. Es una herramienta de diseño basada en Windows que utiliza una interfaz gráfica para una simplificación de la notación de Chen denominada IDEF1-X. La barra de herramientas contiene operaciones para la creación de entidades independientes, dependientes, relaciones de generalización completa e incompleta y asociaciones con distinta cardinalidad. Además, ERWin muestra tres niveles de visualización del modelo (conceptual, lógico y físico) e incluye una interfaz para sistemas manejadores de bases de datos relacionales comerciales.

AMBIENTE DE DESARROLLO

El ambiente de desarrollo de la herramienta creada en esta tesis se conformó aplicando el modelo orientado a objetos, mediante el uso del lenguaje de modelado unificado o ***UML*** (*"Unified Modeling Language"*) para su análisis y diseño.

El lenguaje seleccionado para la implantación es Java, pues tiene la ventaja de ser orientado a objetos, portable, robusto e independiente de la arquitectura de hardware.

También se utilizaron varios patrones de diseño que permitieron la reutilización de código y redujeron sustancialmente el tiempo de programación.

Se utilizó un "framework" para la programación de herramientas gráficas llamado `softengine`[1]. Este framework es un conjunto de paquetes de clases en Java que incluyen clases para el manejo de interfaces gráficas de usuario (creación de ventanas, menús, etc.), clases para el manejo de componentes gráficos propios de la aplicación (entidades, interrelaciones, asociaciones), clases para el manejo de eventos (clics del mouse, selección de un elemento del diagrama), clases para el almacenamiento de datos (en archivos y bases de datos) y diversas clases de utilería.

OBJETIVO

El objetivo de esta tesis es desarrollar una herramienta para la creación de esquemas conceptuales de bases de datos relacionales utilizando diagramas E-R, que agilice y permita un mayor control del proceso de diseño de bases de datos.

ORGANIZACIÓN DE LA TESIS

La tesis está organizada en cuatro capítulos, dos de ellos proporcionan el marco teórico que sustenta la herramienta y los otros dos describen sus características de implantación.

En el capítulo 1 se explican los conceptos básicos de bases de datos necesarios para el diseño de esta herramienta y para su utilización, haciendo énfasis en el modelo Entidad - Relación para el modelado de datos.

[1] Las clases del "framework" softengine son propiedad de SoftEngine, S.A. de C.V. y se explotaron con su autorización.

En el capítulo 2 se definen los conceptos básicos del modelo orientado a objetos y se describe el lenguaje gráfico de modelado UML empleado para representar las características estáticas y dinámicas de la aplicación.

En los capítulos 3 y 4 se describe la aportación de esta tesis, es decir, la herramienta misma. A lo largo del capítulo 3 se describe la herramienta a nivel funcional, detallando las operaciones de su menú y barra de herramientas, así como los elementos gráficos que componen los diagramas que con ella se pueden crear. En el capítulo 4 se explican las estructuras de implantación de la aplicación descritas mediante los diagramas de casos de uso, de clases y de secuencia que representan su funcionalidad.

Finalmente los apéndices contienen una descripción de las clases que componen el sistema, un glosario técnico y un resumen de los conceptos del lenguaje de programación Java.

CAPÍTULO 1

BASES DE DATOS RELACIONALES

1.1. CONCEPTOS BÁSICOS

Una ***base de datos*** es una colección de datos interrelacionados almacenados en una computadora de manera más o menos permanente, que tienen las siguientes características:

1. Los datos son compartidos por diferentes usuarios y programas de aplicación, a través de un mecanismo común para la definición y manipulación de los mismos.

2. No es necesario que los usuarios finales ni los programas de aplicación conozcan las estructuras de almacenamiento [JUÁREZ].

Un ***sistema manejador de bases de datos*** o DBMS ("DataBase Management System") es el software que se utiliza para la administración y el acceso a los datos de una base de datos.

De acuerdo con [JUÁREZ], las principales funciones de un DBMS son:

- ***Abstracción***. Permite al usuario tratar con los datos en términos abstractos, sin que deba conocer la forma en que se almacenan en la computadora.

- ***Seguridad***. Controla la creación de usuarios y los niveles de acceso.

- . ***Integridad***. Permite que la información de la base de datos sea consistente, es decir, confiable.

- ***Reducción de redundancia.*** Evita el almacenamiento innecesario de la misma información en varios lugares.

- ***Compartición de datos***. Permite el acceso concurrente de varios usuarios y programas de aplicación a la base de datos.

- ***Protección contra fallas y recuperación***. Protege a la base de datos y permite su recuperación en caso de alguna falla de hardware o software.

El ***diccionario de datos*** es un conjunto de metadatos (datos relativos a los datos) que definen las estructuras de almacenamiento y la información que requiere el DBMS para su operación.

1.1.1. ARQUITECTURA DE UN SISTEMA DE BASES DE DATOS

Como se observa en la figura 1.1, la arquitectura propuesta por el Grupo de Trabajo en Sistemas de Administración de Bases de Datos ANSI/SPARC se divide en tres niveles conocidos como externo, conceptual e interno. Cada uno de ellos se define mediante un ***esquema***, es decir, la definición de la estructura de la base de datos.

En el ***esquema externo*** cada usuario percibe la base de datos como si estuviera formada únicamente por los datos de su interés. Esto se logra a través de la definición de ***subesquemas*** o ***vistas***. Un subesquema o vista es la parte que cada usuario puede ver y manejar de la base de datos completa [DATE].

El ***esquema conceptual*** se refiere a la representación global de la base de datos, esto es, es una abstracción del mundo real tal como es percibido por los usuarios. Un DBMS proporciona un lenguaje que permite describir la base de datos en términos de algún modelo de datos.

El ***esquema interno*** es una representación de bajo nivel de la base de datos y está formado por los archivos , índices y otras estructuras de almacenamiento que facilitan el acceso a los datos.

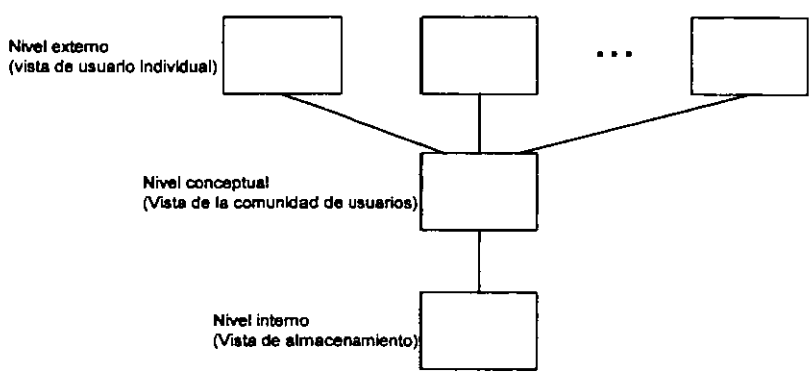

Figura 1.1. Los tres niveles de la arquitectura de un Sistema de Bases de Datos.

1.1.2. LENGUAJES DE BASES DE DATOS

La interacción con una base de datos se realiza a través de cuatro lenguajes: el lenguaje de definición de datos, el lenguaje de manipulación de datos, el lenguaje de consulta y el lenguaje de control.

El ***lenguaje de definición de datos*** o **DDL** (*"Data Definition Language"*) permite definir los esquemas interno y externo de la base de datos.

El ***lenguaje de manipulación de datos*** o **DML** (*"Data Manipulation Language"*) contiene instrucciones para recuperar, insertar, eliminar y actualizar los datos de la base de datos.

El ***lenguaje de consulta*** o **QL** (*"Query Language"*) es una parte del DML que permite recuperar información de la base de datos a través de consultas.

El ***lenguaje de control*** o **CL** (*"Control Language"*) es el que permite la administración de usuarios y control de niveles de acceso.

El lenguaje estándar en bases de datos relacionales es el ***SQL*** ("Structured Query Language"), un estándar definido por la ANSI que incluye instrucciones propias del DDL, DML, QL y CL.

1.2. MODELOS DE DATOS

En la ingeniería de software, como en todas las ramas de la ingeniería, se utilizan modelos. Un ***modelo*** es una representación de la realidad que destaca los aspectos importantes para el problema a resolver (el dominio del discurso). En una gran mayoría de los casos, el problema a resolver es la construcción de una aplicación.

Existen tres tipos de elementos que un modelo puede representar de una aplicación [BRODIEa]:

1. ***Propiedades estáticas***. Los objetos, sus propiedades y las interrelaciones que existen entre ellos.

2. ***Propiedades dinámicas***. Las operaciones sobre los objetos, las propiedades de esas operaciones y las interrelaciones que hay entre ellas.

3. ***Restricciones de integridad***. Reglas que restringen el dominio de algunas propiedades estáticas y/o dinámicas de la aplicación.

Para representar tales características se utiliza un ***modelo de datos*** que, de acuerdo con [BATINI], es "una serie de conceptos que puede utilizarse para describir un conjunto de datos y operaciones para manipular los mismos".

Los modelos de datos más utilizados actualmente son el ***modelo relacional***, que es un modelo lógico utilizado a nivel de implantación en los DBMS y el modelo ***entidad–relación***, el cual es un modelo conceptual de tipo semántico, que representa una visión de alto nivel de abstracción, independiente de la implantación.

1.3. EL MODELO RELACIONAL

Los sistemas administradores de bases de datos más populares en la actualidad (como Oracle, Sybase, Informix, etc.) están basados en el modelo relacional. Este modelo fue planteado por [CODD] a principios de los años 70 y se caracteriza por ser simple, potente y estar definido formalmente con una base matemática.

El concepto básico del modelo relacional es la ***relación***, que en términos sencillos es un conjunto de tuplos o n-adas [JUÁREZ]. Un ***tuplo*** es un conjunto de pares atributo-valor. Un ***atributo*** es una propiedad que caracteriza a una entidad.

Al conjunto de valores del cual un atributo puede asumir un valor se le denomina ***dominio***, por lo que matemáticamente se dice que una relación es un subconjunto del producto cartesiano de una lista de dominios (los correspondientes a su conjunto de atributos).

Existen dos métricas sobre las relaciones:

- ***Grado o aridad***, que es el número de atributos y
- ***Cardinalidad***, que es el número de tuplos.

1.3.1. BASES DE DATOS RELACIONALES

Una ***base de datos relacional*** es una colección de relaciones [JUÁREZ].

De acuerdo con [CODD], para que un DBMS pueda ser considerado completamente relacional debe cumplir con las siguientes reglas:

1. ***Información.***

   Toda información en una base de datos está representada explícitamente en el nivel lógico de exactamente una forma: por valores en relaciones.

2. ***Acceso garantizado.***

   Todos y cada uno de los datos (valores atómicos) en una base de datos relacional deben ser lógicamente accesibles mediante un nombre de relación, un valor de llave principal y un nombre de atributo.

3. ***Tratamiento sistemático de valores NULL.***

   Deben soportarse los valores nulos (NULL, que no son cadenas vacías de caracteres o cadenas de caracteres en blanco y son distintos de cero o cualquier otro número) para representar información faltante en una forma sistemática, independiente de un tipo de dato.

4. ***Catálogo en línea y dinámico basado en el modelo relacional.***

   La descripción de la base de datos se representa a nivel lógico en la misma manera que los datos ordinarios, de manera que los usuarios autorizados puedan aplicar el mismo lenguaje relacional que utilizan con el resto de los datos para consultarlo.

5. ***Sublenguaje de datos sencillo.***

   Un sistema relacional puede soportar varios lenguajes y varios modos de uso terminal (por ejemplo, formas de captura). Sin embargo, debe haber al menos un lenguaje cuyas sentencias se puedan expresar con alguna sintaxis bien definida, como cadenas de caracteres y cuya habilidad para soportar lo

siguiente sea sencilla: definición de datos, definición de vistas, manipulación de datos (interactiva y por programa), restricciones de integridad y límites de transacciones (begin, commit y rollback).

6. ***Actualización de vistas.***

Todas las vistas que teóricamente sean actualizables serán actualizables también a través del sistema.

7. ***Inserción, actualización y borrado de alto nivel.***

La capacidad de manipulación de una relación base o derivada como un operando simple aplica no sólo a la consulta de datos, sino también a la inserción, actualización y borrado.

8. ***Independencia física de los datos.***

Los programas de aplicación y actividades terminales permanecen lógicamente intactas cuando se realizan cambios sobre la representación del almacenamiento o los métodos de acceso.

9. ***Independencia lógica de los datos.***

Los programas de aplicación y actividades terminales permanecen lógicamente intactas cuando se realizan cambios sobre relaciones base que contengan información que teóricamente así lo permita.

10. ***Independencia de integridad.***

Las restricciones de integridad específicas a una base de datos relacional deben poder definirse en el sublenguaje de datos relacional y almacenarse en el catálogo, no en los programas de aplicación. Al menos dos restricciones de integridad deben ser soportadas:

- ***De entidad.*** Ningún atributo que forme parte de la llave principal deberá contener un nulo.

- ***Referencial.*** Para cada llave foránea no nula distinta debe existir un valor de llave principal perteneciente al mismo dominio.

11. Independencia de distribución.

Un DBMS relacional tiene independencia de distribución, es decir, los usuarios no deben darse cuenta si la base está distribuida.

12. No subversión.

Si un sistema relacional tiene un lenguaje de bajo nivel (de un registro a la vez), ese lenguaje no deberá poder utilizarse para ignorar o quebrantar las reglas de integridad o restricciones expresadas en el lenguaje relacional de nivel más alto (múltiples registros a la vez).

Adicionalmente define la regla 0 (cero), que determina que para cualquier DBMS anunciado como o que se precie de ser relacional, ese sistema deberá ser capaz de manejar los datos completamente a través de sus capacidades relacionales.

Un ***esquema relacional*** es una descripción de la estructura de las relaciones que forman una base de datos relacional.

Una ***instancia*** de una base de datos es el conjunto de los valores almacenados en la base de datos en un momento determinado.

1.3.2. DISEÑO DE BASES DE DATOS RELACIONALES

Sin un diseño adecuado, una base de datos puede presentar anomalías para la inserción, actualización y borrado, así como pérdida de información. La ***pérdida de información*** es la obtención de información falsa al realizar operaciones sobre la base de datos.

Para evitar tales problemas se utilizan técnicas y principios de diseño de bases de datos tales como las dependencias lógicas (funcionales y multivaluadas) y la normalización.

DEPENDENCIAS FUNCIONALES

Básicamente, las dependencias funcionales son relaciones entre dos conjuntos de atributos dentro de una relación y representan restricciones de integridad.

Sea R una relación, X y Y subconjuntos arbitrarios de atributos de R. Se dice que Y es ***funcionalmente dependiente*** de X, representado como $X \rightarrow Y$, sí y solo si cada valor de X en R está asociado con exactamente un valor de Y en R [DATE]. Esto significa que cualesquiera dos tuplos de R que coinciden en el valor de X, deben coincidir también en su valor para Y.

Una dependencia funcional que es siempre verdadera se conoce como ***trivial***. Formalmente se dice que una dependencia funcional es trivial cuando el conjunto de atributos Y (cuando $X \rightarrow Y$) es un subconjunto propio del conjunto de atributos X. De otra forma la dependencia funcional es ***no trivial***.

Una dependencia funcional $X \rightarrow Y$ es ***total*** si la eliminación de cualquier atributo $A \in X$, $(X - \{A\})$ no determina funcionalmente a Y. En tanto que una dependencia funcional es ***parcial*** ($X \not\rightarrow Y$) si algún atributo $A \in X$ puede eliminarse de X y la dependencia sigue siendo válida.

Una dependencia ***transitiva*** se define de la siguiente manera: sean X, Y y Z conjuntos de atributos de la relación R(A1, A2, ..., An), decimos que Z depende transitivamente de X si y sólo si se cumplen las siguientes condiciones:

$$X \rightarrow Y$$
$$Y \not\rightarrow X$$
$$Y \rightarrow Z$$
$$X \rightarrow Z$$

Donde la dependencia funcional $Y \rightarrow Z$ no es trivial.

LLAVES

Una ***llave*** es un conjunto de atributos que identifica de manera única a cada tuplo de una relación [DATE].

Sea R una relación. Una ***llave candidata*** para R es un subconjunto del conjunto de atributos de R, digamos X, tal que:

1. No hay dos tuplos distintos de R que tengan el mismo valor para X.
2. No hay un subconjunto propio de X que cumpla con la anterior condición.

Una ***llave principal*** o ***primaria*** es una llave candidata elegida para ser el identificador único de una relación. El resto de las llaves candidatas se conocen como llaves alternas [DATE].

Si X es una llave candidata de R entonces todos los atributos de R dependen funcionalmente de X.

DEPENDENCIAS MULTIVALUADAS

Las dependencias funcionales son un caso particular de un tipo más general de dependencias lógicas denominadas ***dependencias multivaluadas***.

Sea R[XYZ] una relación de orden m + n + r, donde X = (X_1, X_2, ..., X_m), Y=(Y_1, Y_2, ..., Y_n) y Z=(Z_1, Z_2, ..., Z_r) son conjuntos de atributos que tomados por pares son disjuntos. Considérese que:

- Un tuplo (x_1, x_2, ..., x_m) se representa como *x*,
- Un tuplo (y_1, y_2, ..., y_n) se representa como *y*
- Un tuplo (z_1, z_2, ..., z_r) se representa como *z*.
- Y_{xz} representa los tuplos definidos como $Y_{xz} = \{y|(x,y,z) \in R\}$
- El conjunto Y_{xz} se constituye por medio de las operaciones de restricción y proyección de la relación R.

Una ***dependencia multivaluada*** $X \twoheadrightarrow Y$ se cumple para una relación R(XYZ) si y sólo si $Y_{xz} = Y_{xz'}$ para cualquier x, z y z' para los cuales Y_{xz} y $Y_{xz'}$ son conjuntos no vacíos de atributos; es decir, si (x, y, z) y (x, y', z') son tuplos de la relación R entonces (x, y', z) y (x, y, z') también son tuplos de R [JUÁREZ].

NORMALIZACIÓN

Un principio de diseño de bases de datos relacionales establece que toda entidad debe estar representada mediante una relación separada. Las ***formas normales*** o ***NF*** (*"Normal Forms"*) son reglas formales para determinar si una relación representa una sola entidad o no [ALAGIC].

El ***proceso de normalización*** es una reducción sucesiva de una colección de relaciones dada a alguna forma más deseable, a través de la descomposición de las relaciones de la colección original. Este proceso se caracteriza por ser ***reversible***, es decir, siempre es posible tomar la solución del procedimiento y obtener las relaciones originales. La importancia de esta propiedad radica en que no hay pérdida de información en el proceso de ***desnormalización***, es decir, al aplicar el proceso inverso [DATE].

Existen varias formas normales: primera, segunda, tercera, de Boyce–Codd y cuarta, que son las principales; además existen otras de más reciente creación como quinta, la de dominio–llave, la de restricción–unión, etc.

Primera forma normal (1NF)

Una relación está en primera forma normal (1NF) si y sólo si todos los dominios de los cuales los atributos toman sus valores son conjuntos de valores atómicos [JUÁREZ].

Segunda forma normal (2NF)

Sea X el conjunto de todos los atributos de la relación $R(A_1, A_2, ..., A_n)$ que no participan en ninguna llave de R. Se dice que la relación R está en segunda forma normal si y sólo si está en 1NF y cada atributo depende funcionalmente en forma total de cada llave de R [DATE].

Tercera forma normal (3NF)

Una relación R está en tercera forma normal si y sólo si está en 2NF y ninguno de los atributos no primarios depende transitivamente de alguna llave de R [DATE].

Forma normal de Boyce - Codd (BCNF)

Las relaciones que están en 3NF pueden presentar algunas anomalías de inserción, actualización y borrado. La forma normal de Boyce – Codd (BCNF) elimina esas anomalías.

La relación $R(A_1, A_2, ..., A_n)$ está en BCNF si y sólo si la existencia de una dependencia funcional no trivial $X \rightarrow Y$, donde X y Y son conjuntos de atributos de R, implica la existencia de una dependencia funcional $X \rightarrow A_i$, para toda $i = 1, 2, ..., n$.

Cuarta forma normal (4NF)

Una relación R(X, Y, Z) donde X, Y y Z son conjuntos de atributos que por pares son disjuntos, está en 4NF si y sólo si la existencia de una dependencia multivaluada no trivial $X \twoheadrightarrow Y$ implica la existencia de una dependencia funcional $X \rightarrow A_i$, para todos los atributos A_i de R [ALAGIC].

1.4. EL MODELO ENTIDAD-RELACIÓN

El Modelo Entidad–Relación fue propuesto por Peter Chen en 1976 [CHEN] y se basa en la visión del mundo real como un conjunto de elementos básicos denominados entidades y las interrelaciones que existen entre ellos. Su finalidad principal es facilitar el diseño de bases de datos permitiendo representar su esquema conceptual [BATINI].

1.4.1 ENTIDADES Y CONJUNTOS DE ENTIDADES

Una ***entidad*** es un objeto que existe en el mundo real y es identificable de manera única [CHEN].

Los ***conjuntos de entidades*** son colecciones de entidades del mismo tipo. Los conjuntos de entidades no son necesariamente disjuntos; por ejemplo, el conjunto de todas las personas contiene al conjunto de todos los empleados de una empresa. Por lo general los términos entidad y conjunto de entidades se utilizan indistintamente.

1.4.2 INTERRELACIONES Y CONJUNTOS DE INTERRELACIONES

Una ***interrelación*** es una asociación entre 2 o más entidades [CHEN]. Un ***conjunto de interrelaciones*** es una colección de interrelaciones del mismo tipo.

Sean E_1, E_2,..., E_n conjuntos de entidades, entonces un conjunto de interrelaciones R es un subconjunto de:

$$\{(e_1, e_2, ..., e_n) \mid e_1 \in E_1, e_2 \in E_2, ..., e_n \in E_n\}$$

donde $(e_1, e_2, ..., e_n)$ es una interrelación. Cuando n=2 se dice que la interrelación es binaria.

Al igual que en el caso de las entidades, es común que los términos interrelación y conjunto de interrelaciones se utilicen indistintamente.

Como miembro de una interrelación, una entidad desempeña un ***papel***. Por ejemplo, la interrelación PADRE_DE relaciona a dos entidades de tipo PERSONA, en la que una entidad juega el papel de PADRE y otra desempeña el papel de HIJO.

1.4.3 ENTIDADES E INTERRELACIONES DÉBILES.

Cuando una entidad tiene los atributos necesarios para integrar su llave principal se le conoce como ***entidad fuerte***, de lo contrario se trata de una ***entidad débil***, ya que se identifica a través de su interrelación con otra entidad. La llave principal de una entidad débil está formada por la llave principal de la entidad fuerte de la que depende más los atributos de su propia llave.

Una ***interrelación débil*** es aquella en que participa al menos una entidad débil.

1.4.4 RESTRICCIONES

Entre las principales restricciones de un sistema de información se encuentran la cardinalidad y la integridad referencial u obligatoriedad.

La ***cardinalidad*** es una restricción que expresa el número de entidades con las que se puede asociar una entidad a través de una interrelación. Para un conjunto de interrelaciones binario R entre dos conjuntos de entidades E1 y E2, la cardinalidad puede ser:

- 1 : 1. Una entidad en E_1 está asociada con una sola entidad en E_2, y viceversa.
- 1 : n. Una entidad en E_1 está asociada con varias entidades en E_2, pero una entidad en E_2 está asociada con sólo una entidad en E_1.
- m : n. Una entidad en E1 puede estar asociada con más de una entidad en E2 y viceversa.

Las ***restricciones de integridad referencial*** se implantan mediante referencias de los tuplos de una relación a los de otra, mediante el uso de llaves foráneas.

Sean R1 y R2 dos relaciones no necesariamente distintas. Una ***llave foránea*** FK es un conjunto de atributos de R2 tal que ese mismo conjunto de atributos existe en R1 y constituye la llave principal de R1.

1.4.5 MECANISMOS DE ABSTRACCIÓN

La ***abstracción*** es un proceso mental mediante el cual se exaltan las características y propiedades relevantes de un problema para su resolución [BATINI]. Entre las abstracciones más utilizadas en el modelo entidad–relación se encuentran la clasificación, la agregación y la generalización.

La ***clasificación*** se usa para definir un concepto como un tipo de entidades del mundo real, caracterizados por propiedades comunes. Es a través de esta abstracción que se considera una entidad como parte de un conjunto de entidades.

La ***agregación*** define una entidad nueva a partir de un conjunto de otras entidades que representan sus partes componentes. Una entidad agregada es aquella interrelación que se considera como una entidad de más alto nivel.

La ***generalización*** es una abstracción que define una relación de subconjunto entre entidades de dos o más tipos. Al establecer una la relación de ***generalización–especialización*** entre entidades, existen una o varias entidades especializadas (con propiedades específicas) que comparten una serie de propiedades comunes con una entidad genérica.

Por ejemplo, en el mundo real un estudiante universitario y un empleado son personas y por lo tanto tienen un nombre, una dirección, un teléfono, una fecha de nacimiento. Sin embargo, cuando una persona desempeña el papel de empleado tiene atributos adicionales, tales como su RFC, la fecha de su contratación, su salario y el departamento para el cual trabaja. De igual forma, cuando una persona es considerada como estudiante universitario, tiene otros atributos relevantes como su número de cuenta, la carrera que está cursando, su grupo y grado. La entidad PERSONA es en este caso una entidad generalizada, cuyas especializaciones son UNIVERSITARIO y EMPLEADO.

1.4.6 DIAGRAMAS ENTIDAD–RELACIÓN

Los ***diagramas entidad–relación*** o ***DER*** constituyen una técnica para representar la estructura lógica de una base de datos en forma gráfica, proporcionando un medio simple y fácil de leer de las características del diseño para cualquier base de datos.

A continuación se describe la simbología propuesta por Chen para un DER.

Una ***entidad*** se representa por medio de un rectángulo etiquetado con su nombre (figura 1.3).

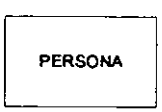

Figura 1.3. Icono de entidad.

Una ***interrelación*** se representa mediante un rombo también etiquetado con su nombre (figura 1.4).

Figura 1.4. Icono de interrelación.

Una ***entidad débil*** se representa como una entidad, sólo que con doble contorno (figura 1.5).

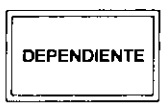

Figura 1.5. Icono de entidad débil.

Una ***interrelación débil*** se representa como una interrelación con doble contorno (figura 1.6).

Figura 1.6. Icono de interrelación débil.

La ***entidad agregada*** se representa como una interrelación contenida dentro de una entidad (figura 1.7).

Figura 1.7. Representación de entidad agregada.

La ***generalización-especialización*** se representa mediante una asociación con un triángulo que parte del conjunto de entidades especializado y que apunta hacia el conjunto de entidades genérico. En la figura 1.8 se muestra el diagrama correspondiente al ejemplo de generalización-especialización anteriormente descrito.

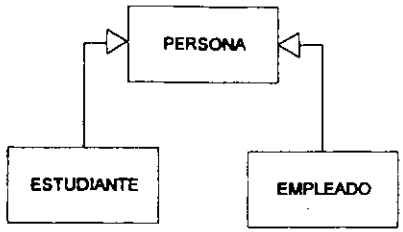

Figura 1.8. Representación gráfica de la generalización-especialización.

Los ***papeles*** se muestran etiquetando las líneas de asociación entre las entidades y las interrelaciones, mientras que las asociaciones entre los elementos del diagrama se representan como líneas rectas que pueden tener algunas de las siguientes indicaciones:

- Una marca del lado de una entidad indicando la cardinalidad.
- Una línea pequeña que cruza la asociación del lado de una entidad, indicando obligatoriedad.
- Un círculo relleno en el extremo del lado de una entidad, el cual indica que la lectura de la interrelación inicia en esa entidad.

Figura 1.9. Representación de papeles y asociaciones.

Para que un DER pueda considerarse correcto debería cumplir con las siguientes características:

- Ser tan simple como sea posible, pero no más.
- Ser completo, capturando todos los hechos relevantes.
- Capturar hechos relevantes solamente, sin detalles innecesarios.
- Usar sustantivos singulares como nombres de entidades.
- Usar verbos o acciones como nombres de interrelaciones.

- Tener llaves principales para todas las entidades.
- No tener atributos multivaluados.
- No mostrar redundancia innecesaria.
- Tener cardinalidades para todas las interrelaciones.

1.4.7 DISEÑO DE BASES DE DATOS CON EL MODELO E–R

Cuando se ha obtenido el esquema conceptual de una base de datos, la siguiente fase del proceso de desarrollo de un sistema de bases de datos es la implantación, para lo cual es necesario realizar la transformación del modelo entidad-relación a un modelo soportado por un DBMS, como es el modelo relacional. A continuación se describen algunos lineamientos para la realización de ese mapeo.

ENTIDADES Y ATRIBUTOS

El primer paso es mapear las entidades, para lo cual cada entidad en el modelo E-R se transforma en una relación cuya llave es la definida en el diagrama E-R. Cada atributo definido en el modelo E-R corresponde a un atributo en el relacional.

En segundo lugar se mapean las interrelaciones, incluyendo tanto sus atributos explícitos como los implícitos. Los atributos explícitos son aquellos que se definieron en el modelo E-R, mientras que los implícitos son las llaves principales de las entidades a las que asocia la interrelación. El mapeo en este caso depende de la cardinalidad de la interrelación:

INTERRELACIONES 1:1

En un principio ambas entidades producen relaciones separadas, sin embargo, de acuerdo con el tipo de interrelación de que se trate se pueden seguir dos caminos:

1. Integrar ambas entidades en una sola relación, cuando hay obligatoriedad en cuanto a ambas entidades.
2. Definir una relación separada, cuando al menos una de las entidades no es obligatoria.

INTERRELACIONES 1 : N

Cuando la entidad del lado *1* es obligatoria, se incluye su llave en la relación correspondiente al lado *n*; de lo contrario, se crea una relación para ambas entidades y otra para la interrelación, cuya llave principal es la llave de la entidad del lado *n*, y que tiene una llave foránea que hace referencia a llave de la entidad del lado 1.

INTERRELACIONES M : N

Toda interrelación *m:n* mapea a una relación. La relación debe incluir aquellas llaves foráneas que corresponden a las llaves de sus dos participantes. La llave principal de esa relación se puede obtener de dos maneras:

1. Tomando las llaves foráneas correspondientes a sus participantes.
2. Definiendo un nuevo atributo simple que sea la llave principal.

INTERRELACIONES RECURSIVAS

Una interrelación recursiva R de una entidad E se modela como una nueva relación que incluye dos atributos, ambos corresponden a la llave principal de E y sus nombres corresponden a los papeles que juega E en R. La llave principal se elige de acuerdo al tipo de interrelación que corresponda (1:1, 1:n, m:n).

ENTIDADES DÉBILES

La interrelación de una entidad débil con la entidad que la determina es una interrelación 1 : n.

GENERALIZACIÓN

Hay dos formas de mapear la generalización:

1. Creando una relación para la entidad genérica con sus atributos propios y una para cada entidad especializada que contenga, además de los propios, los atributos que forman parte de la llave de su entidad genérica.

2. Creando una relación para cada entidad especializada que contenga todos los atributos comunes y los especializados y, de ser necesario, crear una vista con la proyección sobre los atributos genéricos de la unión de todas las relaciones especializadas.

1.4.8 UN EJEMPLO

Para ilustrar el diseño de bases de datos relacionales con el modelo entidad–relación se presenta a continuación un ejemplo completo.

Considérese el caso de un videoclub que desea automatizar su operación. El club tiene miembros (clientes) y una colección de cintas disponibles para su renta. El propósito del sistema es registrar membresías, inventario de las cintas y rentas.

Para cada miembro, el club desea llevar un registro de los siguientes elementos de datos:

- Un número único de identificación o membresía.
- Nombre.
- Dirección.
- Fecha en que se unió al club.
- Las cintas que ha rentado.
- Datos de hasta dos co-suscriptores (con credenciales adicionales).

Para cada cinta, los datos necesarios son:

- Número de identificación (en el código de barras) por cada copia de la cinta.
- Título.
- Clasificación.
- Costo de la renta.
- Fecha de adquisición.
- Precio de adquisición.
- Proveedor.
- Dirección del proveedor.

Para cada renta, el club necesita los siguientes datos:

- Fecha.
- Fecha límite de devolución.
- Fecha de devolución.
- Costo de la renta.

El videoclub espera del sistema que pueda proporcionarle información tal como:

- Una lista de nombres de los miembros que se unieron al club en un mes particular.
- Una lista de miembros con sus co-suscriptores.
- Una lista de películas de determinada clasificación.
- Una lista de películas de determinada clasificación que un determinado miembro aún no ha rentado.
- Un reporte del número de copias por cada cinta.
- Un reporte de las cintas rentadas en un momento determinado.
- Un reporte de las cintas rentadas en un momento determinado, por cada miembro, incluyendo sus co-suscriptores.
- Un reporte de cintas no devueltas a tiempo.
- Un reporte de miembros que aún no han rentado cinta alguna.

MODELADO CON DIAGRAMAS E-R

El primer paso del modelado de datos con DER es identificar las entidades. En este ejemplo las entidades identificables a primera instancia son miembros y cintas, que se llamarán MIEMBRO y CINTA.

De acuerdo con los requerimientos del club, la entidad MIEMBRO tiene como atributos su RFC, su nombre, dirección y fecha de suscripción; como llave principal se define el RFC, ya que es un valor único por cada miembro. La entidad CINTA, por su parte, tiene como atributos un número de identificación, título, clasificación, costo de la renta, fecha de adquisición, precio de adquisición, nombre del proveedor y dirección del proveedor. Su llave principal será el número de identificación contenido en el código de barras. Por otra parte, es necesario llevar un registro del nombre de cada co-suscriptor, para lo cual se creará una entidad CO_SUSCRIPTOR, la cual tendrá como atributo el nombre del mismo.

Para llevar el registro de las cintas que el miembro ha rentado se crea una interrelación, que permita relacionar el miembro con las instancias de cinta que haya rentado. Para cada miembro del club habrá varias cintas rentadas y cada cinta puede ser rentada a varios miembros, por lo que la cardinalidad de esa interrelación es M:N. Esa interrelación se denominará RENTADA_POR y tendrá como atributos la fecha de renta, la fecha límite de devolución, la fecha de devolución y el costo de la renta pagado.

Un aspecto importante a considerar cuando se modela la base de datos, es eliminar al máximo la redundancia. En el ejemplo, cada instancia de CINTA representa un ejemplar, por lo que para cada película es posible que haya varias copias en varias cintas. Los datos básicos de la película se tendrían que repetir tantas veces como el número de copias que se tengan, originando un riesgo de que cada vez que se tengan que actualizar esos datos haya que buscar todas las cintas y cambiar los datos en varios lugares. Para eliminar este problema se creará la entidad PELÍCULA con los siguientes atributos: identificador de la película, título, clasificación y costo de la renta. Y para asociar los elementos en PELÍCULA con las cintas se crea la interrelación GRABADA_EN, que significa que cada ocurrencia de PELÍCULA describe a varias ocurrencias de CINTA (1:N).

La entidad CINTA implica aún más redundancia en este momento, ya que cada cinta contiene la información relativa al proveedor y a su adquisición. La solución es crear una entidad que contenga sólo una instancia de la información redundante. La entidad PROVEEDOR tendrá entonces como atributos el RFC del proveedor, su nombre y su dirección. Y para asociar a los proveedores con las cintas se crea una interrelación ADQUIRIDA_DE que tendrá como atributos la fecha y el precio de

adquisición y su cardinalidad será M:1, ya que cada proveedor puede proveer varias cintas, pero una cinta en particular proviene de un solo proveedor.

Otro punto de redundancia es la entidad PELÍCULA. Si el club deseara actualizar las tarifas de renta de acuerdo con algún parámetro, por ejemplo, cobrar más caro por las películas de estreno y más barato por las que no son muy populares, el proceso de detección de estrenos sobre la lista de películas sería costoso. De esta forma es conveniente modificar la entidad PELÍCULA y crear una entidad nueva denominada CATEGORÍA, la cual contendrá un identificador de la categoría (llave principal) y el cargo correspondiente. Así se elimina el atributo costo de la renta de PELÍCULA. La interrelación que asociará a la categoría con la película se denominará CLASIFICADA_EN, con cardinalidad 1:N (para cada CATEGORÍA hay varias instancias en PELÍCULA). En la figura 1.10 se observa el diagrama E-R final, es decir, el esquema conceptual de la base de datos del videoclub.

MAPEO AL MODELO RELACIONAL

Una vez creado el modelo conceptual de la base de datos se procede a diseñar el esquema relacional, es decir, el mapeo al modelo relacional para ser implantado en un ***RDBMS*** (*"Relational DataBase Management System"*).

De la sección anterior se obtiene la siguiente estructura de las entidades e interrelaciones:

- MIEMBRO(RFC DEL MIEMBRO, NOMBRE, DIRECCIÓN, FECHA DE SUSCRIPCIÓN)
- CO_SUSCRIPTOR(NOMBRE DEL COSUSCRIPTOR)
- CINTA(NÚMERO DE IDENTIFICACIÓN)
- PELÍCULA(IDENTIFICADOR DE LA PELÍCULA, TÍTULO, CLASIFICACIÓN, COSTO DE LA RENTA)
- PROVEEDOR(RFC DEL PROVEEDOR, NOMBRE, DIRECCIÓN)
- CATEGORÍA(IDENTIFICADOR DE LA CATEGORÍA, CARGO)
- RENTADA_POR(FECHA DE RENTA, FECHA LÍMITE DEVOLUCIÓN, FECHA DE DEVOLUCIÓN, PAGO)
- GRABADA_EN()
- CLASIFICADA_EN()
- ADQUIRIDA_DE(FECHA DE ADQUISICIÓN, PRECIO DE ADQUISICIÓN)
- REGISTRADO_POR()

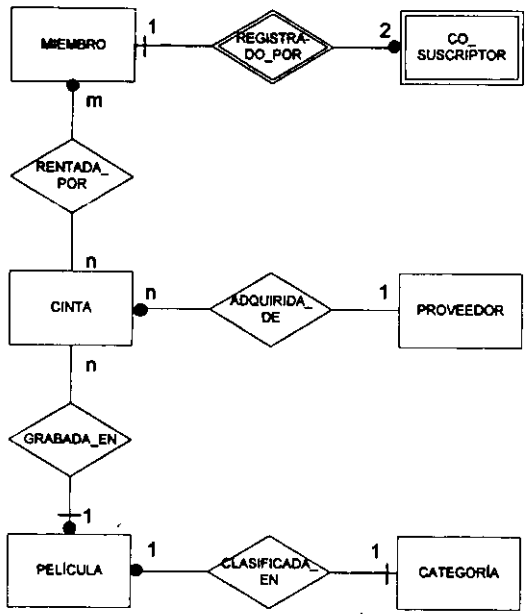

Figura 1.10. Esquema conceptual de la base de datos del videoclub.

El primer paso es representar las entidades como relaciones, subrayando el (o los) atributo(s) de la llave principal:

- MIEMBRO(RFC_MIEMBRO, NOMBRE_MIEMBRO, DIRECCIÓN_MIEMBRO, FECHA_SUSCRIPCIÓN)
- CO_SUSCRIPTOR(RFC_MIEMBRO, NOMBRE_COSUSCRIPTOR)
- CINTA(ID_CINTA)
- PELÍCULA(ID_PELÍCULA TÍTULO, CLASIFICACIÓN)
- PROVEEDOR(RFC_PROVEEDOR, NOMBRE_PROVEEDOR, DIRECCIÓN_PROVEEDOR)
- CATEGORÍA(ID_CATEGORÍA, CARGO)

Nótese que CO_SUSCRIPTOR incluye el atributo RFC_MIEMBRO (que es la llave de MIEMBRO) como parte de su llave, dado que no puede identificarse de manera única, es decir, depende de el miembro que lo ha registrado.

El segundo paso es representar cada interrelación en el DER como una relación:

- RENTADA_POR(RFC_MIEMBRO, ID_CINTA, FECHA_RENTA, FECHA_LÍM_DEVOLUCIÓN, FECHA_DEVOLUCIÓN, PAGO)
- GRABADA_EN(ID_CINTA, ID_PELÍCULA)
- CLASIFICADA_EN(ID_PELÍCULA, ID_CATEGORÍA)
- ADQUIRIDA_DE(ID_CINTA, RFC_PROVEEDOR, FECHA_ADQUISICIÓN, PRECIO_ADQUISICIÓN)
- REGISTRADO_POR(RFC_MIEMBRO, NOMBRE SUSCRIPTOR)

Nótese que para RENTADA_POR se ha incluido FECHA_RENTA como parte de la llave. Esto se debe a que un miembro podría rentar la misma copia de una película más de una vez.

Para determinar la llave principal de una interrelación 1:N se usa la llave principal de la entidad del lado N, de manera que la llave principal de la relación GRABADA_EN es el atributo ID_CINTA de la entidad CINTA. La interrelación ADQUIRIDA_DE se ha mapeado de la misma forma, incluyendo el atributo RFC_PROVEEDOR, ya que es la llave de PROVEEDOR.

Es posible simplificar el esquema buscando combinar relaciones cuando existen relaciones derivadas de interrelaciones en el DER que tengan la misma llave principal que aquellas derivadas de entidades. Por ejemplo:

- La relación ADQUIRIDA_DE tiene la misma llave principal que CINTA. Se pueden combinar estar moviendo RFC_PROVEEDOR, PRECIO_ADQUISICIÓN y FECHA_ADQUISICIÓN desde ADQUIRIDA_DE hacia CINTA, eliminando así la relación ADQUIRIDA_DE.

- La relación GRABADA_EN tienen la misma llave principal que CINTA, por lo tanto se pueden mover sus atributos que no forman parte de la llave principal a la relación CINTA, eliminando así GRABADA_EN.

- La relación CLASIFICADA_EN también tiene la misma llave principal que PELÍCULA. Moviendo el atributo que no es parte de la llave a PELÍCULA se elimina la relación CLASIFICADA_EN.

El esquema relacional final de la base de datos del videoclub es el siguiente:

- MIEMBRO(RFC_MIEMBRO, NOMBRE_MIEMBRO, DIRECCIÓN_MIEMBRO, FECHA_SUSCRIPCIÓN)
- CO_SUSCRIPTOR(RFC_MIEMBRO, NOMBRE_COSUSCRIPTOR)

- CINTA(ID_CINTA, RFC_PROVEEDOR, FECHA_ADQUISICIÓN, PRECIO_ADQUISICIÓN, ID_PELÍCULA)
- PROVEEDOR(RFC_PROVEEDOR, NOMBRE_PROVEEDOR, DIRECCIÓN_PROVEEDOR)
- PELÍCULA(ID_PELÍCULA, TÍTULO, CLASIFICACIÓN, ID_CATEGORÍA)
- CATEGORÍA(ID_CATEGORÍA, CARGO)
- RENTADA_POR(RFC_MIEMBRO, ID_CINTA, FECHA_RENTA, FECHA_LÍM_DEVOLUCIÓN, FECHA_DEVOLUCIÓN, PAGO)
- REGISTRADO_POR(RFC_MIEMBRO, NOMBRE SUSCRIPTOR)

Este esquema relacional puede implantarse en cualquier DBMS, siguiendo sus procedimientos y reglas particulares.

CAPÍTULO 2

ANÁLISIS Y DISEÑO ORIENTADO A OBJETOS

2.1. EL MODELO ORIENTADO A OBJETOS

El modelo orientado a objetos se basa en la visión del mundo real como un conjunto de objetos que colaboran para lograr algún comportamiento. La orientación a objetos es un paradigma aplicable en las diferentes etapas del proceso de desarrollo del software (e incluso en áreas distintas a la ingeniería de software).

Las principales características de este modelo son:

- Facilita el manejo de la complejidad de los problemas a resolver.
- Alienta la reutilización.
- Permite el uso de patrones.
- Apoya la construcción de sistemas flexibles.
- Reduce los riesgos de desarrollo.

2.1.1. ELEMENTOS DE MODELO

El modelo de objetos tiene como base conceptual cuatro elementos fundamentales [BOOCH]:

- ***Abstracción***. Resalta las características esenciales de un objeto que lo distinguen de todos los demás tipos de objetos y proporciona así fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador.

- ***Encapsulamiento***. Es el proceso de ocultar los detalles de un objeto que no contribuyen a sus características esenciales.

- ***Modularidad***. Es la propiedad que tiene un sistema que ha sido descompuesto en un conjunto de módulos cohesivos (que agrupen abstracciones con cierta relación lógica) y débilmente acoplados (minimizando las dependencias entre ellos).

- ***Jerarquía***. Es una clasificación u ordenación de abstracciones.

Además de los elementos fundamentales, existen otros elementos conceptuales (llamados secundarios) que se utilizan en este modelo:

- ***Tipificación***. Las clases representan tipos de datos, de manera que los objetos de distintos tipos no pueden intercambiarse o pueden intercambiarse sólo en ciertas formas restringidas [BOOCH].

- ***Concurrencia***. Es la propiedad que distingue un objeto activo de otro que no lo es.

- ***Persistencia***. Es la propiedad de un objeto por la que su existencia trasciende el tiempo (es decir, el objeto sigue existiendo después de que quien lo creó deja de existir) y/o el espacio (la posición del objeto cambia respecto a la dirección en que fue creado) [BOOCH].

2.1.2. OBJETOS

Un ***objeto*** es una cosa tangible y/o visible, algo sobre lo cual se realiza una acción o algo que puede ser comprendido intelectualmente. Todo objeto está formado por tres elementos [BOOCH]:

- Una ***identidad***, es decir, la propiedad de un objeto de ser distinguido de los demás.

- Un ***estado***, formado por un conjunto de propiedades y sus valores en un momento determinado.

- Un ***comportamiento***, que es la forma en que reacciona el objeto al interactuar con otros objetos y su cambio de estado.

La interacción que define el comportamiento de los objetos se realiza a través de ***mensajes***, los cuales son llamadas a las operaciones o métodos del objeto. Existen varios tipos de métodos:

- ***Constructor***. Crea un objeto y/o inicializa su estado.

- ***Destructor***. Libera el estado del objeto y/o lo destruye.

- ***Modificador***. Altera el estado del objeto.

- ***Selector***. Accede al estado del objeto, pero no lo altera.

- ***Iterador***. Permite acceder a los componentes del objeto en un orden predeterminado.

2.1.3. RELACIONES ENTRE OBJETOS

De acuerdo a las relaciones que establece con los demás, un objeto puede desempeñar uno de los siguientes papeles:

- ***Actor***. Cuando puede operar sobre otros objetos pero ningún objeto opera sobre él.
- ***Servidor***. Cuando nunca opera sobre otros objetos, sólo otros objetos operan sobre él.
- ***Agente***. Cuando puede operar sobre otros objetos y además otros operan sobre él [BOOCH].

ENLACES

Un ***enlace*** es una conexión física o conceptual entre objetos a través de la cual un objeto cliente utiliza los servicios de otro objeto, el servidor. Un enlace establece una vía para el paso de mensajes (en una sola dirección o en ambas) entre los objetos asociados.

AGREGACIÓN

Un **objeto agregado** es aquel que contiene a otro(s). La contención puede ser por valor o por referencia.

2.1.4. CLASE

Una ***clase*** es un conjunto de objetos que comparten una estructura y un comportamiento común [BOOCH]. Un objeto es una instancia de una clase. Una clase es un concepto abstracto que representa la esencia de los objetos que la componen. Se pueden distinguir dos partes de una clase:

- ***Interfaz***. Es la visión externa que los objetos de una clase tienen de los objetos de otra clase, que enfatiza la abstracción ocultando su estructura y detalles de implantación. Existen cuatro niveles de visibilidad para los elementos de la interfaz:
  - *Pública*. Accesible a todos los clientes.

- *Protegida.* Accesible sólo a los clientes que pertenecen a la misma clase y a sus subclases.
- *Privada.* Accesible sólo a los objetos de la misma clase.
- *"Package".* En Java es la visibilidad por omisión e indica que es accesible sólo a los clientes de las clase definidas dentro del mismo paquete.

- ***Implantación.*** Es la visión interna de la clase, que engloba su comportamiento y está formada por la implantación de los métodos de la clase.

2.1.5. RELACIONES ENTRE CLASES

ASOCIACIÓN SIMPLE

Una relación de asociación describe alguna dependencia semántica entre clases. Una asociación de este tipo se caracteriza mediante su ***cardinalidad***, es decir, el número de objetos de una clase que pueden asociarse con los de la otra clase, y los ***papeles*** o ***"roles"*** que juega cada una de las clases que participan en la asociación.

AGREGACIÓN

Indica una relación de contención entre clases, en que la clase que contiene se denomina clase ***agregada*** y la clase contenida se conoce como ***componente***.

DEPENDENCIA

Indica una relación semántica entre clases, en la que un cambio en una clase puede requerir un cambio en la otra clase.

HERENCIA

La ***herencia*** es una relación en la que una clase llamada ***subclase*** comparte la estructura y/o el comportamiento definidos en otra clase llamada ***superclase***. Dado que cualquier clase puede a su vez heredar de otra, la herencia define una estructura llamada ***jerarquía de herencia***.

Un concepto que fortalece el uso de la herencia es el ***polimorfismo***, que se puede entender en diversos sentidos, entre los cuales se encuentran los siguientes:

- En teoría de tipos, es la habilidad de una variable de tomar diferentes tipos, dependiendo de su contenido en un momento determinado.

- Es la posibilidad de utilizar un objeto de una subclase como si fuera un objeto de su superclase; los métodos se evalúan con base en el objeto concreto (en tiempo de ejecución), es decir, si un método está definido en la superclase y en la subclase también, se ejecutará aquel método definido en la subclase.

2.2. EL LENGUAJE DE MODELADO UNIFICADO (UML)

El ***Lenguaje de Modelado Unificado*** o ***UML*** (*"Unified Modeling Language"*) es un lenguaje para especificar, visualizar, construir y documentar los componentes de los sistemas de software, al igual que para elaborar modelos de negocios y de otros sistemas no computacionales [OMG].

Ningún sistema puede describirse desde un sólo punto de vista, ya que deben capturarse los aspectos necesarios de acuerdo con el tipo de problema en cuestión y desde diferentes niveles de abstracción. Como se observa en la siguiente tabla, UML proporciona diferentes herramientas gráficas que plasman diferentes aspectos de un modelo de objetos.

DIAGRAMA	ASPECTO QUE REPRESENTA
Casos de uso	Interacción del sistema modelado con usuarios y otros sistemas.
Clases	Estructura estática del sistema.
Paquetes	Agrupa las clases del sistema, de acuerdo con sus propiedades en común.
Comportamiento	Propiedades dinámicas del sistema; existen dos tipos: de estado y de actividad.
Interacción	Intercambio de mensajes entre objetos; hay dos tipos: de secuencia y de colaboración.
Implantación	Disposición física de los módulos o componentes (de software y hardware) de un sistema; pueden ser de instalación y de componentes.

En esta tesis se utilizaron los diagramas de casos de uso, de paquetes, de clases y de secuencia, mismos que se describen a continuación.

2.2.1. DIAGRAMAS DE CASOS DE USO

Los diagramas de casos de uso permiten visualizar la interacción entre el sistema a modelar y los usuarios u otros sistemas. Sus elementos principales son: casos de uso, actores y asociaciones.

Un ***caso de uso*** es una interacción típica entre un sistema y sus usuarios o aquellos sistemas externos con los que se relaciona. Se representa gráficamente como una elipse etiquetada con su nombre, como se observa en la figura 2.1.

Figura 2.1. Icono de un caso de uso.

Se conoce como ***actor*** a un papel que el usuario o un sistema externo desempeña respecto al sistema, es quien raaliza el caso de uso. Gráficamente se representa como lo muestra la figura 2.2.

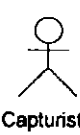

Figura 2.2. Representación gráfica de un actor.

Las asociaciones vinculan a los actores con los casos de uso y a los casos de uso entre sí. En el primer caso un actor se asocia con un caso de uso mediante una asociación unidireccional, como se observa en la figura 2.3.

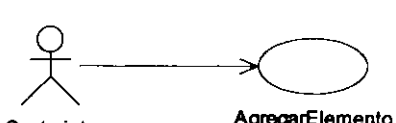

Figura 2.3. Asociación actor-caso de uso.

Las asociaciones entre casos de uso pueden ser de dos tipos:

- ***Uso*** *("uses")*. Cuando algún comportamiento se utiliza en varios casos de uso se crea un caso de uso particular, que es utilizado por los demás. Entre aquellos y el nuevo caso de uso se establece una relación de uso.

- ***Extensión*** *("extends")*. Se dice que un caso de uso extiende a otro cuando es similar a él, pero realiza algún comportamiento adicional.

En la figura 2.4 se muestra un diagrama de casos de uso a manera de ejemplo. El caso de uso ELIMINARMIEMBRO "usa" a "ELIMINACOSUSCRIPTOR" ya que para un miembro hay hasta dos co-suscriptores que deben eliminarse previamente. Antes de eliminar a un miembro o a un co-suscriptor, se deben verificar las rentas de películas que aún no han sido devueltas, así que el caso de uso que verifica las rentas del co-suscriptor es una versión ampliada de aquél que verifica las rentas del miembro, esto es, lo "extiende".

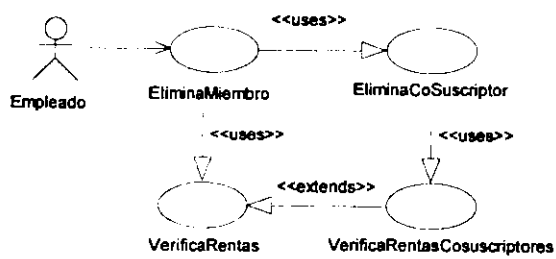

Figura 2.4. Un diagrama de casos de uso.

Los diagramas de casos de uso se utilizan ampliamente durante el análisis de requerimientos, pues generalmente a cada caso de uso corresponde un requerimiento potencial del usuario, pero también se aplican durante la etapa de diseño.

2.2.2. DIAGRAMAS DE PAQUETES

Considerados en ocasiones como parte del diagrama de clases, los paquetes son instrumentos para la organización de las clases de un sistema. Un ***paquete*** es una unidad lógica de alto nivel que agrupa elementos de un modelo con características similares. Además de clases, interfaces, asociaciones, etc., los paquetes pueden contener otros paquetes. Pueden representar subsistemas o módulos. La descripción del sistema puede verse como un paquete único que contiene a todos los elementos de modelado y diagramas definidos en UML.

Puede existir una asociación de dependencia entre paquetes, como la que se muestra en el diagrama de la figura 2.5. El paquete *seerd*, que contiene las clases propias de la herramienta objeto de este trabajo de tesis depende del paquete *sebasic*, que contiene clases básicas como secuencias, iteradores, etc. y de *segui*, que contiene las clases propias de la interfaz gráfica de usuario. Esto significa que los cambios en algunas clases de los paquetes *sebasic* y/o *segui* podrían implicar la necesidad de cambios en *seerd*.

Figura 2.5. Un diagrama de paquetes.

2.2.3. DIAGRAMAS DE CLASES

Un diagrama de clases describe la estructura estática del modelo, es decir, las clases que aparecen en un sistema, permitiendo la definición de sus atributos y operaciones, así como las relaciones entre ellas.

Las ***clases*** se representan como un rectángulo etiquetado con el nombre de la clase, o bien con 3 compartimentos separados por líneas horizontales (figura 2.6). El primer compartimiento es el

destinado al nombre y puede incluir algunas otras propiedades de la clase (como el estereotipo[2]); el segundo compartimiento contiene la lista de atributos de la clase y el último, una lista de operaciones (métodos).

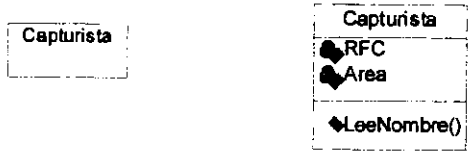

Figura 2.6. Iconos de clase.

Una interfaz es un conjunto de operaciones sin la especificación de su estructura interna. Una interfaz no tiene implantación, atributos, estados ni asociaciones, sólo operaciones, sin embargo sí puede tener relaciones de generalización. En la figura 2.7-a se muestra una interfaz representada con el icono propio de una clase, con el estereotipo <<INTERFACE>> en la parte superior. En la figura 2.7-b se muestra una clase (SEERCREATEENTITY) que implanta a la interfaz SECOMMAND, es decir, que se comporta como un objeto SECOMMAND.

Figura 2.7. Icono de interfaz.

Las ***asociaciones*** representan interrelaciones entre instancias de clases. Como se observa en la figura 2.8, una asociación tiene dos roles, cada rol es una dirección de la asociación y corresponde a una de las clases asociadas. Además tienen cardinalidad, que indica cuantos objetos pueden

[2] El estereotipo indica que el elemento es una subclase de un elemento existente con la misma forma (atributos e interrelaciones) pero con una intención diferente, generalmente representa una diferencia de uso [OMG].

participar en una relación dada; la cardinalidad en UML se representa como lo muestra la tabla de la figura 2.9.

Figura 2.8. Representación de asociación entre clases.

Otra característica de las asociaciones es que pueden indicar ***navegación***, representada en el diagrama como una flecha. La navegación significa que el objeto de la clase origen (de la que parte la flecha) tiene acceso al objeto de la clase destino (a donde apunta la flecha), pero no al revés.

CARDINALIDAD	REPRESENTA...
*	Un rango de 0 a n
1	Uno y sólo uno
1:1	Uno y sólo uno
1..*	Al menos uno y hasta n
0..1	Ninguno o uno

Figura 2.9. Cardinalidad de las asociaciones.

La ***agregación*** se representa mediante un rombo en el extremo de la clase agregada que cuando es vacío indica la agregación por referencia y cuando es relleno, la agregación por valor (figura 2.10).

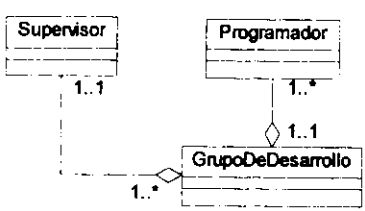

Figura 2.10. Agregación.

La ***generalización-especialización*** es otro tipo de interrelación entre clases y su concepto es equivalente a la herencia. La figura 2.11 muestra la representación de esta relación.

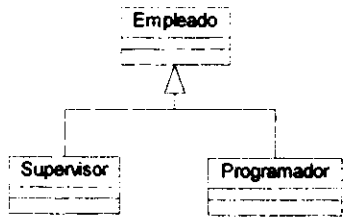

Figura 3.11. Generalización-especialización.

Se pueden insertar comentarios o bloques de texto para describir los elementos del modelo, mediante una nota o restricción asociada al elemento correspondiente, como lo muestra la figura 2.12.

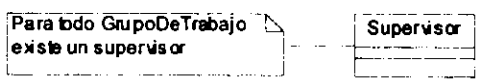

Figura 2.12. Comentarios.

2.2.4. DIAGRAMAS DE SECUENCIA

Los diagramas de interacción describen la colaboración entre grupos de objetos en cierto comportamiento del sistema; generalmente capturan el comportamiento de un caso de uso, mostrando un grupo de objetos ejemplo y los mensajes que pasan entre ellos dentro del mismo [FOWLER].

Un ***diagrama de secuencia*** es un tipo de diagrama de interacción que representa un conjunto de mensajes ordenados intercambiados entre objetos para lograr un comportamiento determinado.

En los diagramas de secuencia un objeto se representa como una caja colocada sobre una línea vertical punteada llamada línea de vida del objeto, la cual representa la vida del objeto durante la interacción. En la caja se escribe el nombre del objeto y de su clase correspondiente, separados por ':'.

Cada mensaje se representa como una flecha que va de la línea de vida de un objeto a la de otro; el orden de ocurrencia de los mensajes se muestra de arriba hacia abajo de la página. Un mensaje puede etiquetarse con su nombre, sus argumentos y alguna información de control.

Es posible representar la autodelegación (es decir, el envío de mensajes de una clase a sí misma) mediante una flecha que retorna a la misma línea de vida del objeto que la lanzó. En la figura 2.13 se muestra la forma básica de un diagrama de secuencia.

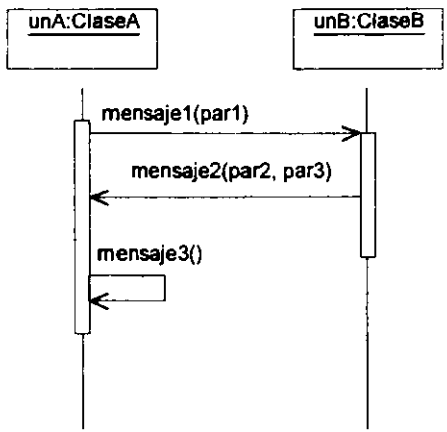

Figura 2.13. Diagrama de secuencia.

2.3. PATRONES DE DISEÑO

Para que el diseño de software orientado a objetos sea reutilizable, se deben cumplir las siguientes condiciones:

- Contemplar los cambios y problemas futuros.
- Evitar al máximo la necesidad de rediseñar.
- No intentar resolver los problemas partiendo de cero, sino utilizar soluciones cuyo funcionamiento ha sido probado anteriormente y que pueden funcionar en el problema actual.

Un patrón es una plantilla que ha sido útil en algún contexto práctico y que puede ser útil en otros contextos. Un ***patrón de diseño*** es una plantilla de objetos con responsabilidades e interacciones estereotípicas , es decir, una plantilla de objetos en interacción que puede utilizarse en más de un caso por analogía [GAMMA].

El patrón ***iterador*** es un patrón de diseño ampliamente utilizado que permite el acceso a los elementos de un objeto agregado sin exponer su representación interna [GAMMA].

El patrón iterador permite tener la representación adecuada del agregado según las necesidades de implantación (ya sea como un arreglo, lista ligada, lista doblemente ligada, etc.), utilizando los mismos métodos de acceso. El iterador toma la responsabilidad para el acceso y representación de la lista (definiendo una interfaz para acceder a sus elementos) y para conocer en todo momento cuál es el elemento actual. Al separar el mecanismo de recorrido permitimos definir un iterador para cada tipo de representación.

En el ejemplo de la figura 2.14 se muestra una ***secuencia*** de atributos (SEERATTRIBUTESEQUENCE), asociada a su iterador correspondiente (SEERATTRIBUTESEQITERATOR).

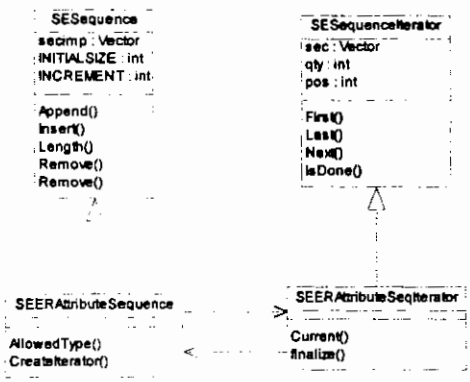

Figura 2.14. Ejemplo del patrón *Iterador*.

CAPÍTULO 3

DESCRIPCIÓN DE LA APLICACIÓN

3.1. FUNCIONALIDAD DE LA HERRAMIENTA

La herramienta desarrollada proporciona un medio sencillo y completo para realizar modelos semánticos de bases de datos, utilizando la simbología de Chen descrita en el capítulo 1.

La funcionalidad general parte de la utilización de un ambiente de ventanas para la creación de diagramas basado en el uso de eventos del mouse sobre botones, menús y un área de dibujo. Estos elementos se describen a detalle en la siguiente sección.

3.2. INTERFAZ CON EL USUARIO

Los elementos gráficos de interfaz con el usuario de esta aplicación se muestran en la figura 3.1, donde se observa la ventana principal de la herramienta.

La ventana principal del sistema consta de cuatro elementos: un menú, barras de herramientas, área de trabajo y barra de mensajes.

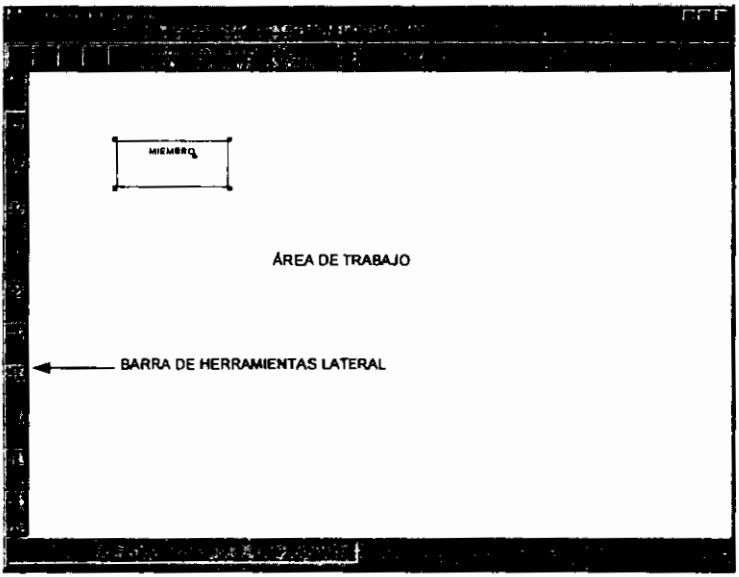

Figura 3.1. Ventana principal de la aplicación.

3.2.1. ELEMENTOS DEL MENÚ

Menú "*File*"

Contiene las operaciones básicas a realizar con los archivos que contienen los diagramas.

- "*New*", crea un nuevo diagrama.
- "*Open*", abre un diagrama existente.
- "*Save*", guarda en disco el diagrama actual.
- "*Save As*", guarda en disco el diagrama actual con el nombre especificado.
- "*Print*", imprime el diagrama actual.
- "*eXit*", sale del sistema.

Menú "*Edit*"

Contiene las operaciones de edición sobre el modelo.

- "*Undo*", deshace las acciones almacenadas en la historia de comandos.
- "*Redo*", vuelve a ejecutar las acciones de la historia de comandos.

Menú "Help"

Presenta la ayuda del sistema.

- "*Help...*". El usuario visualiza únicamente los nombres de las entidades e interrelaciones.
- "*About...*". El usuario puede ver los detalles del modelo, en un nivel conceptual.

3.2.2. ÁREA DE TRABAJO

El área de trabajo es la parte de la herramienta en la que se despliega el DER. Cuando el usuario desea crear un elemento, hará clic sobre el botón del elemento correspondiente y posteriormente sobre esta área en la posición exacta donde quiera colocar el elemento.

3.2.3. BARRAS DE HERRAMIENTAS

La ventana principal de la aplicación contiene dos barras de herramientas: la superior y la izquierda.

La barra superior incluye botones para la ejecución de las opciones del menú, tales como abrir, guardar, salir, deshacer, rehacer, etc.

Abrir

Se utiliza para abrir un diagrama creado con anterioridad.

Guardar

Almacena el diagrama actual en disco.

Salir

Cierra la aplicación.

Deshacer

Deshace los cambios realizados anteriormente, uno a uno. Acciones como guardar y salir no se pueden deshacer.

Rehacer

Rehace los cambios deshechos anteriormente, uno a uno.

La barra izquierda contiene una serie se botones que permiten crear los elementos gráficos de un DER. Para crear cada elemento, el usuario deberá hacer clic en el botón y posteriormente crear el elemento en el área de trabajo.

Crear entidad

Esta herramienta permite agregar una entidad nueva al diagrama. La entidad se creará en el punto del área de trabajo donde el usuario haga clic. Una vez creado el icono de una entidad, el usuario puede definir el nombre de la misma. La figura 3.2. muestra la creación de una entidad.

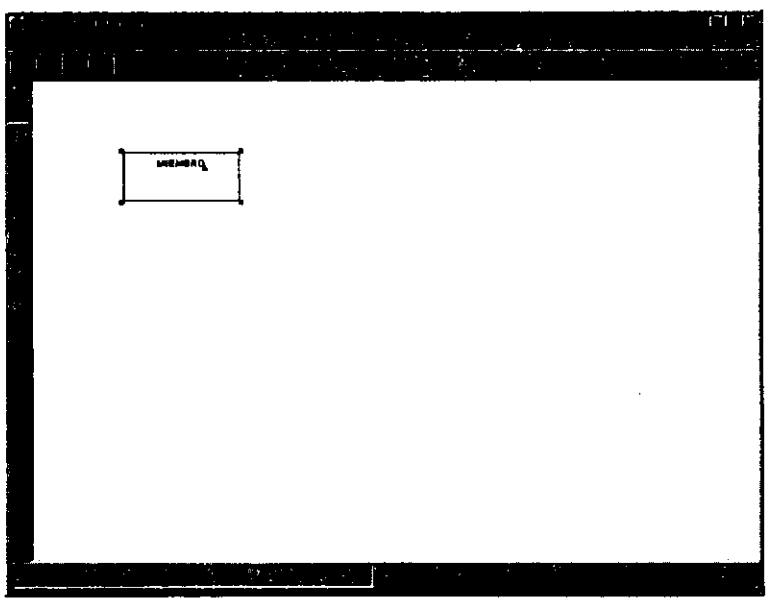

Figura 3.2. Creación de una entidad.

Crear interrelación

Para asociar dos o más entidades el usuario podrá crear una interrelación. Una vez creada, podrá asociarla con cualesquiera entidades en el diagrama mediante el uso de alguno de los botones de creación de asociaciones.

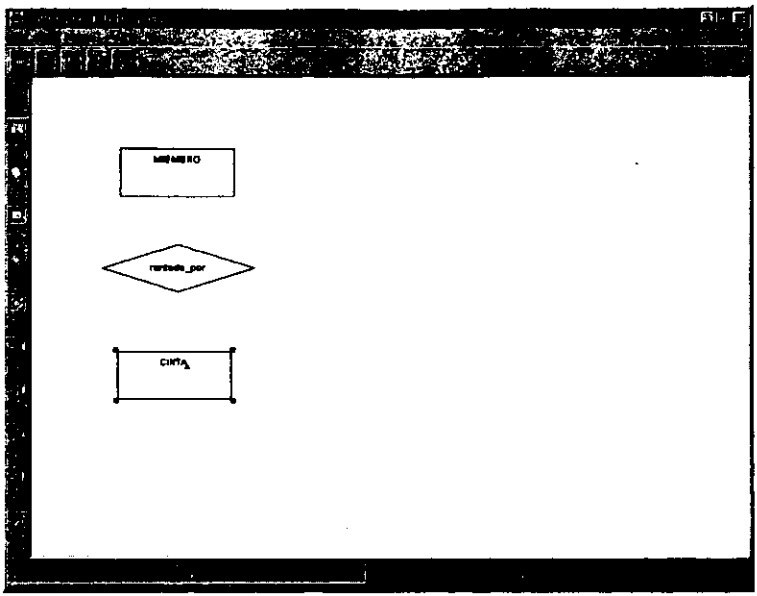

Figura 3.3. Creación de una interrelación.

Crear entidad débil

La creación de una entidad débil es semejante a la de una entidad normal. Este tipo de entidad sólo podrá asociarse con otras mediante una interrelación débil. La figura 3.4. muestra la creación de una entidad débil.

Crear entidad agregada

Al igual que una entidad normal, una entidad agregada se puede relacionar con otras entidades mediante una interrelación normal. La figura 3.5 muestra la creación de una entidad agregada.

Crear interrelación débil

Una interrelación débil es aquella en que al menos una de las entidades que asocia es débil. Si el usuario trata de asociar con una interrelación débil a varias entidades normales sin que exista una débil, el sistema marcará un error. La figura 3.6 muestra la creación de una interrelación débil.

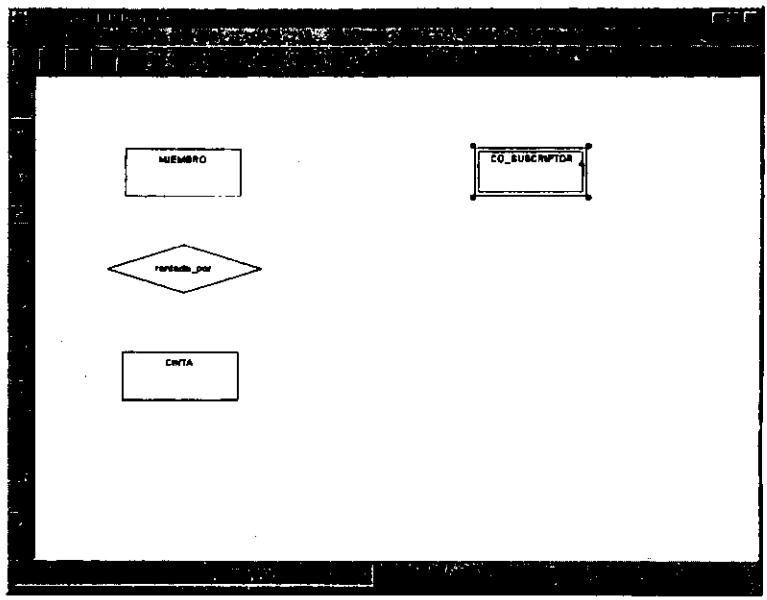

Figura 3.4. Creación de una entidad débil.

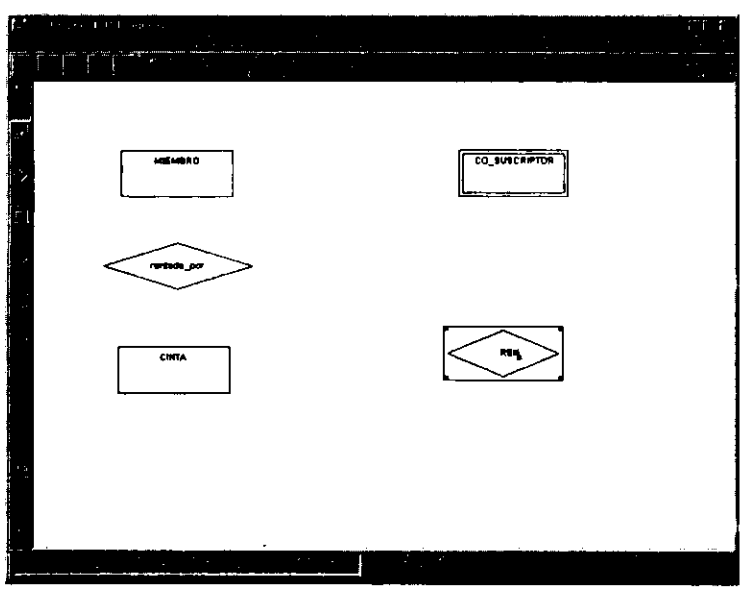

Figura 3.5. Creación de una entidad agregada.

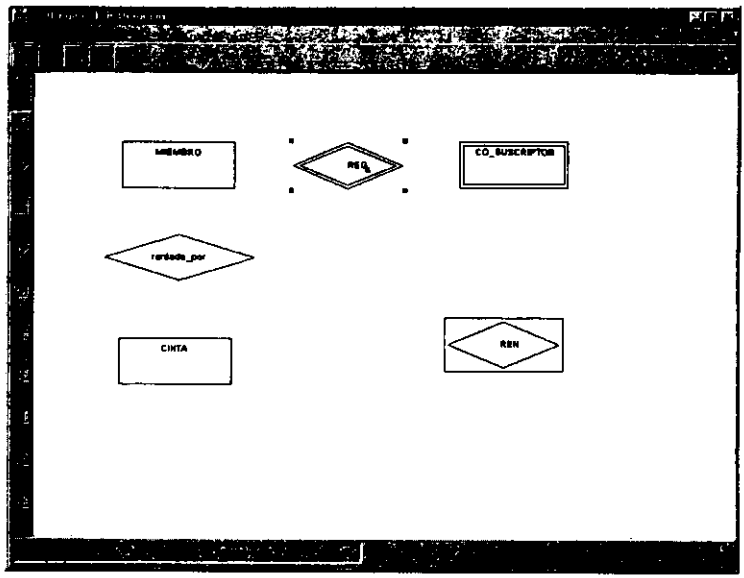

Figura 3.6. Creación de una interrelación débil.

Crear asociación

Este botón permite crear una asociación entre una entidad y una interrelación. El usuario deberá iniciar la creación haciendo clic en el botón, después deberá hacer otro clic en una entidad o interrelación origen y terminar la creación haciendo clic sobre la entidad o interrelación destino (ver figura 3.7).

Crear asociación con lectura

El usuario podrá utilizar este botón para establecer el orden de lectura de la asociación que desea crear. La lectura iniciará en la entidad (normal, débil o agregada) involucrada en esta asociación.

Crear asociación con obligatoriedad

Este botón se utilizará cuando el usuario desee establecer la obligatoriedad de un elemento en una interrelación. La entidad (normal, débil o agregada) involucrada en esta asociación será obligatoria.

Crear asociación con lectura y obligatoriedad

Este botón se utilizará cuando el usuario desee establecer la obligatoriedad de un elemento en una interrelación y el inicio de la lectura en ese elemento.

Crear asociación de generalización-especialización.

Este botón se utilizará cuando el usuario desee establecer una asociación de generalización-especialización entre dos entidades. La asociación deberá partir de la entidad especializada y concluir en la generalizada.

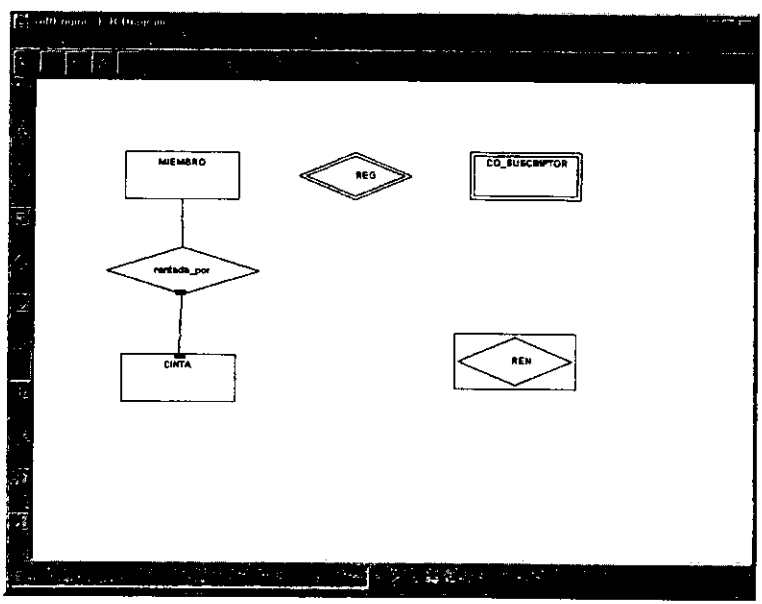

Figura 3.7. Creación de asociaciones.

3.2.4. RESTRICCIONES

Para garantizar que un DER sea semánticamente correcto, esta herramienta define ciertas restricciones en la creación y asociación de elementos. En la siguiente tabla se observan los elementos que legalmente pueden participar en una asociación, marcando la casilla con ✓.

	ENTIDAD	INTERRELACIÓN	ENTIDAD DÉBIL	INTERRELACIÓN DÉBIL	ENTIDAD AGREGADA
ENTIDAD		✓		✓	
INTERRELACIÓN	✓		✓		✓
ENTIDAD DÉBIL		✓		✓	
INTERRELACIÓN DÉBIL	✓		✓		✓
ENTIDAD AGREGADA		✓		✓	

CAPÍTULO 4

ESTRUCTURAS DE IMPLANTACIÓN

4.1. ESTRUCTURA GENERAL

La estructura general de la herramienta es modular, es decir, ésta constituye un módulo susceptible de integrarse con otros mediante una interfaz bien definida. Este módulo está representado mediante el paquete `seerd`. Como lo muestra la figura 4.1, las clases del paquete `seerd` utilizan clases de otros paquetes para su funcionamiento, como las clases de botones contenidas en el paquete `sebuttons`, las clases para la representación gráfica de los objetos de un DER que se encuentran en el paquete `segui`, las clases que corresponden a los comandos que se encuentran en `secommands` y otras clases básicas como el manejo de secuencias y eventos que forman parte del paquete `sebasic`.

A su vez, el paquete `seerd` está estructurado en tres paquetes: en el paquete `seerdapp` se ha colocado la clase con el programa principal de la herramienta, el cual se encarga de su inicialización; el paquete `seerdgui` contiene las clases relacionadas con la GUI, es decir, ventanas, botones, cajas de diálogo, eventos, comandos y la representación gráfica de los elementos de un DER; finalmente, el paquete `seerdme` contiene los elementos conceptuales de un DER o elementos del modelo, es decir, sus componentes a nivel lógico como entidad, interrelación, entidad agregada, etc. (ver figura 4.2).

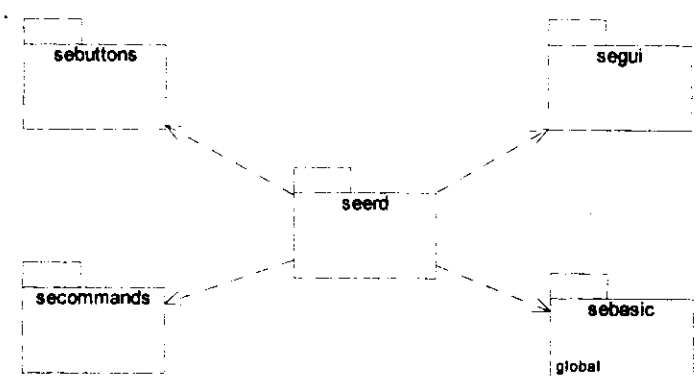

Figura 4.1. Diagrama de paquetes inicial.

Se ha hecho una separación clara entre los objetos de representación gráfica y los elementos del modelo por dos razones: 1) por un principio de diseño orientado a objetos que indica la separación de responsabilidades entre los objetos de un sistema y 2) para controlar el manejo de representaciones múltiples de un mismo elemento del modelo (si un usuario crea dos vistas de una misma entidad y modifica una, debe actualizarse automáticamente la otra, lo cual se controla en el objeto del modelo).

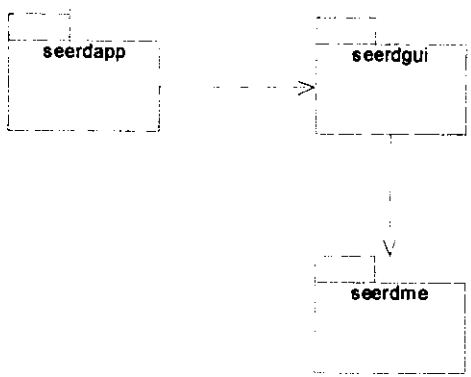

Figura 4.2. Diagrama de paquetes correspondiente al paquete `seerd`.

4.2. ESTRUCTURAS ESTÁTICAS (DIAGRAMAS DE CLASES)

SEERDAPP

El paquete `seerdapp` contiene solamente una clase, `seerdapp`, con el método `main`, es decir, el programa principal. La figura 4.3 muestra el diagrama de clases correspondiente a ese paquete.

Figura 4.3. Diagrama de clases del paquete seerdapp.

SEERDME

En el nivel conceptual se encuentran los elementos del modelo, es decir, los elementos de un DER a nivel lógico. Como se observa en la figura 4.4, todos los elementos del modelo heredan de la clase SEERELEMENT (que hereda de SEDOCUMENTITEM). Los elementos fundamentales de un DER son las entidades y las interrelaciones, representadas con las clases SEERENTITY y SEERRELATIONSHIP. La clase SEERENTITYITEM es una generalización de los elementos de entidad, esto es, las entidades normales (SEEREntity), las entidades agregadas (SEERAGGREGATE) y las entidades débiles (SEERWEAKENTITY). Así mismo, las interrelaciones débiles y la generalización son especializaciones de las interrelaciones normales y se representan por medio de las clases SEERWEAKRELATIONSHIP y SEERGENERALIZATION.

SEERDGUI

Como se mencionó anteriormente, el paquete `seerdgui` contiene las clases de la interfaz gráfica de usuario, que incluye los elementos a dibujar en el área de trabajo y los componentes de la ventana.

La ventana principal está formada por un área de comandos (SECOMMANDAREA), un menú principal (SEMAINMENU), un área de trabajo (SEWORKAREA) y un área de mensajes (SEMESSAGEAREA). También contiene una vista del diagrama (SEERDIAGRAMVIEW), es decir, el objeto que contendrá a todos los elementos que el usuario creará como parte del DER. Esa clase hereda de SEGRAPHICVIEW, la cual maneja el área de trabajo donde habrá de dibujarse el diagrama y controla del elemento que se encuentra seleccionado (figura 4.5).

Durante el proceso de modelado de datos puede resultar conveniente que el analista tenga la posibilidad de visualizar su modelo desde varios puntos de vista. Esto significa que todos los elementos de un DER deberán tener más de una representación gráfica, dependiendo del punto de vista elegido por el usuario, en tiempo de ejecución. Para implantar esta funcionalidad, se aplica un patrón de diseño denominado "fábrica". El objetivo de una fábrica es que, sea cual sea la vista seleccionada, cuando el usuario ejecute la operación de creación de cualquiera de los elementos el procedimiento sea similar y sea la fábrica concreta la que se encargue de crear la representación correspondiente.

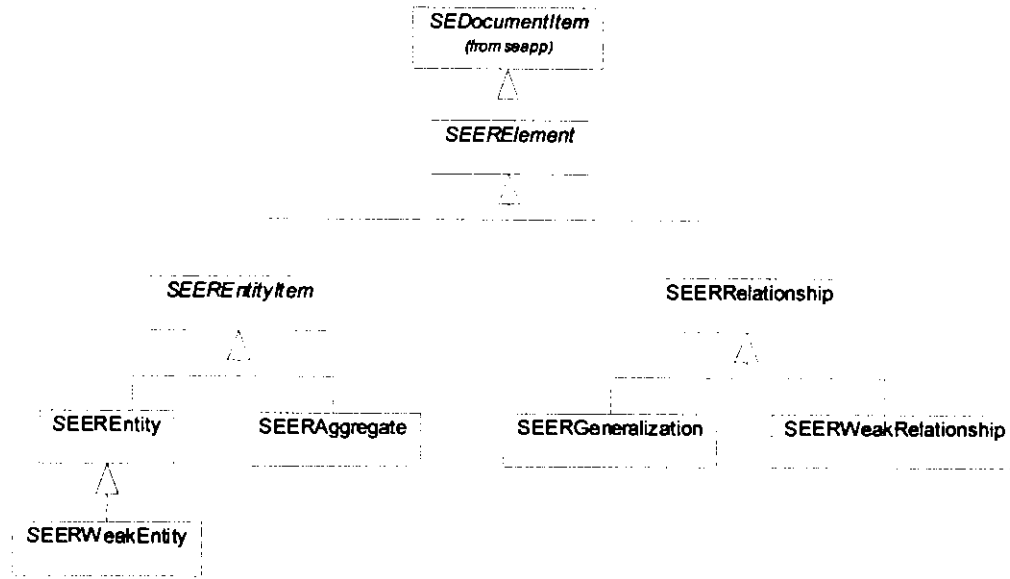

Figura 4.4. Diagrama de clases del paquete seerdme.

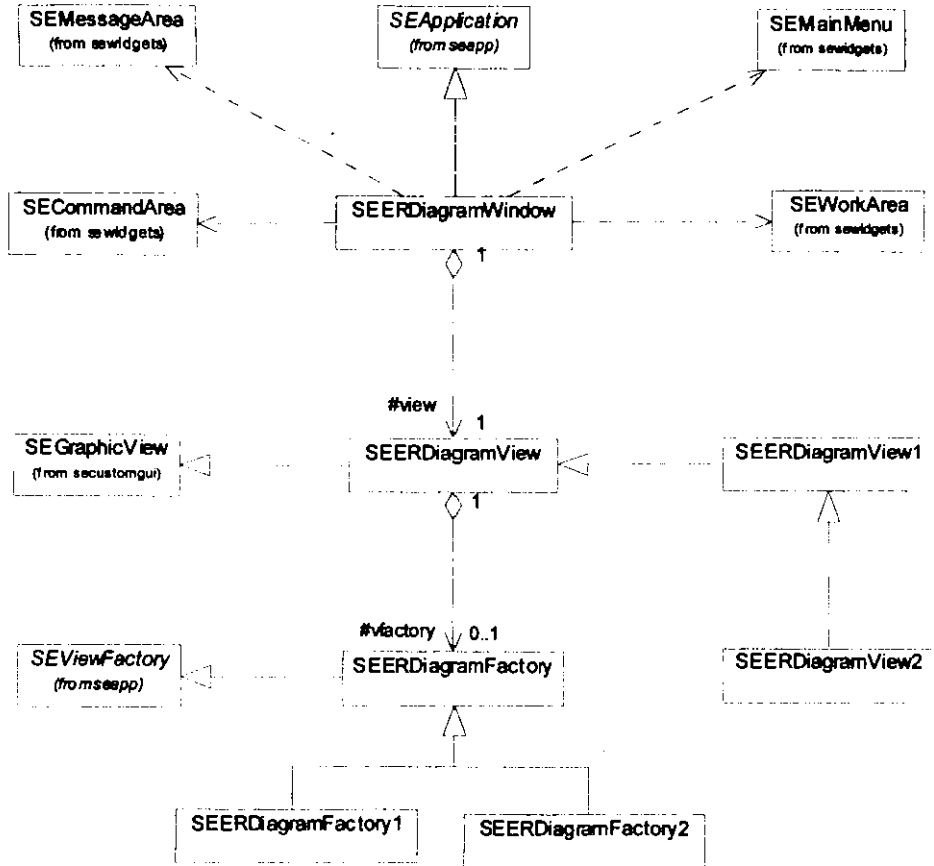

Figura 4.5. Diagrama de clases que conforman un DER.

Las clases SEERDIAGRAMFACTORY, SEERDIAGRAMFACTORY1 y SEERDIAGRAMFACTORY2 crean elementos con diferentes tipos de representación cada una, de acuerdo con la vista elegida por el usuario, que puede ser SEERDIAGRAMVIEW, SEERDIAGRAMVIEW1 o SEERDIAGRAMVIEW2.

Como se puede apreciar en la figura 4.6, las vistas de todos los elementos de un DER heredan de la clase SEERELEMENTVIEW y contienen un nombre, representado por la clase SEERNAMEAREA. Los elementos fundamentales de un DER son las entidades, cuya vista se representa con la clase SEERENTITYVIEW (con sus respectivas vistas alternas SEERENTITYVIEW1 y SEERENTITYVIEW2), y las interrelaciones, cuyas vistas se representan con la clase SEERRELATIONSHIPVIEW. Las vistas de entidades débiles se representan con las clases SEERWEAKENTITYVIEW, SEERWEAKENTITYVIEW1 o SEERWEAKENTITYVIEW2. La representación gráfica de una entidad agregada (SEERAGGREGATEVIEW) hereda de SEERELATIONSHIPVIEW, y tiene también sus vistas alternas con las clases SEERAGGREGATEVIEW1 y SEERAGGRGATEVIEW2. Finalmente, las vistas de interrelaciones débiles heredan también de la interrelación normal y se representan con objetos de la clase SEERWEAKRELATIONSHIP.

La creación de los elementos de un DER se lleva a cabo mediante los botones de las barras de herramientas de la ventana principal. Los botones usados en esta herramienta son subclases de la clase SEDRAWNBUTTON, que implanta botones con icono. El diagrama de la figura 4.7 muestra la relación de herencia entre todos los botones. El botón SEERENTITYBUTTON se utiliza para crear entidades, mientras que SEERRELATIONSHIPBUTTON se usa para crear interrelaciones; SEERAGGREGATE, para elementos agregados; SEERGENERALIZATIONBUTTON, para crear relaciones de generalización-especialización; SEERWEAKENTITYBUTTON, para crear entidades débiles y SEERWEAKRELATIONSHIPBUTTON, para crear interrelaciones débiles. Las asociaciones pueden ser de cuatro tipos: simple, con obligatoriedad, con orden de lectura y con obligatoriedad y orden de lectura. Esas asociaciones se crean utilizando botones de las clases SEERASSOCIATIONBUTTON, SEERASSOCIATIONMBUTTON, SEERASSOCIATIONROBUTTON y SEERASSOCIATIONMROBUTTON, respectivamente.

Para crear un elemento se realiza un proceso que consta de dos pasos: 1) la selección del elemento a crear, que es lo que el usuario hace al hacer clic en el botón correspondiente, y 2) la creación del elemento que el usuario realiza mediante un clic del mouse en la posición del área de trabajo en que desea que el nuevo elemento sea colocado.

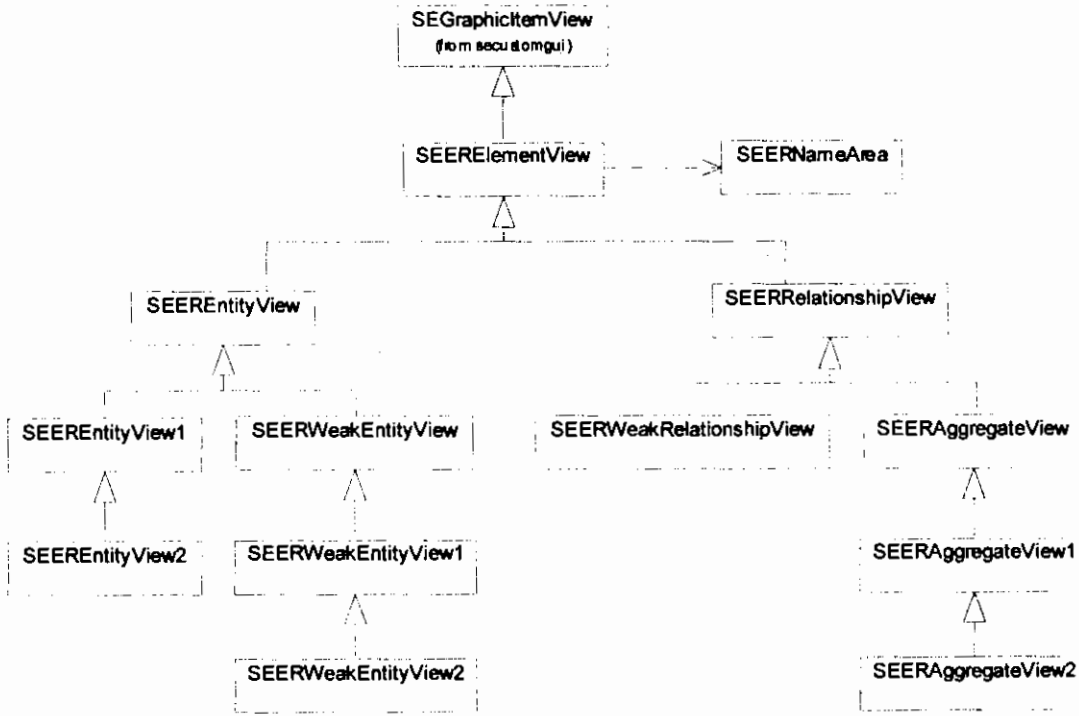

Figura 4.6. Diagrama de clases de las vistas de los elementos de un DER.

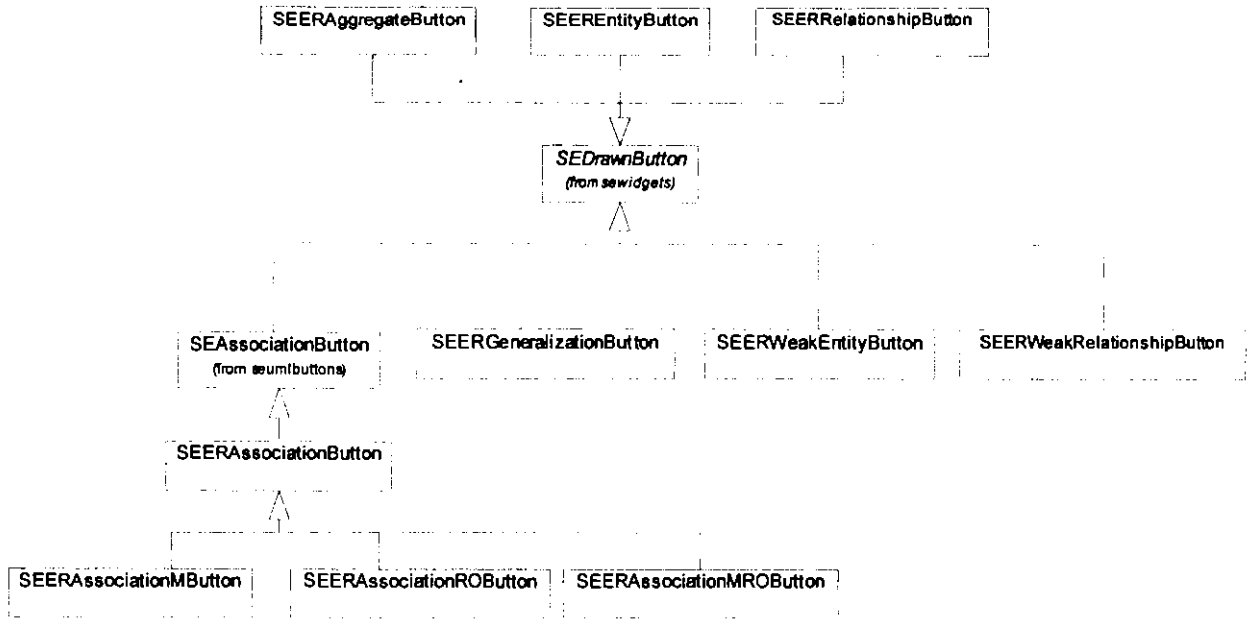

Figura 4.7. Diagrama de clases con la jerarquía de herencia de los botones.

Los dos pasos se realizan a través de comandos, responsables de la ejecución de la operación correspondiente. El diagrama de la figura 4.8 muestra los comandos asociados a los botones. Los comandos SEERSETCREATEAGGREGATE, SEERSETCREATEENTITY, SEERSETCREATE-GENERALIZATION, SEERSETCREATERELATIONSHIP, SEERSETCREATEWEAKENTITY, y SEERSET-CREATEWEAKRELATIONSHIP seleccionan un comando de la clase SEERCREATEAGGREGATE, SEERCREATEENTITY, SEERCREATEGENERALIZATION, SEERCREATERELATIONSHIP, SEERCREATE-WEAKENTITY o SEERCREATEWEAKRELATIONSHIP para su ejecución, mediante la cual crean sus elementos correspondientes.

4.3. CASOS DE USO

Las operaciones que el usuario puede realizar se describen y especifican mediante diagramas de casos de uso. El diagrama de la figura 4.9 representa los casos de uso para la creación de elementos de un DER.

Para crear cada uno de los elementos hay una secuencia común de pasos representada por el caso de uso CREATEITEM, siendo la única diferencia la manera en que se crea la vista de cada elemento. Los casos de uso CREATEAGGREGATEVIEW, CREATEENTITYVIEW, CREATERELATIONSHIPVIEW, CREATEWEAKENTITYVIEW y CREATEWEAKRELATIONSHIPVIEW corresponden a la creación de las vistas específicas de cada tipo de elemento.

En la siguiente sección se describirán los detalles de los casos de uso CreateItem y CreateEntityView; los otros casos se comportan de manera similar.

4.4. ESTRUCTURA DINÁMICA (DIAGRAMAS DE SECUENCIA)

Cada caso de uso se describe por medio de un diagrama de secuencia que especifique el detalle del comportamiento contenido en él.

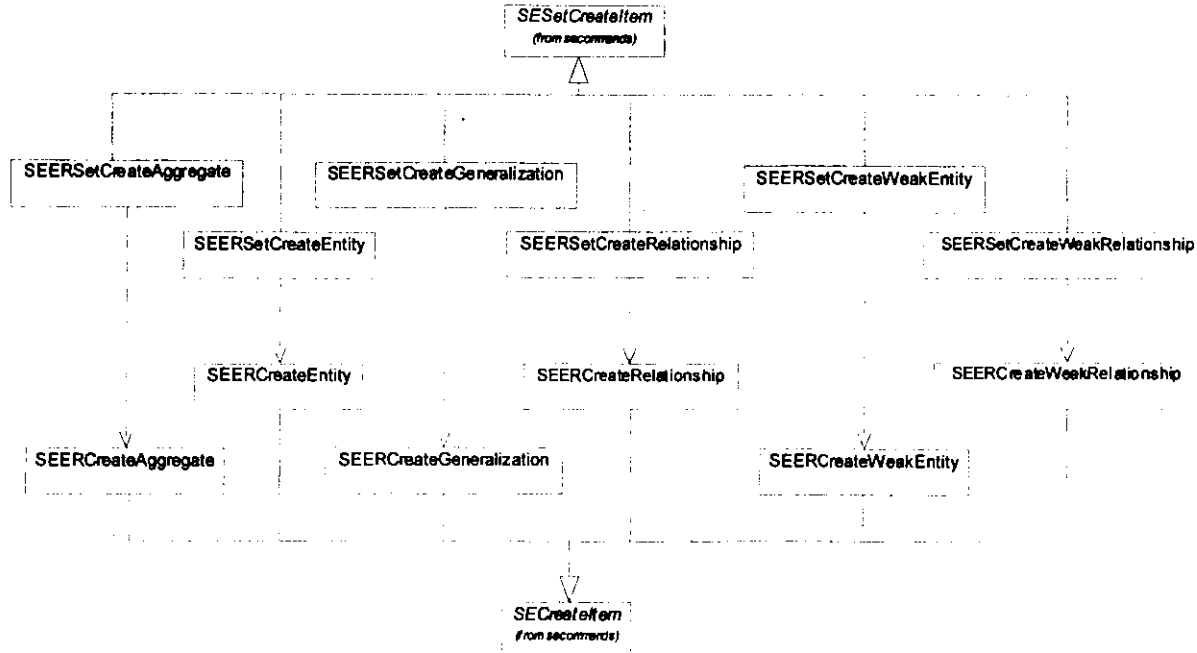

Figura 4.8. Diagrama de clases de los comandos de la aplicación.

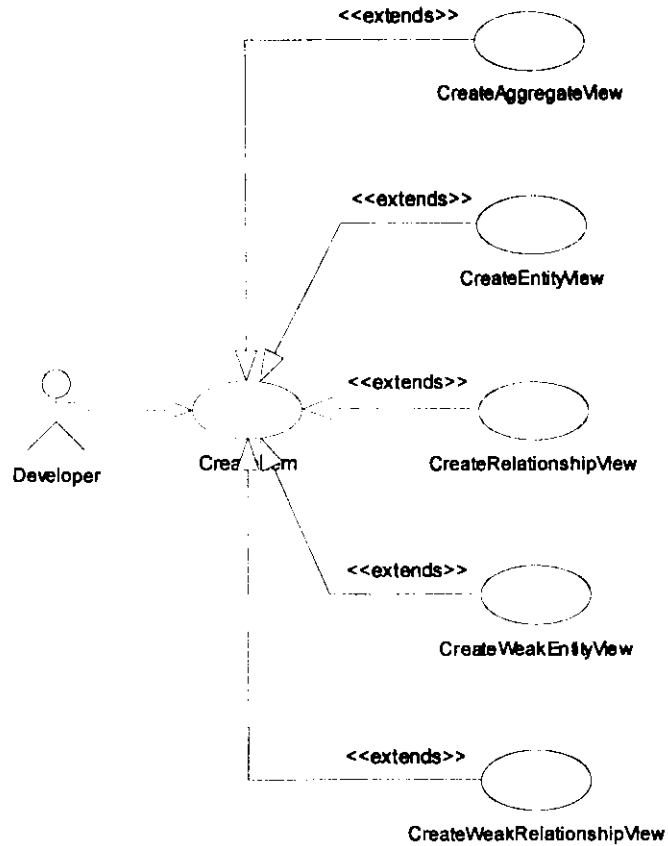

Figura 4.9. Diagrama de casos de uso para la creación de elementos de un DER.

Como se mencionó en la sección anterior, el proceso de creación de un elemento de un DER inicia con el caso de uso CreateItem. La figura 4.10 muestra el diagrama de secuencia donde se observa la sucesión de mensajes entre los objetos que intervienen en la creación de un elemento del DER. Esa sucesión es la siguiente:

1. Se obtiene la posición del mouse donde se habrá de dibujar el elemento.
2. Se crea la vista del elemento que corresponda.
3. Se agrega el comando de creación al histórico de comandos.
4. Se selecciona el objeto creado.
5. Se restablece el estado del sistema.

El diagrama de la figura 4.11 se observa el caso concreto de creación de una vista de entidad, ejecutándose las siguientes acciones:

1. El comando envía el mensaje de creación de una vista de entidad correspondiente a la vista del diagrama.
2. La vista del diagrama envía a su fábrica el mensaje de creación de la vista de entidad.
3. La fábrica crea la vista de entidad en la posición correspondiente del área de trabajo.

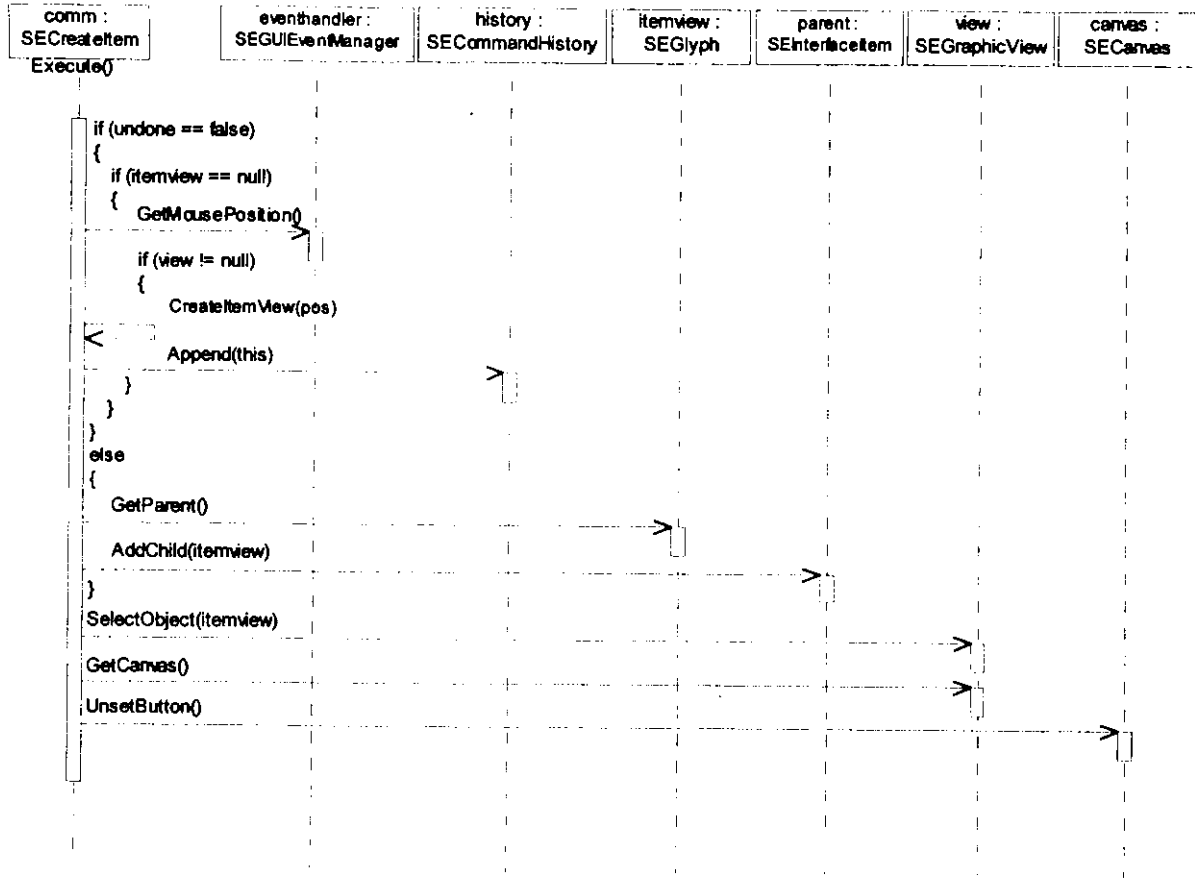

Figura 4.10. Diagrama de secuencia para el caso de uso "CREATEITEM".

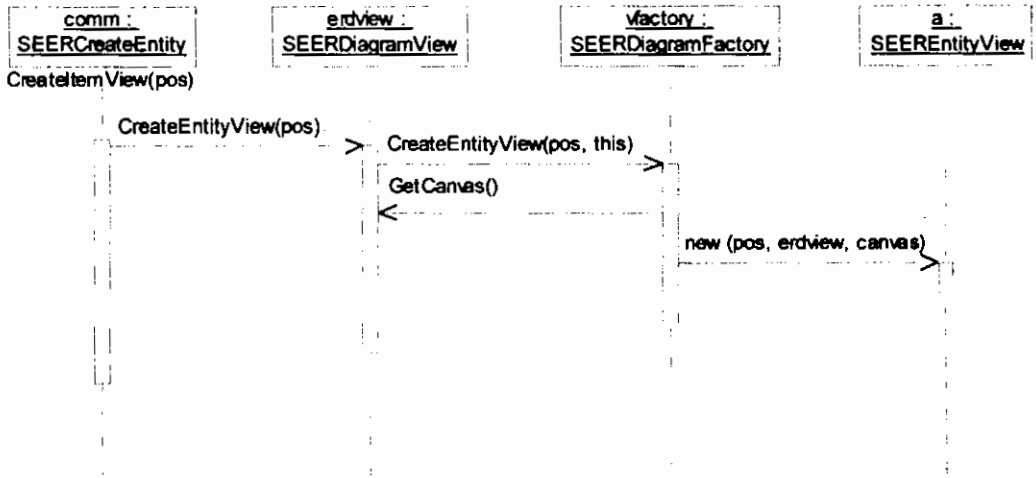

Figura 4.11. Diagrama de secuencia de la creación de una entidad.

CONCLUSIONES

Las necesidades de herramientas para el modelado de datos en el mercado mexicano han sido parcialmente satisfechas por productos extranjeros que aunque son de calidad aceptable, presentan algunas desventajas tales como el incumplimiento de principios básicos de diseño o funcionalidad, el alto costo del producto, la falta o costo excesivo del soporte técnico, etc.

Este tipo de problemas con el software se han presentado en México debido a la enorme dependencia tecnológica que mantiene nuestro país con relación a los países industrializados. La industria nacional del software ha avanzado ampliamente en el ámbito del desarrollo de sistemas a la medida, sin embargo, las herramientas para la ingeniería de software no han sido consideradas como una opción de desarrollo en nuestro país.

La herramienta de origen extranjero mencionada en la introducción, ERWin, incorpora aspectos tales como la generación de código SQL, la ingeniería inversa, la inclusión de interfaces para varios DBMS y el manejo de versiones, las cuales van más allá de los alcances de esta tesis. Por otra parte, la herramienta desarrollada en esta tesis presenta una ventaja sobre ERWin, y es que el usuario no se ve forzado a representar las interrelaciones como entidades, respetándose así la semántica del modelo. Las características adicionales de que la herramienta desarrollada en esta tesis carece representan posibles temas para trabajos de tesis relacionados, que complementen el presente o bien que exploren otras áreas del mercado mexicano que requieren soluciones de software que los productos extranjeros no han podido proporcionar

La aportación de esta tesis es el desarrollo de una herramienta que permite crear modelos conceptuales de bases de datos utilizando diagramas entidad–relación. La funcionalidad presentada en el capítulo 3 muestra la sencillez con que un usuario puede crear este tipo de modelos, por lo tanto el objetivo planteado para esta tesis se ha cumplido.

APÉNDICE A

EL LENGUAJE DE PROGRAMACIÓN JAVA

A.1. INTRODUCCIÓN

La herramienta objeto de esta tesis se ha desarrollado utilizando el lenguaje de programación Java. La selección de este lenguaje se basó fundamentalmente en su portabilidad, pero también por ser un lenguaje orientado a objetos que permite aplicar los más avanzados conceptos de ingeniería de software que actualmente rigen el mercado del desarrollo de sistemas.

Este apéndice no pretende ser un curso ni un tutorial de Java, sino una guía de los conceptos aplicados en esta herramienta.

A.1.1. LA PLATAFORMA JAVA

Java es una plataforma de software que permite el desarrollo de programas independientes del ambiente de hardware y sistema operativo en que deberá ejecutarse (ya sea Windows, Unix, Macintosh, NetWare, etc.).

La portabilidad de las aplicaciones Java se logra mediante el ***bytecode***. El bytecode es el código que resulta de la compilación de programas fuente escritos en el lenguaje de programación Java que contienen instrucciones para una ***máquina virtual***. Una vez generado el bytecode el intérprete de Java se encarga de la ejecución en la máquina física correspondiente.

La plataforma Java está formada por dos componentes principales:

- La ***máquina virtual Java*** o ***JVM*** (*"Java Virtual Machine"*), es la parte del ambiente de ejecución que se encarga de la interpretación del bytecode y tiene una implantación para cada plataforma subyacente.

- La ***interfaz de programación de aplicaciones de Java*** o ***Java API*** (*"Application Programming Interface"*), que es la especificación de las clases básicas de Java.

A.2. FUNDAMENTOS DEL LENGUAJE

En esta sección se describen los elementos fundamentales del lenguaje de programación Java, tales como comentarios, declaraciones, identificadores, variables, etc.

A.2.1. COMENTARIOS

Existen 3 estilos para insertar comentarios:

```
// En una línea
/* En una o
más líneas */
/** Comentario de documentación, interpretado por
   la herramienta javadoc. */
```

Los comentarios de documentación (colocados inmediatamente antes de una declaración) indican que el comentario deberá incluirse en cualquier documentación generada automáticamente mediante la herramienta `javadoc`.

A.2.2. DECLARACIONES

Una declaración es la unidad mínima ejecutable en un programa y generalmente termina con ';':

```
int cont;
pi = 2.1415;
```

A.2.3. IDENTIFICADORES

Los identificadores son nombres de variables, métodos, clases y objetos – todo lo que el programador necesita identificar y utilizar. En Java un identificador inicia con una letra, carácter de subrayado o $. Los caracteres subsecuentes pueden contener dígitos. No tienen una longitud máxima y son sensibles a mayúsculas-minúsculas.

A.2.4. TIPOS DE DATOS

Existen dos categorías de tipos de datos: primitivos y referencias.

Los tipos de datos primitivos son los que se muestran en la siguiente tabla:

TIPO	NOMBRE	TAMAÑO EN BITS / VALORES
Enteros	byte	8
	short	16
	int	32
	long	64
Caracteres	char	16
Punto flotante	float	32
	double	64
Booleanos	boolean	TRUE | FALSE

El valor de las variables que no son de tipo primitivo son precisamente referencias a valores o conjuntos de valores representados por la variable, como en el caso de los arreglos, las clases y las interfaces.

A.2.5. VARIABLES

La declaración de una variable está formada por dos partes: su tipo de datos y su nombre.

El nombre de una variable debe satisfacer las siguientes reglas:

- Debe ser un identificador legal comprendido dentro de una serie de caracteres ***UNICODE***.
- No debe ser una palabra reservada ni una literal booleana.
- No debe tener el mismo nombre que otra variable cuya declaración aparezca en el mismo alcance.

El lugar donde se declara una variable establece su alcance, el cual puede ser uno de las siguientes cuatro categorías:

- ***Variable de instancia.*** Pertenece a una clase o a un objeto, puede ser declarada en cualquier parte de una clase pero no en un método. La variable es accesible a todo el código en la clase.

- ***Variable de clase.*** Se declara como `static` en una clase, lo que significa que todas las instancias de esa clase tienen acceso a la misma variable, es decir, al mismo valor.

- ***Variable local.*** Se declara en cualquier parte de un método o en un bloque dentro de un método. El alcance de una variable local es el bloque donde fue declarada.

- ***Constante***. Se declara como `final`, lo que significa que su valor no puede ser modificado después de su inicialización.

- ***Parámetro de un método.*** Son argumentos formales utilizados para pasarle valores al método. El alcance de un parámetro es todo el método.

- ***Parámetro de un manejador de excepción***. Su alcance se restringe a un bloque de manejo de excepción.

A.2.6. ARREGLOS

Es posible declarar arreglos de cualquier tipo:

```
char   letras[];
int    []arreglo;
String nombres[];
```

La posición de los paréntesis ([]) determina cómo se aplican a la lista, así int []x, y[] equivale a int x[], y[][].

Es posible crear un arreglo utilizando `new`. Incluso se pueden crear arreglos de arreglos:

```
int tabla [][] = new int [4][5];
```

En Java un arreglo es un objeto. Para obtener la longitud de un arreglo es necesario consultar el contenido de su variable miembro `length`.

También se pueden crear arreglos con valores iniciales, de la siguiente manera:

```
String nombres[] = {"Angel", "Mónica", "Arturo", "Celia"};
```

La línea de código anterior equivale a:

```
String nombres[];
nombres = new String[4];
nombres[0] = new String("Angel");
```

```
nombres[1] = new String("Mónica");
nombres[2] = new String("Arturo");
nombres[3] = new String("Celia");
```

A.2.7. OPERADORES

En la siguiente lista se muestra la lista de operadores en orden de precedencia (asociativos de izquierda a derecha I-D y de derecha a izquierda D-I).

	OPERADOR					
I-D	.	[]	()			
I-D	++	--				
I-D	!	~	(cast)			
I-D	*	/	%			
I-D	+	-				
I-D	<<	>>(con signo)	>>>(sin signo)			
I-D	<	>	<=	>=	instanceof	
I-D	==	!=				
I-D	&					
I-D	^					
I-D	|					
I-D	&&					
I-D	||					
D-I	?					
D-I	=	*=	/=	%=	+=	-=
	<<=	>>=	>>>=	&=	^=	|=

El operador + se utiliza para concatenar objetos de la clase String.

A.2.8. CONTROL DE FLUJO

La siguiente tabla muestra las estructuras de control de flujo disponibles en Java:

Bifurcación	`if, else`	`if (boolean) { sentencias; } else { sentencias; }`
	`switch`	`switch (expr1) { case expr2: sentencias; break; case expr3: sentencias; break; case expr4: sentencias; break; default: sentencias; break; }`
Ciclos	`for`	`for (exp_inic; exp_prueba; exp_incremento) { sentencias; }`
	`while`	`while (boolean) { sentencias; }`
	`do`	`do { sentencias; } while (boolean);`

Las sentencias `break`, `continue`, `return` y `label` se utilizan también para controlar el flujo del programa.

A.2.9. CADENAS DE CARACTERES

Una secuencia de caracteres se conoce como cadena y en Java ha sido implantada mediante la clase `String` (miembro del paquete `java.lang`). Los objetos de la clase `String` no pueden ser modificados una vez que se les ha asignado un valor. La clase `StringBuffer` es una clase de cadenas modificables, utilizadas en lugar de objetos de tipo `String` cuando se sabe que el valor de la cadena puede variar.

Las literales de cadena se representan como cadenas entre comillas dobles.

Algunos métodos de las clases `String` y `StringBuffer` son:

MÉTODO	DESCRIPCIÓN
length()	Devuelve el tamaño de la cadena
charAt(int n)	Retorna el carácter ubicado en la posición n de la cadena.
indexof(int character) lastindexof(int character)	Devuelve la posición de la primera ocurrencia del carácter en la cadena,
LastIndexOf(int character)	Devuelve la posición de la última ocurrencia del carácter en la cadena.

A.3. CLASES Y OBJETOS

Las clases representan tipos de datos definidos por el usuario. La declaración de una clase tiene la siguiente forma:

```
[public] [abstract] [final] class <Nombre_clase>
        [extends <super>] [implements <inter>]
{
   <variables de instancia y de clase y métodos>
}
```

Donde:

- `public` indica que la clase es accesible por clases de otros paquetes, ya que si no se especifica así, la clase es visible sólo por otras dentro del mismo paquete.

- `abstract` indica que la clase no puede ser instanciada.

- `final` indica que la clase no puede fungir como superclase para otras clases.

- `extends` identifica a super como la superclase de la clase declarada, insertando así a la clase dentro de la jerarquía de herencia.

- `implements` indica que la clase se comportará como la interfaz a la que implementa, es decir, debe implementar los métodos definidos en esa interfaz.

Una ***clase*** se define en un solo archivo cuyo nombre debe ser exactamente igual al de la clase que contiene y cuya extensión deberá ser ***.java***.
Los niveles de visibilidad son:

- ***Public***. Los objetos de la clase son visibles para los de cualquier otra.

- ***Package***. Los objetos de la clase son visibles sólo para los objetos de las clases definidas dentro del mismo paquete.

- ***Protected***. Los objetos de esa clase sólo son accesibles por objetos de la misma clase y sus subclases.

- ***Private***. Los objetos sólo son accesibles para objetos de la misma clase.

En Java los objetos se crean instanciando una clase, esto es, utilizando `new` seguido del constructor:

```
new Motorcycle();
```

Cuando se crea el objeto Java reserva memoria para él. `new` se utiliza para reservar la memoria que ocupará el objeto y el constructor lo inicializa.

Todo objeto tiene un tiempo de vida durante la ejecución del programa y durante ese tiempo utiliza recursos. Cuando deja de existir referencia alguna hacia un objeto debe liberarse el espacio en memoria que utilizaba, con el fin de evitar que el programa acabe con la memoria disponible. En Java la recolección y liberación de memoria es responsabilidad de un proceso conocido como el **colector automático de basura**. Este proceso monitorea el alcance del objeto y lo marca si ha salido de él. Este proceso lleva un registro de toda la memoria reservada con `new` y quién y

cuántas veces se tiene acceso a ella; cuando el número de accesos llega a cero, entonces la memoria puede recolectarse y liberarse.

A.4. HERENCIA

En la parte superior de la jerarquía de clases de Java está la clase `Object`, de la cual heredan todas las demás. Es importante destacar que en Java no existe la herencia múltiple.

La herencia en Java se indica con la palabra reservada `extends` después del nombre de la clase, seguida del nombre de la superclase, de la siguiente manera:

```
public class Subclase extends Superclase
{
   ...
}
```

La clase del siguiente ejemplo es un applet. Un applet debe heredar de la clase `java.applet.Applet`:

```
public class HelloApplet extends java.applet.Applet
{
   Font f=new Font ("TimesRoman", Font.BOLD,36);

   public void Paint(Graphics g)
   {
      g.setFont(f);
      g.setColor(Color.red);
      g.drawString("Hello, world!", 5, 25);
   }
}
```

Algunas veces es necesario hacer referencia a la instancia de una clase desde el objeto de una subclase, para lo cual se utiliza `super`. Cuando se desea hacer referencia al objeto mismo se utiliza `this`.

A.5. PAQUETES

Las clases se organizan mediante el uso de paquetes o "***packages***". Para indicar que una clase es parte de un paquete se incluye una instrucción como la siguiente como primera línea del archivo de la clase:

```
package softengine.seerd.seergui;
```

La clase que contiene la línea anterior pertenece al paquete `seerdgui`, el cual está contenido en el paquete `seerd` y éste a su vez se encuentra dentro del paquete `softengine`.

Para hacer referencia a una clase que pertenece a otro paquete es necesario importarla, mediante una sentencia como esta:

```
import softengine.sebasic.*;
```

El asterisco significa que se importarán todas las clases del paquete `sebasic`; también se puede especificar una sola clase, colocando su nombre completo en lugar del asterisco.

A.6. INTERFACES

Una **interfaz** es un conjunto de declaraciones de métodos y variables de instancia. Este conjunto define un comportamiento que puede ser implantado por una clase.

Para crear una interfaz se utiliza una sentencia como la siguiente:

```
[public] interface <nombre> [extends <superinterfaces>]
{
    <constantes y signaturas de métodos>
}
```

Donde:

- `public` indica que la interfaz es visible para cualquier clase. Si se omite solo es visible dentro del mismo package.
- `extends` especifica las superinterfaces de la interfaz.

Aunque en Java no existe la herencia múltiple utilizando `extends`, es posible implantarla mediante la implantación de más de una interfaz por una clase determinada.

A.7. MANEJO DE EXCEPCIONES

Cuando ocurre una condición anormal dentro de un método, el método puede lanzar una ***excepción*** para indicar a quien lo llamó que se ha presentado y el tipo de que se trata. El método del objeto cliente puede contener un manejador de excepciones para controlar el flujo de control del programa.

Un manejador de excepciones se crea colocando una parte de código dentro de un bloque `try`, y definiendo un bloque `catch` para cada tipo de excepción que deba manejarse.

Se puede crear un bloque `finally` para realizar tareas de "limpieza" independientemente de lo que haya ocurrido dentro del bloque `try`. Esa limpieza puede referirse al cierre de archivos o la liberación de otros recursos del sistema.

Java proporciona diversas excepciones predefinidas, tales como:

- `ArithmeticException`. Como resultado de una división entre cero,
- `NullPointerException`. Cuando se intenta acceder a un objeto o método antes de instanciarlo), etc.
- `ClassCastException`. Cuando se intenta convertir un objeto de una clase a otra que no es válida.
- `ArrayIndexOutofBoundsException`. Cuando se intenta acceder a un elemento de un arreglo más allá de la definición original del tamaño del arreglo.

Es posible crear excepciones definidas por el programador, con la siguiente sintaxis:

```
class NuevaExcepcion extends Exception{}
```

Para lanzar una excepción definida por el programador se utiliza la siguiente sintaxis:

```
throw new nuevaExcepcion();
```

A.8. CREACIÓN DE INTERFACES GRÁFICAS DE USUARIO

Una interfaz de usuario es cualquier tipo de comunicación entre un programa y sus usuarios. Java permite crear interfaces gráficas para ambientes de ventanas tales como Windows o Motif mediante el API del AWT.

A.8.1. LA INTERFAZ DE USUARIO DE JAVA: AWT

El ***AWT*** ("*Abstract Windowing Toolkit*") contiene una serie de clases de propósito general y multiplataforma para la programación de interfaces gráficas de usuario o ***GUI*** ("*Graphic User Interface*"). Estas clases van desde componentes del ambiente de ventanas hasta clases para dibujo y manejo de eventos.

Las clases para el ambiente de ventanas del AWT pueden dividirse en dos grupos:

- Componentes.
- Componentes de menú.

Para desplegar un componente es necesario agregarlo a un contenedor ("Container"). Un componente sólo pertenece a un contenedor.

A.8.2. COMPONENTES

La raíz de la jerarquía de clases (como se observa en la figura A.1) del AWT es la clase `Component`, la cual es una clase abstracta que define los elementos comunes a todos los componentes de una GUI.

La clase `Component` proporciona servicios tales como:

- Apoyo para dibujo.
- Manejo de eventos
- Control de apariencia: fonts, color, visibilidad.
- Habilitación e inhabilitación de componentes.

- Manejo de imágenes.
- Cursores
- Control de tamaño y posición en la pantalla
- Componentes ligeros ("lightweight").

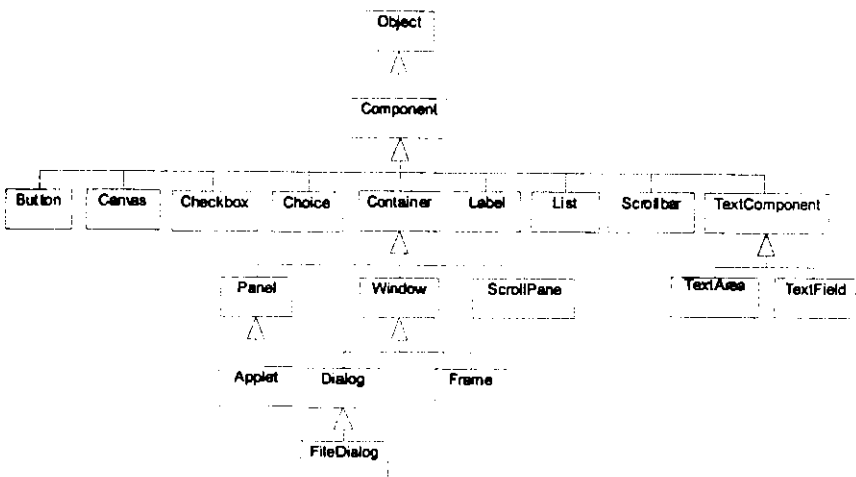

Figura A.1. Jerarquía de clases del AWT.

VENTANAS

El manejo de ventanas se realiza a través de la clase `Frame`. Cualquier aplicación requiere de al menos un frame.

Para responder a los eventos es necesario que el `Frame` tenga un escucha de ventana ("window listener") que implemente los métodos relativos a las operaciones básicas que el usuario puede realizar sobre una ventana (cerrar, minimizar, activar, desactivar, abrir, cerrar, etc.).

CAJAS DE DIÁLOGO

La clase `Dialog` define aquellas ventanas que dependen necesariamente de otras. Una caja de diálogo puede ser modal (cuando inhabilita el acceso a la ventana principal) o no modal. Por omisión, las cajas de diálogo no son modales.

ETIQUETAS

Las etiquetas son cadenas de texto que el usuario no puede editar ni seleccionar. Para colocar etiquetas en una GUI se utilizan objetos de la clase `Label`, las cuales por omisión se alinean a la izquierda del área de dibujo.

CAMPOS Y ÁREAS DE TEXTO

Para manejar campos de texto en una sola línea se utilizan objetos de la clase `TextField`, en tanto que para un campo multilínea se usan objetos de la clase `TextArea`.

BOTONES

La clase `Button` permite crear botones de texto.

A.8.3. COMPONENTES DE MENÚ

Existen dos tipos de menús: "Pull-down" y "Pop-up".

El menú "Pop up" se puede agregar a cualquier objeto de un a clase que herede de `Component`.

Un menú sólo puede existir en una barra de menú o como un submenú de otro menú. La barra de menú se crea instanciando la clase `MenuBar` y debe agregarse a una ventana. Los elementos de un menú pueden ser instancias de la clase `MenuItem`, cuando ejecutarán alguna acción o de la clase `CheckboxMenuItem`, cuando pueden tener solo uno de dos estados posibles.

APÉNDICE B

CLASES

B.1. PAQUETE softengine.seerd.seerdapp

SEERDAPP.JAVA

```
package softengine.seerd.seerdapp;

import softengine.seerd.seerdgui.SEERDiagramWindow;

public class seerdapp
{
   public seerdapp();
   public static void main(String args[]);
}
```

B.2. PAQUETE softengine.seerd.seerdgui

SEERAGGREGATEVIEW.JAVA

```
package softengine.seerd.seerdgui;

import softengine.segui.sewidgets.*;
import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.secustomgui.*;
import softengine.seerd.seerdmc.SEERAggregate;
import java.awt.*;

public class SEERAggregateView extends REERRelationshipView//SEERElementView
{
  public static final int   XOFFSET   = 4;
  public static final int   YOFFSET   = 4;

  public SEERAggregateView(SEPoint pos, SEGlyph parent, SECanvas drawing);
  protected SEERAggregateView(SEPoint pos, int width, int height, SEGlyph parent,
                              SECanvas drawing);

  protected boolean AllowedType(Object obj);
  public void Draw(SECanvas cv);
}
```

SEERAGGREGATEVIEW1.JAVA

```
package softengine.seerd.seerdgui;

import softengine.sebasic.*;
import softengine.seerd.seerdme.*;
import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.sewidgets.*;
import softengine.segui.secustomgui.SEGlyph;
import java.awt.*;


public class SEERAggregateView1 extends SEERAggregateView
{
  public SEERAggregateView1(SEPoint pos, SEGlyph parent, SECanvas drawing);
  public SEERAggregateView1(SEPoint pos, int width, int height, SEGlyph parent,
                            SECanvas drawing);
  public void Draw(SECanvas cv);
}
```

SEERAGGREGATEVIEW2.JAVA

```
package softengine.seerd.seerdgui;

import softengine.seerd.seerdme.*;
import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.sewidgets.*;
import softengine.segui.secustomgui.*;
import java.awt.*;

public class SEERAggregateView2 extends SEERAggregateView1
{
  public SEERAggregateView2(SEPoint pos, SEGlyph parent, SECanvas drawing);
  public SEERAggregateView2(SEPoint pos, int width, int height, SEGlyph parent,
                            SECanvas drawing);
  public void Draw(SECanvas cv);
}
```

SEERAGGREGATEVIEW2.JAVA

```
package softengine.seerd.seerdgui;

import softengine.segui.seguifw.SEInterfaceItem;
import java.awt.Color;
import softengine.segui.sewidgets.SEPen;
import softengine.sebuttons.seumlbuttons.SEAssociationButton;
import softengine.sebasic.SECommand;

public class SEERAssociationButton extends SEAssociationButton
{
  public SEERAssociationButton(SEInterfaceItem par);
  public SEERAssociationButton(SEInterfaceItem par, SECommand comm);
  public void Draw();
}
```

SEERASSOCIATIONENDVIEW.JAVA

```
package softengine.seerd.seerdgui;

import softengine.sebasic.*;
import softengine.segui.sewidgets.*;
import softengine.segui.secustomgui.*;
import softengine.segui.sewidgets.SEPen;

public class SEERAssociationEndView extends SEAssociationEndView
{
   protected SEERAssociationView ep;

   public SEERAssociationEndView(SEGraphicItemView g, SEGlyph parent,
                                 SECanvas drawing);
   public SEERAssociationEndView(SEGraphicItemView g, SEGlyph parent,
                              byte agg, boolean nav, SECanvas drawing);

   protected boolean AllowedType(Object obj);
   public void Draw(SECanvas cv);
}
```

SEERASSOCIATIONMBUTTONJAVA

```
package softengine.seerd.seerdgui;

import softengine.segui.seguifw.SEInterfaceItem;
import java.awt.Color;
import softengine.segui.sewidgets.SEPen;
import softengine.sebuttons.seumlbuttons.SEAssociationButton;
import softengine.sebasic.SECommand;

public class SEERAssociationMButton extends SEERAssociationButton
{
  public SEERAssociationMButton(SEInterfaceItem par);
  public SEERAssociationMButton(SEInterfaceItem par, SECommand comm);
  public void Draw();
}
```

SEERASSOCIATIONROBUTTONJAVA

```
package softengine.seerd.seerdgui;

import softengine.segui.seguifw.SEInterfaceItem;
import java.awt.Color;
import softengine.segui.sewidgets.SEPen;
import softengine.sebuttons.seumlbuttons.SEAssociationButton;
import softengine.sebasic.SECommand;

public class SEERAssociationROButton extends SEERAssociationButton
{
  public SEERAssociationROButton(SEInterfaceItem par);
  public SEERAssociationROButton(SEInterfaceItem par, SECommand comm);
  public void Draw();
}
```

SEERASSOCIATIONMROBUTTONJAVA

```
package softengine.seerd.seerdgui;

import softengine.segui.seguifw.SEInterfaceItem;
import java.awt.Color;
import softengine.segui.sewidgets.SEPen;
import softengine.sebuttons.seumlbuttons.SEAssociationButton;
import softengine.sebasic.SECommand;

public class SEERAssociationMROButton extends SEERAssociationButton
{
  public SEERAssociationMROButton(SEInterfaceItem par);
  public SEERAssociationMROButton(SEInterfaceItem par, SECommand comm);
  public void Draw();
}
```

SEERASSOCIATIONVIEW.JAVA

```
package softengine.seerd.seerdgui;

import softengine.segui.sewidgets.*;
import softengine.segui.secustomgui.*;

public class SEERAssociationView extends SEAssociationView
{
   private boolean mandatory;
   private boolean reading;
   private String  cardinality;

   public SEERAssociationView(SEGraphicItemView g, SEGraphicView parent, boolean man,
                              boolean read, String card, SECanvas drawing)
                                                    throws SENotAllowedAssocEnd;
   private boolean AllowedSourceSimpleAsso(SEGraphicItemView g);
   protected boolean AllowedSource(SEGraphicItemView g);
   protected boolean AllowedTarget(SEGraphicItemView g);
   protected void CreateSource(SEGraphicItemView item, SECanvas drawing);
   protected void CreateTarget(SEGraphicItemView item, SECanvas drawing);
   protected void SourceException(SEGraphicItemView item) throws SENotAllowedAssocEnd ;
   protected void TargetException(SEGraphicItemView item) throws SENotAllowedAssocEnd;
}
```

SEERCREATEAGGREGATE.JAVA

```
package softengine.seerd.seerdgui;

import softengine.segui.secustomgui.*;
import softengine.secommands.*;
import softengine.sebasic.*;
import softengine.secommands.SECreateItem;

public class SEERCreateAggregate extends SECreateItem
{
   public SEERCreateAggregate(SEGraphicView docview);

   public SECommand Clone();
   protected SEGlyph CreateItemView(SEPoint pos);
}
```

SEERCREATEASSOCIATION.JAVA

```
package softengine.seerd.seerdgui;

import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.secustomgui.*;
import softengine.secommands.*;
import softengine.segui.sewidgets.*;

public class SEERCreateAssociation extends SECreateAssociation
{
  public SEERCreateAssociation(SEGraphicView docview);

  public SECommand Clone();
  protected SEAssociationView CreateAssociation(SEGraphicItemView iobj)
                                        throws SEDontAllowedAssocEnd;
}
```

SEERCREATEENTITY.JAVA

```
package softengine.seerd.seerdgui;

import softengine.seerd.seerdme.*;
import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.secustomgui.*;
import softengine.secommands.SECreateItem;

public class SEERCreateEntity extends SECreateItem
{
   public SEERCreateEntity(SEGraphicView docview);

   public SECommand Clone();
   protected SEGlyph CreateItemView(SEPoint pos);
}
```

SEERCREATEGENERALIZATION.JAVA

```
package softengine.seerd.seerdgui;


import softengine.sebasic.*;
import softengine.segui.secustomgui.*;
import softengine.segui.sewidgets.*;
import softengine.secommands.*;
import java.awt.*;
import java.awt.event.*;

public class SEERCreateGeneralization extends SECreateAssociation
{
  public SEERCreateGeneralization(SEGraphicView docview);
  protected SEAssociationView CreateAssociation(SEGraphicView docview,
                                                SEGraphicItemView iobj)
                                        throws SEDontAllowedAssocEnd;
  public SECommand Clone();
}
```

SEERCREATERELATIONSHIP.JAVA

```
package softengine.seerd.seerdgui;

import softengine.seerd.seerdme.*;
import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.secustomgui.*;
import softengine.secommands.SECreateItem;

public class SEERCreateRelationship extends SECreateItem
{
  public SEERCreateRelationship(SEGraphicView docview);

  public SECommand Clone();
  protected SEGlyph CreateItemView(SEPoint pos);
}
```

SEERCREATEWEAKENTITY.JAVA

```
package softengine.seerd.seerdgui;

import softengine.seerd.seerdme.*;
import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.secustomgui.*;
import softengine.secommands.SECreateItem;

public class SEERCreateWeakEntity extends SECreateItem
{
  public SEERCreateWeakEntity(SEGraphicView docview);

  public SECommand Clone();
  protected SEGlyph CreateItemView(SEPoint pos);
}
```

SEERCREATEWEAKRELATIONSHIP.JAVA

```
package softengine.seerd.seerdgui;

import softengine.seerd.seerdme.*;
import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.secustomgui.*;
import softengine.secommands.SECreateItem;

public class SEERCreateWeakRelationship extends SECreateItem
{
  public SEERCreateWeakRelationship(SEGraphicView docview);


  public SECommand Clone();
  protected SEGlyph CreateItemView(SEPoint pos);
}
```

SEERDIAGRAM.JAVA

```
package softengine.seerd.seerdgui;

import softengine.sebasic.*;
import softengine.segui.*;
import softengine.seapp.*;
import java.awt.*;
import java.awt.event.*;

public class SEERDiagram extends SEDocument
{
  public SEERDiagram();
}
```

SEERDIAGRAMFACTORY.JAVA

```
package softengine.seerd.seerdgui;

import softengine.seapp.SEViewFactory;
import softengine.segui.secustomgui.SEAssociationView;
import softengine.segui.secustomgui.SEGraphicItemView;
import softengine.segui.secustomgui.SEGraphicView;
import softengine.segui.sewidgets.SECanvas;
import softengine.segui.secustomgui.SEDontAllowedAssocEnd;
import softengine.sebasic.SEPoint;
import softengine.segui.secustomgui.SEGlyph;

public class SEERDiagramFactory extends SEViewFactory
{
  public SEERDiagramFactory();

  public SEAssociationView CreateERAssociationView(SEGraphicItemView iobj,
                                                   SEGraphicView view , boolean mandatory,
                                                   boolean reading, byte cardinality,
                                                   SECanvas drawing)
                                                   throws SEDontAllowedAssocEnd;
  public SEERAttributeView CreateAttributeView(SEERAttribute a, SEERAttributeArea parent);
  public SEEREntityView CreateEntityView(SEPoint pos, SEGraphicView view);
  public SEERRelationshipView CreateRelationshipView(SEPoint pos, SEGraphicView view);
  public SEERWeakEntityView CreateWeakEntityView(SEPoint pos, SEGraphicView view);
  public SEERAggregateView CreateAggregateView(SEPoint pos, SEGraphicView view);
  public SEERWeakRelationshipView CreateWeakRelationshipView(SEPoint pos,
                                                             SEGraphicView view);
  public SEERGeneralizationView CreateGeneralizationView(SEGraphicItemView iobj,
                                                         SEGraphicView view,
                                                         SECanvas drawing)
                                                         throws SEDontAllowedAssocEnd;
}
```

SEERDIAGRAMFACTORY1.JAVA

```
package softengine.seerd.seerdgui;

import softengine.sebasic.SEPoint;
import softengine.segui.secustomgui.SEGraphicView;

public class SEERDiagramFactory1 extends SEERDiagramFactory
{
  public SEERDiagramFactory1();

  public SEERAggregateView CreateAggregateView(SEPoint pos, SEGraphicView view);
  public SEERAttributeView CreateAttributeView(SEGlyph parent);
  public SEEREntityView CreateEntityView(SEPoint pos, SEGraphicView view);
  public SEERWeakEntityView CreateWeakEntityView(SEPoint pos, SEGraphicView view);
}
```

SEERDIAGRAMFACTORY2.JAVA

```
package softengine.seerd.seerdgui;

import softengine.sebasic.SEPoint;
import softengine.segui.secustomgui.SEGraphicView;

public class SEERDiagramFactory2 extends SEERDiagramFactory
{
  public SEERDiagramFactory2();

  public SEERAggregateView CreateAggregateView(SEPoint pos, SEGraphicView view);
  public SEERAttributeView CreateAttributeView(SEGlyph parent);
  public SEEREntityView CreateEntityView(SEPoint pos, SEGraphicView view);
  public SEERWeakEntityView CreateWeakEntityView(SEPoint pos, SEGraphicView view);
}
```

SEERDIAGRAMVIEW.JAVA

```
package softengine.seerd.seerdgui;

import softengine.seerd.seerdme.*;
import softengine.sebasic.*;
import softengine.segui.secustomgui.*;
import softengine.segui.secustomgui.SEDontAllowedAssocEnd;
import softengine.segui.sewidgets.*;
import softengine.segui.*;
import softengine.seapp.SEViewFactory;

public class SEERDiagramView extends SEGraphicView
{
   protected SEERDiagramFactory vfactory;

   public SEERDiagramView();

   public SEERAggregateView CreateAggregateView(SEPoint pos);
   public SEERAssociationView CreateAssociationView(SEGraphicItemView iobj,
                                    SECanvas drawing) throws SENotAllowedAssocEnd;
   public SEAssociationView CreateERAssociationView(SEGraphicItemView iobj,
                                    SECanvas drawing) throws SENotAllowedAssocEnd;
   public SEAssociationView CreateGeneralizationView(SEGraphicItemView iobj)
                                          throws SENotAllowedAssocEnd;
   public SEEREntityView CreateEntityView(SEPoint pos);
   public SEERRelationshipView CreateRelationshipView(SEPoint pos);
   public SEERWeakEntityView CreateWeakEntityView(SEPoint pos);
   public SEERWeakRelationshipView CreateWeakRelationshipView(SEPoint pos);
   public SEViewFactory CreateViewFactory();
}
```

SEERDIAGRAMVIEW1.JAVA

```
package softengine.seerd.seerdgui;

import softengine.sebasic.*;
import softengine.segui.*;
import softengine.seapp.SEViewFactory;
import java.awt.*;
import java.awt.event.*;

public class SEERDiagramView1 extends SEERDiagramView
{
   public SEERDiagramView1();

   public SEViewFactory CreateViewFactory();
}
```

SEERDIAGRAMVIEW2.JAVA

```
package softengine.seerd.seerdgui;

import softengine.sebasic.*;
import softengine.segui.*;
import java.awt.*;
import java.awt.event.*;
public class SEERDiagramView2 extends SEERDiagramView1
{
  public SEERDiagramView2();
  public SEViewFactory CreateViewFactory();
}
```

SEERDIAGRAMWINDOW.JAVA

```
package softengine.seerd.seerdgui;

import softengine.secd.*;
import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.sewidgets.*;
import softengine.segui.secustomgui.*;
import softengine.seapp.SEApplication;
import softengine.sebuttons.*;
import softengine.sebuttons.segenbuttons.*;
import softengine.secommands.*;
import softengine.sebuttons.seumlbuttons.SEAssociationButton;
import java.awt.*;
import java.awt.event.*;

public class SEERDiagramWindow extends SEApplication
{
   protected SEERDiagramView view;
   private   SECommandArea   lcarea;   // Left command area.

   public SEERDiagramWindow(String doctitle);
   private void AddFileMenu(SEMainMenu mmenu);
   private void AddEditMenu(SEMainMenu mmenu);
   private void AddViewMenu(SEMainMenu mmenu);
   private void AddBrowseMenu(SEMainMenu mmenu);
   private void AddReportMenu(SEMainMenu mmenu);
   private void AddToolsMenu(SEMainMenu mmenu);
   private void AddOptionsMenu(SEMainMenu mmenu);
   private void AddPresentationSubmenu(SEMenu sm);
   private void AddHelpMenu(SEMainMenu mmenu);
   private void DefineCommands();
   public void DefineInterface();
   private void DefineMainMenu();
}
```

SEERELEMENTVIEW.JAVA

```
package softengine.seerd.seerdgui;

import softengine.segui.secustomgui.*;
import softengine.segui.sewidgets.*;
import softengine.seerd.seerdme.*;
import softengine.segui.secustomgui.SEGraphicItemView;
import softengine.secommands.*;
import softengine.sebasic.*;
import softengine.segui.*;
import java.awt.*;

public class SEERElementView extends SEGraphicItemView
{
  public static final int   EWIDTH        = 120; // Default Width
  public static final int   EHEIGHT       = 50;  // Default Height
  public static final int   CHILDHEIGHT   = 20;  // Default CHILD Area height.
  public static final int   XOFFSET       = 5;
  public static final int   YOFFSET       = 5;
  public static final int   SOFFSET       = 5;

  public SEERElementView(SEPoint pos, SEGlyph parent, SECanvas drawing);
  public SEERElementView(SEPoint pos, int width, int height, SEGlyph parent,
                         SECanvas drawing);
  public SEERElementView(int x, int y, int width, int height, SEGlyph parent,
                         SECanvas drawing);

  protected boolean AllowedType(Object o);
  protected void DrawSeparators(SECanvas cv, int dx);
  protected int NextChildYPosition();
  protected void SetAttributeAreas();
  protected void UpdateWidth();
  protected void InitElementView();
  protected  void SetElementView();
  public int GetChildYOffset();
  public int GetChildXOffset();
  public int GetSiblingOffset();
}
```

SEERENTITYBUTTON.JAVA

```
package softengine.seerd.seerdgui;

import softengine.segui.*;
import softengine.segui.sewidgets.*;
import softengine.segui.seguifw.SEInterfaceItem;
import softengine.sebasic.SECommand;
import java.awt.*;
import java.awt.event.*;

public class SEEREntityButton extends SEDrawnButton
{
  public SEEREntityButton(SEInterfaceItem par);
  public SEEREntityButton(SEInterfaceItem par, SECommand comm);

  public void Draw();
}
```

SEERENTITYVIEW.JAVA

```
package softengine.seerd.seerdgui;

import softengine.seerd.seerdme.SEEREntity;
import softengine.segui.sewidgets.*;
import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.secustomgui.*;
import java.awt.*;

public class SEEREntityView extends SEERElementView
{
   public SEEREntityView(SEPoint pos, SEGlyph parent, SECanvas drawing);
   protected SEEREntityView(SEPoint pos, int width, int height, SEGlyph parent,
                            SECanvas drawing);

   protected void SetName(SECanvas drawing);
   protected boolean AllowedType(Object obj);
   public void Draw(SECanvas cv);
}
```

SEERENTITYVIEW1.JAVA

```
package softengine.seerd.seerdgui;

import softengine.sebasic.*;
import softengine.seerd.seerdme.*;
import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.sewidgets.*;
import softengine.segui.secustomgui.SEGlyph;
import java.awt.*;

public class SEEREntityView1 extends SEEREntityView
{
  public SEEREntityView1(SEPoint pos, SEGlyph parent, SECanvas drawing);
  protected SEEREntityView1(SEPoint pos, int width, int height, SEGlyph parent,
                            SECanvas drawing);

  public void Draw(SECanvas cv);
}
```

SEERENTITYVIEW2.JAVA

```
package softengine.seerd.seerdgui;

import softengine.seerd.aeerdme.*;
import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.sewidgets.*;
import softengine.segui.secustomgui.*;
import java.awt.*;

public class SEEREntityView2 extends SEEREntityView1
{
  public SEEREntityView2(SEPoint pos, SEGlyph parent, SECanvas drawing);
}
```

SEERGENERALIZATIONBUTTON.JAVA

```
package softengine.seerd.seerdgui;

import softengine.segui.seguifw.SEInterfaceItem;
import softengine.segui.sewidgets.SEDrawnButton;
import java.awt.Color;
import softengine.segui.sewidgets.SEPen;
import softengine.sebuttons.seumlbuttons.SEAssociationButton;
import softengine.sebasic.SECommand;

public class SEERGeneralizationButton extends SEDrawnButton
{
  public SEERGeneralizationButton(SEInterfaceItem par);
  public SEERGeneralizationButton(SEInterfaceItem par, SECommand comm);

  public void Draw();
}
```

SEERGENERALIZATIONENDVIEW.JAVA

```
package softengine.seerd.seerdgui;

import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.sewidgets.SECanvas;
import softengine.segui.secustomgui.*;
import java.awt.*;

public class SEERGeneralizationEndView extends SEAssociationEndView
{
  boolean              target;
  SEPoint              ep;

  private static final int  RD = 20;
  private static final int  EM = 7;

  public SEERGeneralizationEndView(SEGraphicItemView g, SEGlyph parent, boolean tar);

  public void Draw(SECanvas cv);

  protected boolean AllowedType(Object obj);
  protected SEPoint GetEndPoint();

  private SEPoint ComputePoint(int x1, int y1, int x3, int y3, int dist, float m);
  private void DrawEndSymbol(SECanvas cv,SEPen pen);
}
```

SEERGENERALIZATIONVIEW.JAVA

```
package softengine.seerd.seerdgui;

import softengine.segui.*;
import softengine.segui.secustomgui.*;
import java.awt.*;
import java.awt.event.*;

public class SEERGeneralizationView extends SEAssociationView
{
  public SEERGeneralizationView(SEGraphicItemView g, SEGlyph parent);
  protected boolean AllowedSource(SEGraphicItemView g);
  protected boolean AllowedTarget(SEGraphicItemView g);
  protected void CreateSource(SEGraphicItemView item);
  protected void CreateTarget(SEGraphicItemView item);
  protected void SourceException(SEGraphicItemView item) throws SENotAllowedAssocEnd;
  protected void TargetException(SEGraphicItemView item) throws SEDontAllowedAssocEnd;
}
```

SEERNAMEAREA.JAVA

```
package softengine.seerd.seerdgui;

import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.secustomgui.*;
import softengine.segui.seguifw.*;
import softengine.secommands.SEPropagateToChild;
import softengine.segui.secustomgui.SEGlyph;
import softengine.segui.sewidgets.SECanvas;
import java.awt.*;
import java.awt.Font;

public class SEERNameArea extends SEGlyph
{
  protected SETextLine     name;
  protected int            yoffset;

  public static final int   NOFFSET   = 5;
  public static final Font  NAMEFONT  = new Font("SansSerif", Font.BOLD, 10);

  public SEERNameArea(SEGlyph parent, SECanvas drawing);
  public SEERNameArea(int nyoffset, byte halign, Font f, SEGlyph parent, SECanvas drawing);
  public SEERNameArea(int nyoffset, String str, SEGlyph parent, SECanvas drawing);
  public SEERNameArea(int x, int y, int w, int h, SEGlyph parent, SECanvas drawing);
  public SEERNameArea(SERect r, SEGlyph parent, SECanvas drawing);

  public void DefineGlyph(SEGlyph parent, SECanvas drawing);

  protected boolean AllowedType(Object obj);
  protected void SetGlyphEvents();

  public void Draw(SECanvas cv);
  public static int FontMaxHeight(SECanvas cv);
}
```

SEERRELATIONSHIPBUTTON.JAVA

```
package softengine.seerd.seerdgui;

import softengine.segui.*;
import softengine.sebasic.SECommand;
import softengine.segui.seguifw.*;
import softengine.segui.secustomgui.*;
import softengine.segui.sewidgets.*;
import java.awt.*;
import java.awt.event.*;

public class SEERRelationshipButton extends SEDrawnButton
{
  public SEERRelationshipButton(SEInterfaceItem par);
  public SEERRelationshipButton(SEInterfaceItem par, SECommand comm);

  public void Draw();
}
```

SEERRELATIONSHIPVIEW.JAVA

```
package softengine.seerd.seerdgui;

import softengine.seerd.seerdme.*;
import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.secustomgui.*;
import softengine.segui.sewidgets.*;
import java.awt.*;

public class SEERRelationshipView extends SEERElementView
{
  public static final int XOFFSET = 0;
  public static final int YOFFSET = 0;

  public SEERRelationshipView(SEPoint pos, SEGlyph parent, SECanvas drawing);

  protected SEERRelationshipView(SEPoint pos, int width, int heigth, SEGlyph parent,
                                 SECanvas drawing);
  protected SEERRelationshipView(int x, int y, int width, int heigth, SEGlyph parent,
                                 SECanvas drawing);

  protected boolean AllowedType(Object obj);
  protected SERect GetNameRect(int h2, int dx);
  protected void SetName(SECanvas drawing);
  protected void UpdateWidth();

  public void Draw(SECanvas cv);
  public int GetChildXOffset();
}
```

SEERSETCREATEAGGREGATE.JAVA

```
package softengine.seerd.seerdgui;

import softengine.secommands.SESetCreateItem;
import softengine.segui.secustomgui.SEGraphicView;

public class SEERSetCreateAggregate extends SESetCreateItem
{
  public SEERSetCreateAggregate(SEGraphicView docview);
}
```

SEERSETCREATEASSOCIATION.JAVA

```
package softengine.seerd.seerdgui;

import softengine.secommands.SESetCreateItem;
import softengine.segui.secustomgui.SEGraphicView;

public class SEERSetCreateAssociation extends SESetCreateItem
{
   boolean    mandatory;
   boolean    reading;
   String     cardinality;

   public SEERSetCreateAssociation(SEGraphicView docview, boolean man, boolean read,
                                   String card);
}
```

SEERSETCREATEASSOCIATION.JAVA

```
package softengine.seerd.seerdgui;

import softengine.secommands.SESetCreateItem;
import softengine.segui.secustomgui.SEGraphicView;

public class SEERSetCreateEntity extends SESetCreateItem
{
  public SEERSetCreateEntity(SEGraphicView docview);
}
```

SEERSETCREATEGENERALIZATION.JAVA

```
package softengine.seerd.seerdgui;

import softengine.segui.*;
import softengine.segui.sewidgets.*;
import softengine.secommands.*;
import java.awt.*;
import java.awt.event.*;

public class SEERSetCreateGeneralization extends SESetCreateItem
{
  public SEERSetCreateGeneralization(SEGraphicView docview);
}
```

SEERSETCREATERELATIONSHIP.JAVA

```
package softengine.seerd.seerdgui;

import softengine.secommands.SESetCreateItem;
import softengine.segui.secustomgui.SEGraphicView;

public class SEERSetCreateRelationship extends SESetCreateItem
{
  public SEERSetCreateRelationship(SEGraphicView docview);
}
```

SEERSETCREATEWEAKENTITY.JAVA

```
package softengine.seerd.seerdgui;

import softengine.secommands.SESetCreateItem;
import softengine.segui.secustomgui.SEGraphicView;

public class SEERSetCreateWeakEntity extends SESetCreateItem
{
  public SEERSetCreateWeakEntity(SEGraphicView docview);
}
```

SEERSETCREATEWEAKRELATIONSHIP.JAVA

```
package softengine.seerd.seerdgui;

import softengine.secommands.SESetCreateItem;
import softengine.segui.secustomgui.SEGraphicView;

public class SEERSetCreateWeakRelationship extends SESetCreateItem
{
  public SEERSetCreateWeakRelationship(SEGraphicView docview);
}
```

SEERWEAKENTITYBUTTON.JAVA

```
package softengine.seerd.seerdgui;

import softengine.segui.*;
import softengine.segui.sewidgets.*;
import softengine.segui.seguifw.SEInterfaceItem;
import softengine.sebasic.SECommand;
import java.awt.*;
import java.awt.event.*;

public class SEERWeakEntityButton extends SEDrawnButton
{
  public SEERWeakEntityButton(SEInterfaceItem par);
  public SEERWeakEntityButton(SEInterfaceItem par, SECommand comm);

  public void Draw();
}
```

SEERWeakEntityView.java

```
package softengine.seerd.seerdgui;

import softengine.seerd.seerdme.SEERWeakEntity;
import softengine.segui.sewidgets.*;
import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.secustomgui.*;
import softengine.seerd.seerdme.SEERWeakEntity;
import java.awt.*;

public class SEERWeakEntityView extends SEEREntityView
{
  public static final int  XOFFSET   =  5;
  public static final int  YOFFSET   =  5;
  public static final int  INOFFSET  =  4;

  public SEERWeakEntityView(SEPoint pos, SEGlyph parent, SECanvas drawing);
  protected SEERWeakEntityView(SEPoint pos, int width, int height, SEGlyph parent,
                               SECanvas drawing);

  public void Draw(SECanvas cv);

  protected boolean AllowedType(Object obj);
  protected void SetName(SECanvas drawing);
}
```

SEERWeakEntityView1.java

```
package softengine.seerd.seerdgui;

import softengine.seerd.seerdme.*;
import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.sewidgets.*;
import softengine.segui.secustomgui.*;
import java.awt.*;

public class SEERWeakEntityView1 extends SEERWeakEntityView
{
   public static final int XOFFSET = 10;
   public static final int YOFFSET = 8;

  public SEERWeakEntityView1(SEPoint pos, SEGlyph parent, SECanvas drawing);
  public SEERWeakEntityView1(SEPoint pos, int width, int height, SEGlyph parent,
                             SECanvas drawing);

  public void Draw(SECanvas cv);
  public int GetChildYOffset();
  public int GetChildXOffset();
  public int GetSiblingOffset();
}
```

SEERWEAKENTITYVIEW2.JAVA

```
package softengine.seerd.seerdgui;

import softengine.seerd.seerdme.*;
import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.sewidgets.*;
import softengine.segui.secustomgui.*;
import java.awt.*;

public class SEERWeakEntityView2 extends SEERWeakEntityView1
{
  public SEERWeakEntityView2(SEPoint pos, SEGlyph parent, SECanvas drawing);
  protected SEERWeakEntityView2(SEPoint pos, int width, int height, SEGlyph parent,
                                SECanvas drawing);

  public void Draw(SECanvas cv);
}
```

SEERWEAKRELATIONSHIPBUTTON.JAVA

```
package softengine.seerd.seerdgui;

import softengine.segui.*;
import softengine.sebasic.SECommand;
import softengine.segui.seguifw.*;
import softengine.segui.secustomgui.*;
import softengine.segui.sewidgets.*;
import java.awt.*;
import java.awt.event.*;

public class SEERWeakRelationshipButton extends SEDrawnButton
{
  public SEERWeakRelationshipButton(SEInterfaceItem par);
  public SEERWeakRelationshipButton(SEInterfaceItem par, SECommand comm);
  public void Draw();
}
```

SEERWEAKRELATIONSHIPVIEW.JAVA

```
package softengine.seerd.seerdgui;

import softengine.seerd.seerdme.SEERRelationship;
import softengine.sebasic.*;
import softengine.segui.*;
import softengine.segui.secustomgui.*;
import softengine.segui.sewidgets.*;
import java.awt.*;

public class SEERWeakRelationshipView extends SEERRelationshipView
{
  public static final int   XOFFSET   = 8;
  public static final int   YOFFSET   = 4;

  public SEERWeakRelationshipView(SEPoint pos, SEGlyph parent, SECanvas drawing);
  protected SEERWeakRelationshipView(SEPoint pos, int width, int height, SEGlyph parent,
                                     SECanvas drawing);

  protected boolean AllowedType(Object obj);

  public void Draw(SECanvas cv);
  public int GetChildXOffset();
  public int GetChildYOffset();
}
```

B.3. PAQUETE softengine.seerd.seerdme

SEERAGGREGATE.JAVA

```
package softengine.seerd.seerdme;

import softengine.sebasic.*;
import java.awt.*;
import java.awt.event.*;

public class SEERAggregate extends SEEREntityItem
{
  SEERRelationship       rel;   // Contained relationship.

  public SEERAggregate();

  public void SetRelationship(SEERRelationship r);
}
```

SEERELEMENT.JAVA

```
package softengine.seerd.seerdme;

import softengine.sebasic.*;
import softengine.segui.*;
import softengine.seapp.SEDocumentItem;

public abstract class SEERElement extends SEDocumentItem
{
  SEString                  ename;       // Item name.
  SEERAttributeSequence     attributes;  // Attibutes.
  SEERKeySequence           keys;        // Keys.
  SEERKey                   pkey;        // Primary key.

  protected SEERElement();
  public void AddAttribute(SEERAttribute attr);
  public void AddKey(SEERKey key);
  public SEString GetName();
  public void SetName(SEString str);
  public void SetName(String str);
}
```

SEERENTITY.JAVA

```
package softengine.seerd.seerdme;

import softengine.sebasic.*;
import java.awt.*;
import java.awt.event.*;

public class SEEREntity extends SEEREntityItem
{
  public SEEREntity();
}
```

SEERENTITYITEM.JAVA

```
package softengine.seerd.seerdme;

import softengine.sebasic.*;
import softengine.segui.*;

public abstract class SEERElement
{
  SEString                  rolename;

  protected SEERElement();
  public SEString GetRolename();
  public void SetRolename(SEString role);
  public void SetRolename(String role);
}
```

SEERGENERALIZATION.JAVA

```
package softengine.seerd.seerdme;

import softengine.sebasic.*;
import softengine.segui.*;
import softengine.seapp.SEDocumentItem;
import java.awt.*;
import java.awt.event.*;

public class SEERGeneralization extends SEDocumentItem
{
   SEERElement           generic;       // Entidad generica.
   SEERElementSequence   spec;          // Secuencia de Entidades especializadas.

  public SEERGeneralization(SEERElement ent);
  public void AddSpecialized(SEERElement ent);
  public void SetGenericElement(SEERElement ent);
}
```

SEERMEMBER.JAVA

```
package softengine.seerd.seerdme;

import softengine.sebasic.*;
import softengine.segui.*;
import softengine.seapp.SEDocumentItem;

public abstract class SEERMember extends SEDocumentItem
{
  SEERElementItem     item;          // Element item asociado.
  boolean             reading;       // Marca de inicio de lectura.
  char                order;         // Orden de la interrelacion para el elemento asociado.
  SEString            rolename;      // Rol dentro de la interrelacion.

  protected SEERMember();
  public SEERElementItem GetItem();
  public boolean StartsReading();
  public char GetOrder();
  public SEString GetRolename();
  public void SetItem(SEERElementItem i);
  public void SetReading(boolean read);
  public void SetOrder(char o);
  public void SetRolename(SEString role);
  public void SetRolename(String role);
}
```

SEERRELATIONSHIP.JAVA

```
package softengine.seerd.seerdme;

import softengine.sebasic.*;
import java.awt.*;
import java.awt.event.*;

public class SEERRelationship extends SEERElement
{
  SEERMemberSequence       items; // Entidades o agregados que forman parte de la
                                  // interrelacion.

  public SEERRelationship();

  public void AddMember(SEERMember mem);
  public void RemoveMember(SEERMember mem);
}
```

SEERWEAKENTITY.JAVA

```
package softengine.seerd.seerdme;

import softengine.sebasic.*;
import java.awt.*;
import java.awt.event.*;

public class SEERWeakEntity extends SEEREntity
{
  SEERRelationship      identified_by;  // Associated relationship.

  public SEERWeakEntity();

  public void SetIdentifyingRel(SEERRelationship rel);
}
```

SEERWEAKRELATIONSHIP.JAVA

```
package softengine.seerd.seerdme;

import softengine.sebasic.*;
import java.awt.*;
import java.awt.event.*;

public class SEERWeakRelationship extends SEERRelationship
{
  public SEERWeakRelationship();
}
```

BIBLIOGRAFÍA

[ALAGIC] ALAGIC, S.

Relational database technology.

Springer-Verlag, EUA: 1986.

[ATZENI] ATZENI, P. ; DEANTONELLIS, V.

Relational database theory.

The Benjamin/Cummings Publishing, EUA:1993.

[BATINI] BATINI, C.;CERI, S.;NAVATHE, S.B.

Diseño conceptual de bases de datos. Un enfoque de Entidades – Interrelaciones.

Addison – Wesley / Díaz de Santos, EUA:1994.

[BEYNON] BEYNON-DAVIES, P.

Relational database systems: a programatic approach.

BLACKWELL Scientific Publications, Reino Unido:1991.

[BOOCH] BOOCH, G.

Object-Oriented Analysis and Design with applications.

The Benjamin/Cummings Publishing, EUA:1994.

[BRODIEa] BRODIE, M.L.

On the development of data models en *On conceptual modeling.*

[BRODIEb] BRODIE, M.L., et al.

On conceptual modeling.

Springer-Verlag, EUA: 1984.

[CAMPIO] CAMPIONE, M. y WALRATH, K.

The Java Tutorial Second Edition. Object oriented programming for the Internet.

Sun Microsystems, EUA: 2000.

http://java.sun.com/books/tutorial.html.

[CHEN] CHEN, P.P.

The entity-relationship model. Toward a unified view of data.

ACM Transactions on Database Systems, 1:9-3, 1976.

[CODD] CODD, E.

Is your DBMS really relational? y Does your DBMS Run by the Rules?

ComputerWorld, Oct. 14 y Oct. 21: 1985.

[DATE] DATE, C.J.

An introduction to database systems. Volume I.

Addison-Wesley, EUA:1995.

[FOWLER] FOWLER, M. ; SCOTT, K.

UML Distilled. Applying the Standard Object Modeling Language.

Addison Wesley Longman, EUA:1997.

[GAMMA] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J.

Design patterns. Elements of reusable object-oriented software.

Addison Wesley Publishing Company, EUA: 1995.

[HANSEN] HANSEN, G.W.; HANSEN, J.

Database management and design.

Prentice Hall, EUA:1992.

[HARES] HARES, J.S. ; SMART, J.D.

Object orientation: technology, techniques, management and migration.

John Wiley & Sons, Reino Unido:1993.

[HAWRYSZ] HAWRYSZKIEWYCZ, I.T.

Database analysis and design.

MacMillan Publishing Company, EUA:1984.

[JIA] JIA, XIAOPING

Object-Oriented Software Development using Java: principles, patterns and frameworks

Addison-Wesley Longman, EUA:2000.

[JUÁREZ] JUÁREZ, C.

Diseño de bases de datos relacionales y SQL.

Notas de curso, México: 1991.

[KORTH] KORTH, H.F. y SILBERSCHATZ, A.

Fundamentos de bases de datos.

McGraw-Hill Interamericana, España: 1993.

[KRUCHTEN] KRUCHTEN, P.

Rational Unified Process: Best Practices for Software Development Teams (White paper).

Rational Software Inc., EUA: 1998.

[LOUCOP] LOUCOPOLOS, Pericles y ZICARI, Roberto.

Conceptual modeling, databases and CASE: an integrated view of information systems development.

John Wiley & Sons, EUA: 1992.

[NAUGHTON] NAUGHTON, P.; SCHILDT, H.

Java 2: the complete reference

Osborne-McGraw Hill, EUA: 1999.

[OMG] OMG

OMG Unified Modeling Language Specification.

OMG, EUA:1999.

[PREE] PREE, W.

Design patterns for object oriented software development.

Addison Wesley Publishing / ACM Press, EUA:1995.

[ROB] ROB, P. y CORONEL, C.

Database systems. Design, implementation and management.

Course Technology, EUA:1995.

[ULLMANa] ULLMAN, J.D.

Principles of database and knowledge-base systems.

Computer Science Press, EUA: 1988.

[ULLMANb] ULLMAN, J. y WIDOM, J.

A First Course in Database Systems.

Prentice Hall, EUA:1997.

[WERTZ] WERTZ, C. J.

Relational Database Design. A practitioner's guide.

CRC Press, EUA:1993.

[WHITMIRE] WHITMIRE, S.A.

Object oriented design measurement.

John Wiley & Sons, EUA:1997.

[WINBLAD] WINBLAD, A.L.; EDWARDS, S.D.; KING, D.R.

Object Oriented Software.

Addison Wesley Publishing, EUA:1990.

GLOSARIO

A

actor Papel desempeñado por un objeto cuando puede operar sobre otros objetos pero ningún otro objeto opera sobre él. (Véase *servidor* y *agente*).
Papel que un usuario desempeña respecto a un sistema como parte de un caso de uso.

abstracción Proceso mental mediante el cual se exaltan las características y propiedades relevantes de un problema para su resolución o de un objeto para distinguirlo de los demás tipos de objetos.

agente Papel desempeñado por un objeto cuando puede operar sobre otros objetos y además otros objetos operan sobre él. (Véase *actor* y *servidor*).

agregación Mecanismo de abstracción mediante el cual se define una entidad nueva a partir de un conjunto de otras entidades que representan sus partes componentes.

Tipo de relación entre clases que indica contención de la clase componente por la clase agregada.

applet Programa en Java que se ejecuta dentro del contexto de un navegador de web que soporta el lenguaje.

aridad Métrica de una relación que se refiere a su número de atributos.

asociación simple Tipo de relación entre clases en la que se describe alguna dependencia semántica entre ellas.

atributo Propiedad que caracteriza a una relación.

AWT Interfaz de programación de aplicaciones de Java que proporciona las clases necesarias para la programación de interfaces gráficas de usuario.

B

base de datos — Colección de datos interrelacionados almacenados en una computadora de manera más o menos permanente.

base de datos relacional — Colección de relaciones. Base de datos basada en el modelo relacional.

bytecode — Código independiente de la plataforma resultado de la compilación del código fuente en Java y que será ejecutado por el intérprete de Java.

C

cardinalidad — Métrica de una relación que se refiere a su número de tuplos.
Número de entidades con las que se puede asociar una entidad a través de una interrelación.

caso de uso — Interacción típica entre un sistema y sus usuarios o aquellos sistemas externos con los que se relaciona.

CL — Lenguaje de control. Permite la administración de usuarios y el control de niveles de acceso.

clase — Conjunto de objetos que comparten una estructura y un comportamiento común.

clasificación — Mecanismo de abstracción mediante el cual se define un concepto como un tipo de entidades del mundo real, caracterizadas por propiedades comunes.

comportamiento — Forma en que reacciona un objeto al interactuar con otros y manera en que cambia su estado.

concurrencia — Acceso simultáneo a un objeto.

conjunto de entidades Colección de entidades del mismo tipo.

conjunto de interrelaciones Colección de interrelaciones del mismo tipo.

constructor Tipo de método que crea un objeto y/o inicializa su estado.

D

DBMS Sistema Manejador de Bases de Datos. Software que se utiliza para la administración y el acceso a los datos de una base de datos.

DDL Lenguaje de manipulación de datos, permite definir el esquema conceptual de una base de datos.

dependencia Tipo de relación entre clases en la que un cambio en una clase obliga a un cambio en la otra.

dependencia funcional Relación entre dos subconjuntos de atributos de una relación. Véase capítulo 1, Diseño de bases de datos relacionales.

desnormalización Proceso inverso de la normalización.

destructor Tipo de método que libera el estado de un objeto y/o lo destruye.

diagrama de interacción Tipo de diagrama en UML que describe la colaboración entre grupos de objetos en cierto comportamiento de un sistema.

diagrama de secuencia Tipo de diagrama de interacción que representa un conjunto de mensajes ordenados intercambiados entre objetos para lograr un comportamiento determinado.

DML Lenguaje de manipulación de datos. Contiene instrucciones para recuperar, insertar, eliminar y actualizar los datos de una base de datos.

dominio Conjunto de valores que puede asumir un atributo.

E

encapsulamiento Propiedad que oculta los detalles de implantación de un objeto a los demás, los cuales sólo pueden acceder a los datos de ese objeto mediante una interfaz bien definida.

enlace Conexión física o conceptual entre objetos a través de la cual un objeto cliente utiliza los servicios de otro objeto servidor. Establece una vía para el paso de mensajes entre los objetos enlazados.

entidad Objeto que existe en el mundo real y es identificable de manera única.

entidad fuerte Entidad que tiene los atributos necesarios para integrar su llave principal.

entidad débil Entidad que se identifica a través de su interrelación con otra entidad.

esquema Definición de la estructura de una base de datos.

esquema conceptual Representación global de la base de datos.

esquema externo Esquema de una base de datos percibido por el usuario.

esquema interno Representación de bajo nivel de una base de datos, formado por las estructuras de almacenamiento.

esquema relacional Descripción de la estructura de las relaciones que forman una base de datos relacional.

estado Conjunto de propiedades de un objeto y sus valores reales en un momento determinado.

excepción Situación que ocurre durante la ejecución de un programa y que origina que el flujo de control se desvíe de su curso normal.

extensión Tipo de asociación entre casos de uso en la que un caso de uso amplía o extiende el comportamiento definido en el otro.

F

forma normal Regla que debe cumplir una relación para asegurar que cada entidad de un sistema esté representada por medio de una relación separada.

G

generalización-especialización Mecanismo de abstracción que define una relación de subconjunto entre entidades de dos o más tipos.

grado Métrica de una relación que se refiere a su número de atributos.

H

herencia Relación entre clases en la que una clase llamada *subclase* comparte la estructura y/o el comportamiento definidos en otra clase llamada *superclase*.

I

identidad Propiedad de un objeto de ser distinguido de los demás.

implantación Parte de una clase que representa su visión interna, englobando su comportamiento; está formada por la implantación de sus métodos.

ingeniería inversa Proceso mediante el cual se produce un modelo lógico a partir de código ejecutable.

integridad Consistencia de una base de datos.

interfaz Parte de una clase que representa la visión externa que los objetos de una clase tienen de los objetos de otra, ocultando su estructura y detalles de implantación.
Protocolo de comportamiento que un objeto de una clase puede llevar a cabo. No tiene implantación, atributos, estados ni asociaciones, sólo operaciones.

interrelación Asociación entre dos o más entidades.

interrelación débil Interrelación en la que participa al menos una entidad débil.

instancia En el contexto de bases de datos, conjunto de valores almacenados en la base de datos en un momento determinado.

iterador Tipo de método que permite acceder a los componentes de un objeto en un orden predeterminado.

J

Java API Interfaz de programación de aplicaciones de Java. Especificación de las clases de Java.

jerarquía Forma de clasificación de abstracciones.

JVM Máquina Virtual de Java. Parte del ambiente de ejecución que se encarga de la interpretación de los bytecodes y tiene una implantación para cada plataforma subyacente.

L

llave Conjunto de atributos que identifica de manera única a cada tuplo de una relación.

llave candidata Subconjunto del conjunto de atributos de una relación del que no hay dos tuplos que tengan un mismo valor y para el que no hay un subconjunto propio que cumpla con la condición anterior.

llave foránea Subconjunto del conjunto de atributos (FK) de una relación R2 tal que existe una relación R1 con una llave candidata (CK) para la cual cada valor de FK es idéntico al valor de CK en algún tuplo de R1.

llave principal Llave candidata elegida para ser el identificador único de una relación.

M

modelo Representación de la realidad que destaca los aspectos importantes para el problema a resolver.

modelo de datos Colección de conceptos bien definidos que permiten representar las características de una aplicación determinada.

modificador Tipo de método que altera el estado de un objeto.

modularidad Propiedad del software según la cual los sistemas están formados por un conjunto de módulos cohesivos y débilmente acoplados.

multihilo Capacidad de poder iniciar más de un flujo de ejecución dentro de un programa.

O

objeto Cosa tangible o visible sobre la cual se realiza una acción o algo que puede ser comprendido intelectualmente. Instancia de una clase.

objeto agregado Objeto que contiene a otro u otros, ya sea por valor o por referencia.

P

package Nivel de visibilidad en el que los objetos de una clase son visibles sólo para los objetos de las clases definidas dentro del mismo paquete.

papel Parte o actividad que desempeña una entidad en una interrelación.

paquete Unidad lógica de alto nivel que agrupa elementos de un modelo con características similares.

patrón de diseño Plantilla de objetos con responsabilidades e interacciones estereotípicas.

pérdida de información Obtención de información falsa al realizar operaciones sobre la base de datos.

persistencia Propiedad de un objeto por la que su existencia trasciende el tiempo y/o el espacio.

polimorfismo En teoría de tipos, habilidad de una variable de asumirse como de diferentes tipos, dependiendo de su contenido en un momento determinado. Posibilidad de utilizar un objeto de una subclase como si fuera un objeto de su superclase.

portabilidad Habilidad del software de ser ejecutado en diversas plataformas de hardware.

private Nivel de visibilidad en el que los objetos de una clase sólo son accesibles para objetos de la misma clase.

protected Nivel de visibilidad en el que los objetos de la clase sólo son accesibles por objetos de la misma clase y sus subclases.

public Nivel de visibilidad en el que los objetos de la clase son visibles para los de cualquier otra.

Q

QL Lenguaje de consulta. Parte del DML que permite recuperar información de una base de datos a través de consultas.

R

redundancia En el contexto de bases de datos, almacenamiento de la misma información en varios lugares.

relación Conjunto de tuplos o n-adas.

robustez

S

secuencia Objeto agregado compuesto por objetos del mismo tipo que requieren ser recorridos de manera secuencial.

seguridad En el contexto de bases de datos, es el control de creación de usuarios y niveles de acceso.

selector Tipo de método que accede al estado de un objeto, pero no lo altera.

servidor Papel desempeñado por un objeto cuando no opera sobre otros objetos, sólo otros objetos operan sobre él. (Véase *actor* y *agente*).

subclase Clase que hereda la estructura y/o comportamiento de otra(s) clase(s) llamada(s) *superclase(s)*.

superclase Clase de la cual otra(s) clase(s) llamadas subclases hereda(n) la estructura y/o comportamiento.

T

tipificación Elemento fundamental del paradigma orientado a objetos que define que las clases representan tipos de datos.

tuplo Conjunto de pares atributo-valor.

U

UML Lenguaje de Modelado Unificado. Lenguaje gráfico útil para especificar, visualizar, construir y documentar componentes de un sistema de software, al igual que para elaborar modelos de negocios y de otros sistemas no computacionales.

UNICODE Estándar para la representación de caracteres como enteros que utiliza 16 bits, es decir, puede representar más de 65,000 caracteres únicos.

uso Tipo de asociación entre casos de uso en la que uno de ellos utiliza un comportamiento definido en el otro.

V

variable de instancia Variable definida en una clase cuyos valores son accesibles dentro de una sola de sus instancias.

variable de clase Variable definida en una clase cuyos valores son accesibles dentro de todas sus instancias.