ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Ingeniería Técnica en Informática de Sistemas

Aplicación Web para el Diseño de Bases de Datos usando Esquemas EER y Traducción Automática a SQL

Realizado por

José David Quero Sánchez

Dirigido por

José Galindo Gómez

Departamento

Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, Julio 2004

UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Ingeniería Técnica en Informática de Sistemas

Reunido el tribunal examinador en el día de la fecha, constituido por:

Presidente Dº/Dª. _____

Secretario Dº/Dª. _____

Vocal Dº/Dª. _____

para juzgar el proyecto Fin de Carrera titulado:

_____

_____

del alumno Dº/Dª. _____

dirigido por Dº/Dª. _____

ACORDÓ POR _____ OTORGAR LA CALIFICACIÓN DE _____

Y PARA QUE CONSTE, SE EXTIENDE FIRMADA POR LOS COMPARECIENTES DEL TRIBUNAL, LA PRESENTE DILIGENCIA.

Málaga, a de del 200__

El Presidente El Secretario El Vocal

Fdo: Fdo: Fdo:

Índice

Capítulo 1

Introducción

En esta primera sección vamos a introducir muy brevemente un resumen de los aspectos principales de este proyecto. En primer lugar, mostraremos las motivaciones que nos han llevado a realizar este trabajo; en segundo lugar, explicaremos cuáles son los objetivos que se han establecido sobre él; por último, comentaremos las secciones en las que está dividida esta memoria, junto a una descripción breve de su contenido.

1.1 Motivación

En la sociedad de la información en la que estamos inmersos, muchas instituciones, empresas, gobiernos, etc., requieren almacenar grandes cantidades de información. Un requisito fundamental de lo anterior, es que debe hacerse de forma simple y ordenada, con el objetivo de poder acceder fácilmente a la información allí almacenada. Otro de los objetivos es poder modificar y estructurar convenientemente todos los datos ya incluidos. Por ello, la utilización de bases de datos potentes y eficientes se hace indispensable, así como facilitar la tarea del diseñador a la hora de crearlas. Un modelo de datos es un conjunto de conceptos que pueden servir para describir la estructura de una base de datos. De la necesidad de facilitar esta tarea, surge esta aplicación, gracias a la cual se puede modelar la base de datos de una forma más simple. Los modelos de datos de alto nivel o conceptuales disponen de conceptos muy cercanos al modo como la generalidad de usuarios percibe los datos, mientras que los modelos de datos de bajo nivel o físicos proporcionan conceptos que describen los detalles de cómo se almacenan los datos. El modelo entidad-relación (en adelante, modelo ER) es uno de los modelos conceptuales de alto nivel más utilizados. Fue introducido por Peter Chen en 1976 [CHE76] y se ha visto ampliado con el modelo EER (del inglés, Enhanced Entity-Relationship), que es el utilizado por Derytas para modelar las bases de datos. Además de permitir construir diagramas para el modelo EER, Derytas contiene una implementación del algoritmo que permite traducir un modelo EER en un esquema de una base de datos relacional, de manera que proporciona de forma automática las sentencias, en lenguaje SQL, para crear físicamente la base de datos.

1.2 Objetivos

El objetivo del proyecto es desarrollar una aplicación que nos permita dibujar esquemas EER para modelar una base de datos de forma sencilla e intuitiva. A partir de este esquema se obtiene el código para la creación de esta base de datos, escrito en SQL estándar, aunque teniendo en cuenta las restricciones de Oracle por ser uno de los SGBD (Sistema Gestor de Bases de Datos) más utilizados. El esquema se puede guardar en un fichero de texto para, posteriormente, poder leerlo del mismo y poder seguir trabajando con ese esquema. Otro de los objetivos que se pretenden conseguir con este proyecto es obtener una aplicación universal, de manera que pueda ser utilizada en cualquier plataforma, independientemente del hardware o del sistema operativo. Este es el motivo por el que se ha decidido desarrollar una aplicación *Web*, para que el único requisito sea tener un ordenador con conexión a Internet.

1.3 Organización de la Memoria

A continuación se describe brevemente la organización y el contenido de esta memoria.

Capítulo 2. Herramientas Utilizadas

Aquí se exponen los lenguajes de programación utilizados para la programación de la aplicación. La elección de los mismos se ha realizando basándonos en la ventajas que aporta cada uno según las necesidades de funcionamiento de la aplicación. Así, gracias a HTML, cualquier persona con conexión a *Internet* puede acceder a Derytas. El *applet* de Java proporciona una "pizarra" donde poder dibujar y, gracias a JavaScript se puede establecer la comunicación entre la página HTML y el *applet*.

Capítulo 3. Bases de Datos

En este capítulo se introduce al lector en los conceptos de las bases de datos, desde la definición de las mismas, hasta la generación del código para crearlas, pasando por cómo se pueden modelar, extendiendo de esta manera las nociones sobre modelado de bases de datos que se han expuesto en la introducción.

Capítulo 4. Derytas: Programación de la Aplicación

Capítulo en el que se comenta todo lo relacionado con el desarrollo de la aplicación, como las normas de programación adoptadas y la organización del código. También se distingue la programación según la herramienta utilizada, destacando las clases más importantes y los métodos más relevantes.

Capítulo 5. Derytas: Manual de Usuario

Este capítulo contiene toda la información necesaria para poder utilizar la aplicación y cumplir con sus dos objetivos principales: la creación de esquemas EER y la generación del *script* para crear la base de datos que representa dicho esquema.

Conclusiones y Líneas Futuras

En este apartado se exponen las conclusiones acerca del proyecto y las posibles expansiones y modificaciones del mismo.

Referencias

Aquí se indican los libros que se han consultado y las direcciones de las páginas *Web* que se han visitado para la realización de este proyecto.

Capítulo 2

Herramientas Utilizadas

En este capítulo se muestran los lenguajes empleados para desarrollar Derytas, comenzando por una visión general, pasando por sus orígenes, bases, evolución y finalizando en el uso de los mismos.

2.1 El Lenguaje HTML

HTML (HyperText Markup Language) es el lenguaje utilizado para crear documentos de hipertexto, un lenguaje de marcas mediante el cual se le dice a un explorador de Internet como mostrar una página *Web* [W3ROM]. Los ficheros HTML describen la distribución y estilo de cada uno de los elementos de una página *Web*, a base de combinar el texto de la página con los diferentes comandos del lenguaje. El resultado de la presentación de un documento HTML es muy similar a lo que se puede conseguir con un procesador de textos, sin embargo, el código HTML puede ser interpretado y visualizado en una gran variedad de entornos.

El marco de trabajo de su invención y creación han sido atribuidos a Tim Berners-Lee, un programador del Centro Europeo para Física de Partículas (CERN). Berners-Lee desarrolló el HTML para:

- Ofrecer un medio que permitiera a los científicos publicar, buscar y recibir información 24 horas al día.

- Crear un lenguaje internacional de codificación en ordenadores que facilitara el acceso universal independientemente de la plataforma, red, o terminal.

Los ficheros HTML contienen texto plano y pueden ser editados con cualquier aplicación sencilla que exporte texto sin formato, aunque existen editores especializados que simplifican esta tarea, automatizando determinadas labores de la creación de documentos de hipertexto.

Básicamente, el lenguaje HTML consta de una serie de órdenes o directivas, que indican al visor que estemos utilizando, la forma de representar los elementos (texto, gráficos, etc.) que contenga el documento.

Las directivas de HTML pueden ser de dos tipos, cerradas o abiertas. Las directivas cerradas son aquellas que tienen una palabra clave que indica el principio de la directiva y otra que indica el final. Entre la directiva inicial y la final se pueden encontrar otras directivas. Las directivas abiertas constan de una sola palabra clave. Para diferenciar las directivas del resto del texto del documento se encierran entre los símbolos < y >. Las directivas cerradas incluyen el carácter / antes de la palabra clave para indicar el final de la misma. Una directiva puede contener "parámetros" u opciones que modifican su comportamiento por defecto. Estos parámetros se indican a continuación de la palabra clave de la directiva. En la Tabla 2.1 se exponen algunos ejemplos y su utilidad.

Tipo de Directiva	Formato	Utilidad
Directiva abierta	** **	Fuerza un salto de línea
Directiva abierta con parámetros	**<HR WIDTH="60%">**	Dibuja una línea horizontal, con el tamaño especificado en porcentaje
Directiva cerrada	**<CENTER>** Texto centrado **</CENTER>**	Centra el texto
Directiva cerrada con parámetros	**<BODY bgcolor="#FF00FF"> </BODY>**	Establece el color de fondo

Tabla 2.1: Ejemplos de directivas de HTML

El lenguaje HTML ha sido diseñado teniendo presentes dos normas básicas:

1. Define la estructura y componentes de un documento, no la forma concreta en que estos componentes se presentan cuando un cliente los visualiza.

2. No está atado a ningún entorno particular. El contenido de un documento HTML puede ser interpretado y visualizado en ordenadores con características muy diferentes.

Estas dos condiciones son muy importantes, ya que permiten que una determinada información pueda ser vista por usuarios diferentes, con independencia de las capacidades de su entorno de trabajo. Un documento HTML sólo contiene la especificación de los elementos

que lo componen, y es responsabilidad de cada cliente el mostrar esta información de la manera más adecuada, en función de sus capacidades y las características del entorno.

Se puede decir que HTML es un lenguaje interpretado, ya que son los *browsers* o navegadores los encargados de procesar y representar su contenido. Un *browser* tiene mucha libertad para ajustar la presentación de un documento HTML en función de los recursos disponibles.

En general, un *browser* no respeta nada del posible formato que contenga el documento HTML que interpreta, por lo que se ignoran los saltos de línea o los espacios múltiples entre palabras. Si se desea incluir un salto de línea, se debe indicar explícitamente a través de su correspondiente etiqueta; por tanto, la inclusión de espacios adicionales entre líneas de código HTML ayuda a mejorar la legibilidad del documento, pero no afecta a la forma en que éste se representa.

La estructura general de un documento escrito en HTML contendría básicamente las directivas que se exponen en la Tabla 2.2, en la que se indica el orden en el que pueden ponerse [W3MAR].

<HTML>	Indica el inicio del documento.
<HEAD>	Inicio de la cabecera.
<TITLE>	Inicio del título del documento.
</TITLE>	Final del título del documento.
</HEAD>	Final de la cabecera del documento.
<BODY>	Inicio del cuerpo del documento.
</BODY>	Final del cuerpo del documento.
</HTML>	Final del documento.

Tabla 2.1: Estructura general de un documento HTML

Aparte de estas directivas básicas existen muchas otras mediante las cuales podemos definir desde el tipo de letra, hasta diseñar una tabla o insertar una imagen. Todas esas directivas y su utilización las podemos encontrar en cualquier manual de HTML.

2.1.1 La Evolución de HTML

Desde su creación, el lenguaje HTML ha evolucionado rápidamente, impulsado por la necesidad de ampliar sus capacidades y adaptarse a los nuevos requerimientos de sus usuarios. Este cambio ha sido impulsado por organismos internacionales, como el *World Wide Web Consortium* (http://www.w3.org) o por empresas como Netscape y Microsoft.

La versión inicial se denominó **HTML/1.0**, y supuso un gran avance como sistema mundialmente aceptado de presentación de texto con formato. Sin embargo, pronto se descubrieron sus limitaciones. La introducción de los entornos gráficos de acceso al Web reclamaba mayores capacidades de formato que no fueron contempladas en esta especificación inicial.

Por ello, algunas empresas, principalmente Netscape, añadieron nuevos elementos y atributos al lenguaje HTML, que contribuyeron en buena medida al posterior éxito del Web: tablas, formularios, imágenes flotantes, mapas, etc. La siguiente estandarización, que incorporó parte de esas modificaciones, se denominó **HTML 2.0**; trataba de poner un poco de orden en el caos que introdujeron las aportaciones individuales de cada empresa, a menudo incompatibles entre sí.

Existen numerosos comités de estandarización y grupos de trabajo, integrados por organizaciones internacionales y fabricantes de software, tendentes a conseguir un futuro modelo único de HTML, compatible con todos los *browsers.*

A la hora de desarrollar páginas HTML, se debe tener presente el tipo de *browsers* que utilizarán los potenciales usuarios de un servicio de información. En la actualidad, la mayor parte de los usuarios del Web emplean clientes gráficos, que cada vez son más compatibles en sus capacidades.

La rápida evolución de HTML, y las incorporaciones que ha recibido, principalmente por parte de Netscape y Microsoft, ha roto con la filosofía original con que se diseñó, que buscaba un **lenguaje universal** de descripción de documentos. Cada vez es más común encontrarse con páginas cuya visualización depende de las características concretas de un tipo de *browser*,

que dificultan o hacen imposible el acceso para los usuarios que no puedan (o quieran) utilizarlo. Detrás de estas versiones especiales de HTML se suelen esconder las estrategias comerciales de empresas, que tratar de obtener una mayor cuota de usuarios en el lucrativo negocio de las redes de telecomunicaciones.

Sin embargo, se debe reconocer que las ideas propuestas por los fabricantes de clientes Web han 'acelerado' la evolución del Web e Internet, presentando propuestas novedosas que, en general, mejoran los servicios que recibe el usuario.

2.2 El Lenguaje JavaScript

JavaScript es el siguiente paso, después del HTML, que puede dar un programador de la *Web* que decida mejorar sus páginas y la potencia de sus proyectos [W3ALJ].

JavaScript es un lenguaje de programación interpretado, bastante sencillo y pensado para hacer las cosas con rapidez. Se utiliza para crear pequeños programas encargados de realizar acciones dentro del ámbito de una página *Web* en el ordenador del visitante de nuestras páginas (algo muy interesante, porque los servidores *Web* suelen estar sobrecargados). Con JavaScript podemos crear efectos especiales en las páginas y definir interactividades con el usuario. El navegador del cliente es el encargado de interpretar las instrucciones JavaScript y ejecutarlas, de modo que el mayor recurso, y tal vez el único, con que cuenta este lenguaje es el propio navegador. JavaScript es un lenguaje con muchas posibilidades, ya que también permite la posibilidad de programas más grandes, orientados a objetos, con funciones, estructuras de datos complejas, etc.

JavaScript proporciona los medios para:

- Controlar las ventanas del navegador y el contenido que muestran.

- Programar páginas dinámicas simples. Hasta la aparición de JavaScript, una vez cargada una página HTML, ésta no se podía cambiar. JavaScript permite, entre otras cosas, la interacción de la página con el ususario.

- Evitar depender del servidor Web para cálculos sencillos.

- Capturar los eventos generados por el usuario y responder a ellos sin salir a Internet.

- Simular el comportamiento de las macros *CGI* [W3ALV, W3CAR] cuando no es posible usarlas. CGI son las siglas de Common Gateway Interface, una especificación que va a realizar la función de interfaz o pasarela entre el servidor web y los programas, llamados programas CGI, haciendo uso del protocolo http y el lenguaje HTML. Los CGI se escriben habitualmente en el lenguaje Perl, pero pueden ser escritos en otros lenguajes, como C, C++ o Visual Basic.

- Comprobar los datos que el usuario introduce en un formulario antes de enviarlos.

- Comunicarse con el usuario mediante diversos métodos, como `alert`, que muestra una ventana con información para el usuario o `prompt`, que muestra una ventana en la que el usuario puede introducir datos.

La característica de JavaScript que más simplifica la programación es que, aunque el lenguaje soporta cuatro tipos de datos, no es necesario declarar el tipo de las variables, argumentos de funciones ni valores de retorno de las funciones. El tipo de las variables cambia implícitamente cuando es necesario, lo que dificulta el desarrollo de programas complejos, pero ayuda a programar con rapidez macros sencillas. En esto, JavaScript se separa totalmente de lenguajes como C, C++ o Java.

2.2.1 Orígenes de JavaScript

JavaScript ha sido inventado por Netscape, después de hacer sus navegadores compatibles con los *applets*. Comenzó a desarrollar un lenguaje de programación que permitiese crear pequeños programas en las páginas y que fuese mucho más sencillo de utilizar que Java. El nombre original de JavaScript fue LiveScript, pero se modificó para aprovechar el tirón de Java. Al ser código totalmente interpretado, JavaScript es más lento que Java (aunque Java

también es interpretado, pasa una compilación previa a su ejecución), pero en la práctica no suele ser un factor de importancia.

Netscape 2.0 fue el primer navegador que entendía JavaScript y su estela fue seguida por los navegadores de la compañía Microsoft a partir de la versión 3.0.

Obviamente, el objetivo de Netscape al introducir JavaScript es tratar de establecer un estándar de programación de macros ejecutables en el navegador web, que de ser adoptado por los webmasters (diseñadores de páginas web), facilitaría la implantación de los navegadores de Netscape en el mercado. En respuesta a este reto, MicroSoft soporta una versión parcial de JavaScript, con el nombre de JScript, en su *Internet Explorer*. El primer inconveniente de este estado de cosas es que las macros JavaScript sólo se ejecutan con normalidad en navegadores Netscape, por lo que el Webmaster es responsable de configurar la página para que pueda verse decentemente en un navegador que no sea Netscape.

Una solución sería utilizar en nuestras macros el subconjunto de funciones comunes a JavaScript y JScript, para soportar los navegadores Netscape y MicroSoft, pero esta solución nos obligaría a renunciar a muchas de las características del lenguaje.

2.2.2 Las Bases del Lenguaje JavaScript

El lenguaje JavaScript se inserta en documentos HTML, de forma que su código queda reflejado en la propia página y no es llamado o cargado de ninguna fuente externa (por ejemplo un archivo). Se trata de un lenguaje interpretado puro (ni compilación, ni generación de intermedios codificados de ningún tipo) y sensible a mayúsculas, aunque algunas implementaciones ignoran en parte este último extremo.

Hemos de establecer que **JavaScript no tiene nada que ver con Java** [W3UNAV], salvo en sus orígenes, aunque la herencia de ambos de C++ hace que la traslación de conocimiento y modo de trabajo sea muy directa. Actualmente son productos totalmente distintos y no guardan entre si más relación que la sintaxis muy parecida y poco más. Algunas diferencias entre estos dos lenguajes son las siguientes:

- **Compilador**. Para programar en Java necesitamos un compilador. Sin embargo, JavaScript no es un lenguaje que necesite que sus programas se compilen, sino que éstos se interpretan por parte del navegador cuando éste lee la página.

- **Orientado a objetos**. Java es un lenguaje de programación orientado a objetos, mientras que JavaScript no lo es.

- **Propósito**. Java es mucho más potente que Javascript, esto es debido a que Java es un lenguaje de propósito general, con el que se pueden hacer aplicaciones de lo más variado, sin embargo, con JavaScript sólo podemos escribir programas para que se ejecuten en páginas web.

- **Estructuras fuertes**. Java es un lenguaje de programación fuertemente tipado. Esto quiere decir que al declarar una variable tendremos que indicar su tipo y se prohíben operaciones entre variables de tipos distintos. Por su parte JavaScript no tiene esta característica, y podemos meter en una variable la información que deseemos, independientemente del tipo de ésta.

- **Otras características.** Como vemos Java es mucho más complejo, aunque también más potente, robusto y seguro. Tiene más funcionalidades que JavaScript y las diferencias que los separan son lo suficientemente importantes como para distinguirlo fácilmente.

2.2.3 Normas del Código en JavaScript

Las normas para poder escribir cualquier código de JavaScript se basan en 5 puntos básicos y que debemos cumplir siempre [W3AME]. Estas normas son las siguientes:

1. Todo el código está dentro de funciones

2. Todo el código de JavaScript se desarrolla en la página HTML según el formato:

```
<SCRIPT Language="JavaScript">
<!—
// Aquí irá su código
-->
</SCRIPT>
```

3. Las etiquetas <SCRIPT> y </SCRIPT> deben colocarse entre las etiquetas <HEAD> y </HEAD>

4. Las etiquetas <TITLE> no pueden estar colocadas entre las de <SCRIPT>

5. La llamada a la función se hace a través de un evento de un elemento del documento

Una observación a tener en cuenta es que el *tag* (etiqueta) utilizado es **<SCRIPT LANGUAGE=...>...</SCRIPT>**. En LANGUAGE se indica qué lenguaje *script* se utilizará (JavaScript, VBScript, …). Aquí indicamos JavaScript, ya que es lo que nos ocupa en estos momentos. También es interesante observar las acotaciones de comentario <!-- al comienzo y --> al final del área de escritura de código. Esto es para ocultarlo a los navegadores más antiguos, que no dan soporte a lenguajes *script*.

Si bien el código JavaScript puede incluirse en cualquier lugar de una página HTML, el modo de comportarse puede ser diferente. Lo habitual es hacerlo antes del cuerpo del documento, es decir, antes de la etiqueta <BODY>.

2.3 El Lenguaje Java

En el presente apartado se expone el lenguaje Java, caracterizado principalmente por su seguridad, sencillez e independencia de la plataforma [CUE00], siendo el lenguaje en el que ha sido programado la mayor parte de la aplicación.

2.3.1 Orígenes de Java

Java tiene su origen en 1991 [W3UPM], cuando un grupo de investigadores de Sun Microsystems, liderado por James Gosling, trabajaba en técnicas para producir software capaz de ser incluido dentro de cualquier aparato electrónico (teléfonos, faxes, videos y electrodomésticos en general) para proveerlos de "inteligencia". La reducida potencia de cálculo y memoria de estos aparatos llevó a desarrollar un lenguaje sencillo capaz de generar código de tamaño muy reducido. En principio, consideraron la posibilidad de utilizar lenguajes ya existentes como C++ o Smalltalk, pero pronto se hizo patente la necesidad de definir una máquina hipotética o virtual capaz de garantizar la portabilidad de las aplicaciones

debido a la existencia de distintos tipos de CPU [W3UPM] y a los distintos cambios, a la vez que cumpliera los requisitos de seguridad derivados de su uso en dispositivos de uso común. Por tanto, Java es el lenguaje de programación y, además, la definición de la máquina virtual (Java Virtual Machine) encargada de ejecutar las aplicaciones.

A pesar de los esfuerzos realizados por sus creadores, ninguna empresa de electrodomésticos se interesó por el nuevo lenguaje, por lo que iba a caer en desuso dado que los directivos de Sun no veían un mercado potencial y el grupo fue disuelto. Sin embargo, Gosling intuyó las posibilidades del lenguaje para proporcionar contenidos activos en las páginas Web y se embarcó en la construcción de un *browser* (navegador) basado en Java, HotJava, que constituyó la demostración general de las capacidades del lenguaje, comenzando las historia de Java como lenguaje íntimamente ligado a internet.

Como lenguaje de programación se presentó a finales de 1995 y desde entonces ha sido uno de los temas que más interés ha despertado en el mundo de la informática, mereciendo, incluso, la atención de publicaciones no especializadas. La clave fue la incorporación de un intérprete Java en el programa Netscape Navigator, versión 2.0, produciendo una verdadera revolución en internet. La mayor parte de las aplicaciones Java se utilizaron para dotar de contenido dinámico e interactivo a las páginas del World Wide Web, mediante los llamados *applets*. Posteriormente el uso de Java se fue extendiendo a un gran número de sistemas y aplicaciones, en las que el modelo ofrecido por Java de un entorno distribuido y completamente transportable entre plataformas es enormemente atractivo.

Java 1.1 apareció a principios de 1997, mejorando sustancialmente la primera versión del lenguaje, encontrándose en el momento de escribir este documento por la versión 1.4.1.03

2.3.2 Principales Características de Java

Java es un lenguaje de programación orientado a objetos, con una sintaxis similar a C o C++, pero ofreciendo una mayor simplicidad y robustez en el ciclo de desarrollo: las construcciones y características más complicadas de C y C++ han sido eliminadas y el lenguaje contiene mecanismos implícitos para garantizar la seguridad de las aplicaciones construidas con él.

También incorpora dos mecanismos a la hora de escribir programas simples, potentes y robustos: un tratamiento interno de multitarea y un sistema de excepciones que normaliza el procesado de errores por parte del programador.

La principal característica de Java es que es independiente de la plataforma, pudiendo ejecutarlo sobre distintas arquitecturas y sistemas operativos sin que sea necesario modificar el código del programa. Esta independencia se logra debido a que el lenguaje está soportado por dos elementos fundamentales: el compilador y la máquina virtual. El compilador traduce los programas a un formato especial llamado *bytecodes*, que es el formato que se le pasa a la máquina virtual. Tanto el compilador Java como la máquina virtual son específicos para cada plataforma, por lo que para poder ejecutar un programa Java en una determinada plataforma debe existir previamente una máquina virtual para ella, por lo que Sun Microsystems dispone de un entorno de ejecución para la mayoría de las plataformas.

Otra ventaja es que cuenta con un gran número de clases preexistentes que no deja de aumentar, presentando una gran riqueza en cuanto al tipo de funciones que permiten realizar. En la Tabla 2.3 se exponen algunos ejemplos.

Clase	Utilidad
Math	Proporciona métodos para realizar las operaciones matemáticas más habituales
Date	Se utiliza para manejar fechas y horas
Integer	Tiene como miembro una variable de tipo entero, y proporciona los métodos para trabajar con ella
String	Proporciona a Java la capacidad para el manejo de cadenas de caracteres
Vector	Representa un array de objetos que puede crecer y reducirse. Además, permite acceder a los elementos con un índice
Frame	Es una ventana de la que pueden depender otras ventanas y que puede tener una barra de menús
FileDialog	Muestra una ventana de diálogo en la que se puede seleccionar un fichero según el modo de apertura del fichero (para lectura o para escritura)

Tabla 2.2: Ejemplos de clases de Java

Sun describe el lenguaje Java como *"simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, portable, de altas prestaciones, multitarea y dinámico"* [W3SUN]:

- Simple: Java ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y más confusas de éstos. Los lenguajes más difundidos son C y C++, pero adolecen de falta de seguridad, característica muy importante para los programas que se usan en *Internet*, por ello Java se diseñó para ser parecido a ellos y así facilitar un rápido y fácil aprendizaje.

  Java elimina muchas de las características de otros lenguajes [FRO00] para mantener reducidas las especificaciones del lenguaje y añadir características muy útiles, como el *garbage collector* (reciclador de memoria dinámica), que se encarga de liberar memoria no usada.

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

  - no se admite aritmética de punteros
  - no existen referencias
  - no existe la definición de registros (`struct`)
  - no existe la definición de tipos (`typedef`)
  - no existe la definición de macros (`#define`)
  - no existe la necesidad de liberar memoria (`free`)

- Orientado a objetos: Java implementa la tecnología básica de C++ con algunas mejoras y elimina algunas cosas para mantener el objetivo de la simplicidad del lenguaje. Java trabaja con sus datos como objetos y con interfaces a esos objetos. Soporta encapsulación, herencia y polimorfismo. Hace uso de la definición de entidades formadas por métodos y variables que reciben el nombre de clases, la instancia de una clase recibe el nombre de objeto.

- Distribuido: Java se ha construido con extensas capacidades de interconexión TCP/IP. Existen librerías de rutinas para acceder e interactuar con protocolos como http y ftp. Esto permite a los programadores acceder a la información a través de la red con tanta facilidad como a los ficheros locales.

  Java en sí no es distribuido, sino que proporciona las librerías y herramientas para que los programas puedan ser distribuidos.

- Interpretado: La máquina virtual Java es un programa que se ejecuta sobre el sistema operativo del ordenador (por lo que es dependiente de la plataforma) y ejecuta directamente el código objeto mediante la interpretación de los *bytecodes*, aunque no se trata de un intérprete tradicional pues éstos ya han pasado por las etapas de validación del compilador Java.

- Robusto: Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución. La comprobación de tipos en Java ayuda a detectar errores, lo antes posible, en el ciclo de desarrollo. Java obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error. Maneja la memoria para eliminar las preocupaciones por parte del programador de la liberación o corrupción de memoria.

  Además, para asegurar el funcionamiento de la aplicación, realiza una verificación de los *bytecodes*, que son el resultado de la compilación de un programa Java.

- Seguro: La seguridad en Java tiene dos facetas. Por una parte se eliminan características C y C++ para prevenir el acceso ilegal a la memoria como los punteros o el casting implícito.

  Por otra parte, el código Java pasa muchos tests antes de ejecutarse en la máquina virtual. Pasa a través de un verificador de *bytecodes* que comprueba el formato de los fragmentos de código y aplica un probador de teoremas para detectar fragmentos de código que falsee punteros, viole derechos de acceso sobre objetos o intente cambiar el tipo o clase de un objeto.

Si los *bytecodes* pasan la verificación sin generar ningún mensaje de error, entonces sabemos que:

- El código no produce desbordamiento de operandos en la pila.
- El tipo de los parámetros de todos los códigos de operación son conocidos y correctos.
- No ha ocurrido ninguna conversión ilegal de datos.
- El acceso a los campos de un objeto se sabe que es legal.
- No hay ningún intento de violar las reglas de acceso y seguridad.

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

Las clases importadas de la red se almacenan en un espacio de nombres privado, asociado con el origen. Cuando una clase del espacio de nombres privado accede a otra clase, primero se busca en las clases predefinidas (del sistema local) y luego en el espacio de nombres de la clase que hace la referencia. Esto imposibilita que una clase suplante a una predefinida.

En resumen, las aplicaciones de Java resultan extremadamente seguras.

- Arquitectura neutral: El compilador Java compila su código a un fichero objeto de formato independiente de la arquitectura de la máquina en que se ejecutará (este fichero con tiene los *bytecodes*). Cualquier máquina que tenga el sistema de ejecución (*run-time,* que sí es dependiente de la máquina) puede ejecutar ese código objeto, sin importar en modo alguno la máquina en que ha sido generado.

- Portable: Más allá de la portabilidad básica por ser de arquitectura independiente, Java implementa otros estándares de portabilidad para facilitar el desarrollo. Los enteros son siempre enteros de 32 bits en complemento a 2 y las cadenas de caracteres utilizan Unicote (no ASCII). Además, Java construye sus interfaces de

usuario a través de un sistema abstracto de ventanas de forma que las ventanas puedan ser implantadas en entornos diferentes (Unix, Pc, Mac, etc.).

- Altas prestaciones: Para los casos en que la velocidad del intérprete Java no resulte suficiente, existen mecanismos como los compiladores **JIT** (Just In Time), que se encargan de traducir, a medida que va siendo necesario, los *bytecodes* a instrucciones de código máquina. También existen otros mecanismos como los compiladores incrementales y sistemas dedicados para tiempo real.

- Multitarea: Java permite muchas actividades simultáneas en un programa. Los *threads* son pequeños procesos o piezas independientes de un gran proceso. Al estar los *threads* construidos en el lenguaje, son más fáciles de usar y más robustos que sus homólogos en otros lenguajes que no los soportan de manera nativa.
  Esta característica permite mejorar el rendimiento interactivo y el comportamiento en tiempo real.

- Dinámico: Java se beneficia todo lo posible de la tecnología orientada a objetos. Java no intenta conectar todos los módulos que comprenden una aplicación hasta el tiempo de ejecución. Las librerías nuevas o actualizadas no paralizarán las aplicaciones actuales (siempre que mantengan el API anterior). Esto permite actualizar el código "en caliente" y facilita el mantenimiento del software.

  Por otro lado, Java proporciona mecanismos para cargar dinámicamente clases desde la red, de manera que nuevos contenidos de información podrán ser tratados por manejadores específicos.

2.3.3 *Applets* de Java

Un *applet* es una mini-aplicación escrita en Java [W3UPM]. Se ejecuta en un navegador compatible con Java (Netscape Navigator 2.x o superior, Microsoft Internet Explorer 3.0 o superior, HotJava,…) al cargar una página HTML. Esta página HTML incluye información sobre el *applet* a ejecutar por medio de los tags `<APPLET>... </APPLET>`.

Algunas características de los *applets* son [W3UPM, W3JAL]:

- Los ficheros de Java compilados (*.class) se descargan a través de la red desde un servidor hasta el browser o navegador en cuya Java Virtual Machine se ejecutan.
- Pueden traer también a través de la red ficheros de imágenes y sonido.
- Los *applets* no tienen ventana propia: se ejecutan en la ventana del browser (en un "panel" incrustado en la página HTML, puesto que la clase *Applet* desciende de *Panel*).
- Tienen importantes restricciones de seguridad, que se chequean al llegar al browser: sólo pueden acceder a una limitada información sobre el ordenador en el que se ejecutan, sólo pueden leer y escribir ficheros en el servidor del que han venido, etc. Los *applets* "de confianza" (trusted) pueden pasar por encima de estas restricciones. Para que un *applet* sea "de confianza" debe estar firmado y tener un certificado que informe al usuario acerca de la seguridad del *applet*. Estos certificados están emitidos por entidades encargadas de verificar dicha seguridad (por ejemplo, VeriSign, cuya página se puede visitar en http://www.verisign.com).
- Los *applets* se pueden probar sin necesidad de browser con la aplicación appletviewer de Sun. También se pueden probar con algunos entornos de desarrollo como Visual J++ 6.0.
- Con un poco de trabajo adicional (básicamente para añadir un método main() que cree una ventana e introduzca en ella el *applet*), las *applets* pueden ser al mismo tiempo aplicaciones y pueden ser ejecutadas de ambas formas.
- Los *applets* no tienen un método *main()* con el que comienzan la ejecución, y derivan de la clase *Applet* que, a su vez, deriva de la clase *Panel.* Los *applets* pueden redefinir cuatro métodos fundamentales heredados de *Applet* que controlan su ejecución:
    - *init():* se llama en cuanto el browser carga el *applet* y se ocupa de todas las tareas de inicialización.
    - *start():* se llama en cuanto el *applet* se hace visible después de haber sido inicializado y también cada vez que se hace visible después de haber estado oculto. En este método se deberían crear *threads* para aquellas tareas que dejarían sin recursos al *applet* o al browser.
    - *stop():* se llama al ocultar el *applet*. En este método se deben parar los *threads* que estén corriendo.

- *destroy():* se llama cuando se va a descargar el *applet* para liberar los recursos que tenga reservados (excepto la memoria). No es necesario redefinir este método.

- Los *applets* también disponen de métodos relacionados con la obtención de información: *getAppletInfo(), getAppletContext(), getParameterInfo(), getParameter(), getCodeBase(), getDocumentBase(), isActive()*. También tienen un método para mostrar información en la barra de tareas y métodos relacionados con imágenes y sonido.
- Se heredan otros muchos métodos de las superclases de *Applet* que tienen que ver con la generación de interfaces gráficas de usuario (AWT) y se suelen redefinir ciertos métodos heredados de otras superclases que están por encima de *Applet* en la jerarquía: los más importantes son *paint()* y *update()*. Debido a que los *applets* son aplicaciones gráficas que aparecen en una zona de la ventana del browser es necesario redefinir estos dos métodos con frecuencia.

2.3.3.1 Incluir un *Applet* en una Página HTML

Se hace mediante el tag doble `<APPLET>...</APPLET>`. La forma general es la siguiente, donde los elementos opcionales aparecen entre corchetes:

```
<APPLET CODE="miApplet.class"
[CODEBASE="unURL"][NAME="unName"]
     WIDTH="wpixels" HEIGHT="hpixels"
     [ALT="TextoAlternativo"]>
          [texto alternativo]
     [<PARAM NAME="MyName1" VALUE="valueOfMyName1">]
     [<PARAM NAME="MyName2" VALUE="valueOfMyName2">]
</APPLET>
```

El atributo **CODEBASE** indica la URL relativa o absoluta del directorio que contiene el *applet*.

El atributo **CODE** indica la clase que se debe cargar para ejecutar el *applet*.

El atributo **NAME** permite dar un nombre opcional al *applet* para poder comunicarse con otros *applets* que se estén ejecutando en la misma página.

El texto alternativo **ALT** muestra el texto para *browsers* que reconocen el tag `<applet>` pero no pueden ejecutar el *applet*.

Los tags **PARAM** permiten pasar parámetros del fichero HTML al programa *Java* del *applet*. Cada parámetro tiene un *nombre* (no distingue entre mayúsculas y minúsculas) y un *valor*, ambos en forma de *String*. El *applet* recupera estos parámetros y si es necesario convierte los *String* en valores numéricos. Los parámetros se obtienen con el método *getParameter(String name)*. El programador deberá prever unos valores por defecto para los parámetros del *applet*, para el caso de que en la página HTML no se definan.

Si queremos obtener los resultados de alguna función de un *applet* para gestionarlos exteriormente, tenemos la posibilidad de llamar a dicha función desde nuestra *Web* utilizando JavaScript. Para ello debemos anteponer al nombre de la función el nombre del *applet*, que se lo damos a través de la etiqueta **NAME** en la cabecera del *applet*:

```
<APPLET CODE=Nombre.class NAME=Nombre width=300 height=250>
```

En el siguiente ejemplo se detalla como se puede llamar a una función llamada `Saludar` de un *applet* llamado `Ejemplo` desde JavaScript:

- Código JavaScript (en la cabecera de la página HTML):

```
<SCRIPT LANGUAJE="JavaScript">
   function Saluda()
   {
        alert(Ejemplo.Saludar());
   }
</SCRIPT>
```

- Código de llamada al *applet* (dentro del cuerpo de la página HTML):

```
<APPLET CODE=Ejemplo.class NAME=Ejemplo width=90 height=70>
   <PARAM NAME="Parametro" VALUE="David">
</APPLET>
```

- Código de llamada de la función `Saluda` (dentro del cuerpo de la página HTML):

```
<FORM>
  <INPUT TYPE="button" VALUE="Saludar" onClick="Saluda()">
</FORM>
```

Al ejecutar el ejemplo anterior observaríamos el *applet* y un botón llamado *Saludar* que pulsaríamos. De este modo, llamamos a la función `Saluda()` de JavaScript, que se encarga de llamar a la función `Saludar()` del *applet* y mostrar por pantalla el mensaje *"Hola David"*.

Capítulo 3

Bases de Datos

En este capítulo vamos, en primer lugar, a realizar una introducción a las bases de datos, para continuar viendo cómo se modelan, centrándonos en el modelo Entidad Relación (ER) y en el modelo Entidad Relación Extendido (EER). Seguidamente, hablaremos del lenguaje SQL, que nos permite crear las bases de datos y manipular los datos que éstas contienen.

3.1 Definición de Base de Datos

Una **base de datos** es un conjunto de datos relacionados entre sí [ELM04]. Por **datos** se entienden entes o hechos conocidos que pueden registrarse y que tienen un significado implícito.

Una base de datos tiene las siguientes propiedades implícitas:

- Una base de datos representa algún aspecto del mundo real, en ocasiones llamado *minimundo*. Las modificaciones del *minimundo* se reflejan en la base de datos.

- Una base de datos es un conjunto de datos lógicamente coherente, con cierto significado inherente. Una colección aleatoria de datos no puede considerarse propiamente una base de datos.

- Toda base de datos se diseña, construye y puebla con datos para un propósito específico.

- Está dirigida a un grupo de usuarios y tiene ciertas aplicaciones preconcebidas que interesan a dichos usuarios.

En otras palabras, una base de datos tiene una fuente de la cual se derivan los datos, cierto grado de interacción con los acontecimientos del mundo real y un público que está activamente interesado en el contenido de la base de datos.

Las bases de datos pueden ser de cualquier tamaño y tener diversos grados de complejidad.

La generación y el mantenimiento de las bases de datos pueden ser manuales o mecánicas.

Un sistema de gestión de bases de datos (SGBD, en inglés DBMS) es un conjunto de programas que permite a los usuarios crear y mantener una base de datos. Por tanto, el SGBD es un sistema de software de *propósito general* que facilita el proceso de definir, construir y manipular bases de datos para diversas aplicaciones. Para definir una base de datos hay que especificar los tipos de datos, las estructuras y las restricciones de los datos que se almacenarán en ella. **Construir** una base de datos es el proceso de guardar los datos en algún medio de almacenamiento controlado por el SGBD. En la **manipulación** intervienen funciones como consultar la base de datos, actualizar la base de datos y generar informes a partir de los datos. Al conjunto formado por la base de datos y el software se le llama sistema de base de datos.

3.2 Modelos de Datos, Esquemas y Ejemplares

Un **modelo de datos** es un conjunto de conceptos que pueden servir para describir la estructura de una base de datos. Con el concepto de estructura de una base de datos nos referimos a los tipos de datos, las relaciones y las restricciones que deben cumplirse para esos datos. Los modelos de datos contienen además un conjunto de **operaciones básicas** para especificar lecturas y actualizaciones de la base de datos.

Se han propuesto muchos modelos de bases de datos y pueden clasificarse dependiendo de los tipos de conceptos que ofrecen para describir la estructura de la base de datos. Los modelos de datos de **alto nivel** o **conceptuales** disponen de conceptos muy cercanos al modo como la generalidad de los usuarios percibe los datos, mientras que los modelos de datos **de bajo nivel** o **físicos** proporcionan conceptos que describen los detalles de cómo se almacenan los datos. Entre estos dos extremos hay una clase de modelos de datos **de representación** (o **de implementación**), cuyos conceptos, pueden ser entendidos por los usuarios finales aunque no están demasiado alejados de la forma en que los datos se organizan dentro del ordenador.

Los modelos de datos de alto nivel utilizan conceptos como entidades, atributos y relaciones.

Los modelos de datos de representación o implementación son los más utilizados en los SGBD comerciales, y entre ellos se cuentan los modelos más comunes: el relacional, el de red, el jerárquico y el orientado a objetos. Representan los datos valiéndose de estructuras de registros, por lo que a veces se les denomina modelo de datos basado en registros.

Los modelos de datos físicos describen cómo se almacenan los datos en el ordenador.

3.2.1 Esquemas y Ejemplares

En cualquier modelo de datos es importante distinguir entre la *descripción* de la base de datos y la base de datos misma. La descripción se conoce como **esquema de la base de datos**. Este esquema se especifica durante el diseño y no es de esperar que se modifique muy a menudo. En la mayoría de los modelos de datos se utilizan ciertas convenciones para representar los esquemas en forma de diagramas, así que la representación de un esquema se denomina diagrama del esquema. Los diagramas de esquema sólo ilustran algunos aspectos del esquema, como los nombres de los tipos de registros y de los elementos de información, y algunas clases de restricciones.

Los datos reales de la base de datos pueden cambiar con mucha frecuencia. Los datos que la base de datos contiene en un determinado momento de denominan **estado de la base de datos** (o conjunto de **ocurrencias** o **ejemplares**). En un estado dado de la base de datos, cada elemento del esquema tiene su propio *conjunto actual* de ejemplares. Cada vez que se inserta o elimina un registro, o que se modifica el valor de un elemento de información, cambia el estado de la base de datos.

3.2.2 Modelo Conceptual de una Base de Datos

El modelo o esquema conceptual es una descripción concisa de los requerimientos de información de los usuarios, y contiene descripciones detalladas de los tipos de datos, las relaciones y las restricciones. Puesto que estos conceptos no incluyen detalles de la implementación, suelen ser más fáciles de entender. El esquema conceptual de alto nivel también puede servir como referencia para asegurarse de satisfacer todos los requerimientos de los usuarios y de que no haya conflictos entre dichos requerimientos. Este enfoque permite

a los diseñadores de la base de datos concentrarse en especificar las propiedades de los datos, sin preocuparse de detalles de almacenamiento.

3.2.3 Conceptos del modelo ER

El modelo Entidad Relación describe los datos como entidades, relaciones y atributos [ELM04].

- **Entidad:** concepto, objeto o cosa que existe en el mundo. En la Tabla 3.1 a) de la vemos cómo se representan las entidades.

- **Atributos:** describen las entidades y las relaciones, y tienen un dominio y un valor. Una entidad particular tendrá un **valor** para cada uno de sus atributos. Estos valores llegan a ser la mayor parte de la información que se almacena en una base de datos.

  - **Tipos de atributos**: en el modelo Entidad Relación se manejan varios tipos de atributos:

    - **Simples**: aquellos que no se pueden descomponer en otros: indivisibles. En la Tabla 3.1 b) vemos cómo se representan los atributos simples.

    - **Compuestos**: se pueden dividir en simples. Los atributos compuestos son útiles para modelar situaciones en las que un usuario en ocasiones hace referencia al atributo compuesto como una unidad, pero otras veces se refiere específicamente a sus componentes. En la Tabla 3.1 c) vemos cómo se representan los atributos compuestos.

    - **Univaluados**: toman un único valor para una entidad.

    - **Multivaluados**: pueden tomar u conjunto de valores para una entidad y pueden tener límites inferior y superior. En la Tabla 3.1 d) vemos cómo se representan los atributos multivaluados.

- **Derivado:** aquél que puede obtenerse a partir de otro atributo, el cuál es un **atributo almacenado.** En algunos casos, una entidad podría no tener ningún valor aplicable para un atributo; en situaciones de este tipo se crea un valor especial llamado **nulo**. En la Tabla 3.1 e) vemos cómo se representan los atributos derivados.

- **Atributos clave o llave**: aquellos que toman valores únicos y distinguen cada instancia que pertenezca a una entidad del mismo tipo. Sus valores pueden servir para identificar de manera única a cada instancia que pertenezca a una entidad. Pueden ser simples (formados por un único atributo) o compuestos (formados por varios atributos simples). Si son compuestos, deben ser mínimos (sin atributos superfluos o innecesarios). Si tienen atributos superfluos se llamará **superllave**. En la Tabla 3.1 f) vemos como se representan los atributos llave.

Cada uno de los atributos simples está asociado a un **conjunto de valores** o **dominio** que especifica los valores que es posible asignar a ese atributo para cada entidad individual.

- **Entidades débiles:** son entidades sin llave que dependen de otras entidades que las "poseen" (**entidad propietaria**) y las identifican unívocamente. En la Tabla 3.1 g) vemos cómo se representan las entidades débiles.

  Se llama relación de identificación (o identificativa) a la relación que relaciona la entidad débil con su entidad propietaria. Las entidades débiles siempre tienen restricción de participación total con respecto a su relación de identificación, porque una entidad débil no se puede identificar sin una entidad propietaria. En la Tabla 3.1 j) vemos cómo se representan las relaciones de identificativas.

  Las entidades débiles tienen una **llave parcial**, que es el conjunto de atributos que pueden identificar de manera unívoca las entidades débiles relacionadas con la misma entidad propietaria. La llave de la entidad débil está formada por su propia llave parcial y la llave de la entidad propietaria. En la Tabla 3.1 h) vemos cómo se representan las llaves parciales.

- **Relaciones:** relaciona varias entidades (E_1, E_2,…, E_N), es un subconjunto del producto cartesiano (E_1 x E_2 x…x E_N).

  El número de entidades participantes en la relación se llama **grado**, de forma que una relación puede ser binaria (formada por dos entidades), ternaria (formada por tres entidades),… En general, las relaciones pueden ser de cualquier grado, pero las más comunes son las binarias. En la Tabla 3.1 i) vemos cómo se representan las relaciones.

- Una **relación recursiva** es una relación sobre la misma entidad, pero desempeñando un papel distinto (**rol**) en cada parte.

- Restricciones en las relaciones: limitan las posibles combinaciones de entidades que pueden participar en las relaciones y se determinan a partir de la situación del *minimundo* que representan estas relaciones.

    - De Cardinalidad: especifica el número de entidades que pueden participar en una relación. En relaciones binarias, las más comunes son 1:1 (de uno a uno), 1:N (de uno a muchos) y N:M (de muchos a muchos).

    - De Participación: especifica si la existencia de una entidad depende de estar relacionada con otra o no. Puede ser de dos tipos:

        - Total: si todas las entidades deben relacionarse.

        - Parcial: si no todas las entidades tienen que relacionarse.

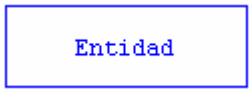
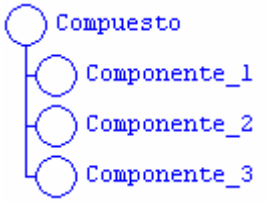
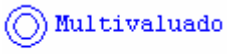
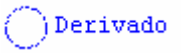
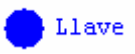
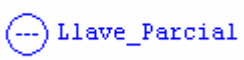
 **a) Entidades**	 **b) Atributos simples**
 **c) Atributos compuestos**	 **d) Atributos multivaluados**
 **e) Atributos derivados**	 **f) Atributos llave**
 **g) Entidades débiles**	 **h) Atributos llave parcial**
 **i) Relaciones**	 **j) Relaciones identificativas**

Tabla 3.1: Símbolos para el modelo ER

3.2.4 Ejemplos

En este apartado se muestran figuras que muestran ejemplos de cómo se conectan los símbolos de la Tabla 3.1.

Ejemplo 3.1: La Figura 3.1 es un ejemplo de una entidad con sus atributos. La entidad representa un cliente, y sus atributos el DNI, el nombre, la dirección (compuesta por Calle, Número y Ciudad), la fecha de nacimiento y la edad que se calcula a partir de la fecha de nacimiento del cliente. Se observa cómo cada atributo está conectado a través de una línea a la entidad o atributo al que pertenece.

Figura 3.1: Entidad con atributos

Ejemplo 3.2: En la Figura 3.2 se muestra un ejemplo de relación binaria. Esta relación muestra cómo un cliente puede comprar artículos en una o más tiendas (teniendo en cuenta que se considera cliente al comprar en alguna tienda) y cómo en una tienda pueden comprar varios clientes, o no comprar ninguno. Observamos cómo los atributos de la relación se conectan con líneas a ella y cómo las entidades participantes en la relación también se conectan mediante líneas a la misma. En caso de participación total (Cliente) estas líneas son dobles, y en el caso en el que la participación no es total (Tienda) las líneas son simples.

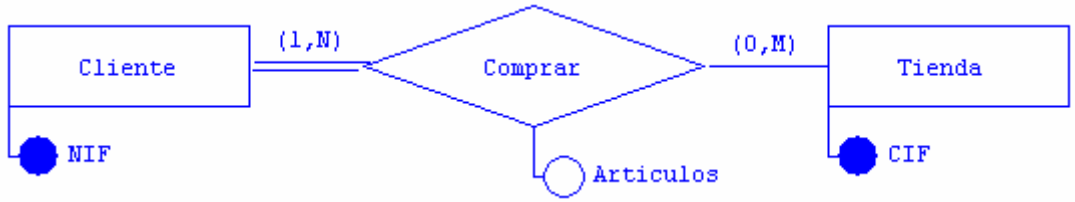


Figura 3.2: Relación binaria

Ejemplo 3.3: En la Figura 3.3 se tiene un ejemplo de relación ternaria. Muestra cómo un autor puede publicar en una o varias editoriales y cómo el autor puede publicar uno o más libros. También indica que una editorial puede publicar uno o más autores y uno o más libros. Del mismo modo muestra que un libro puede estar publicado por uno o varios autores y en una o más editoriales.

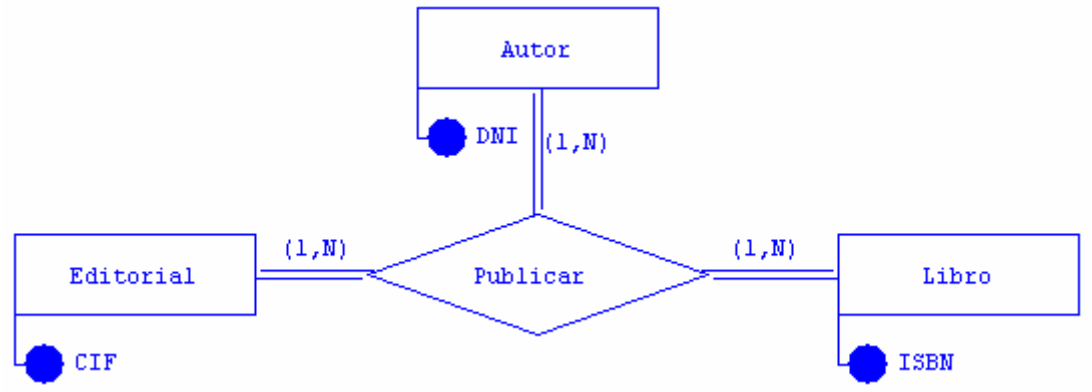


Figura 3.3: Relación ternaria

Ejemplo 3.4: En la Figura 3.4 tenemos un ejemplo de relación identificativa y entidad débil. La entidad débil Sucursal_Banco no tiene sentido si no existe la entidad Banco, quedando este hecho reflejado mediante la relación Sucursal. Es interesante ver cómo la entidad débil tiene participación total con cardinal (1,1), hechos característicos de todas las entidades débiles. Otro hecho a destacar es la presencia de llave parcial en la entidad débil que, para el ejemplo, representa un identificador para cada sucursal.

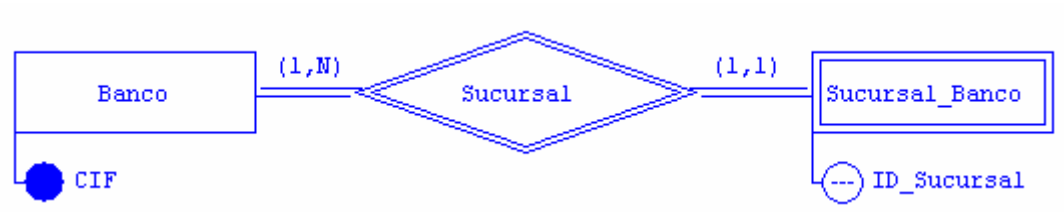


Figura 3.4: Relación identificativa

3.3 Conceptos del Modelo Entidad Relación Extendido (EER)

El modelo EER abarca todos los conceptos del modelo ER y además incluye los conceptos de subclase y superclase y los conceptos relacionados de especialización y generalización; otro concepto que también tiene el modelo EER es el de categoría.

- **Subclase**: grupo de elementos con algo en común, que pertenecen a una entidad.

- **Superclase**: entidad de la que procede una subclase.

- **Especialización**: proceso para definir un conjunto de subclases de una entidad (llamada superclase).

- **Generalización**: proceso inverso a la especialización, consistente en identificar las características comunes a varias Entidades y generalizar todas ellas en una superclase, con las entidades originales como subclases.

- En las figuras 3.6 a 3.9 se muestra cómo se representan las especializaciones o generalizaciones, donde tenemos varios ejemplos de subclases (Perro, Gato, Activo, Jubilado, Camionero, Taxista, Programador y Tecnico) y de superclases (Animal, Trabajador, Conductor e Informatico).

3.3.1 Restricciones sobre Especializaciones/Generalizaciones

Son restricciones que se aplican a una sola especialización o a una sola generalización para determinar las entidades de la superclase que se pueden convertir en miembros de cada subclase. Se tienen los siguientes tipos [BDGA02]:

- Especialización definida por atributo: es aquella en la que todas las subclases tienen la condición de pertenencia sobre el mismo atributo discriminador. Es decir, todas las entidades pertenecientes a una subclase determinada, deben tener el mismo valor para este atributo (el nombre de la subclase). Este atributo se representa como un círculo seguido de su nombre, que se une mediante una línea al círculo que representa a la especialización. En la Figura 3.5 se muestra una especialización definida por el atributo “Tipo”, donde las entidades con “Tipo” = “Piso” podrán pertenecer a la subclase “Piso” y las que tengan “Tipo” = “Chalet” podrán pertenecer a la subclase “Chalet”.

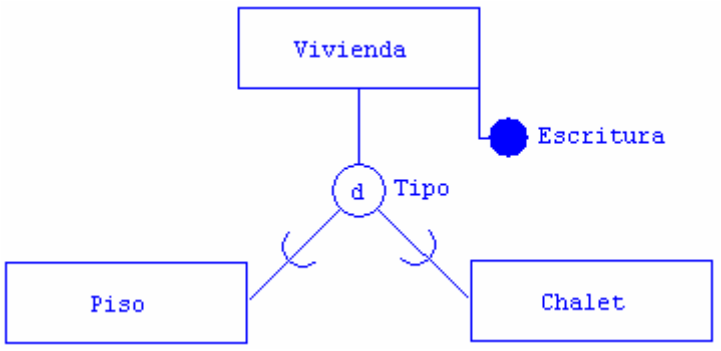


Figura 3.5: Especialización definida por atributo

- Especialización definida por el usuario: es aquella en la que no existe condición para determinar la pertenencia a una subclase. Como ejemplo, se pueden observar todas las figuras 3.6 a 3.9.

- Restricción de Disyunción: determina si la pertenencia a una subclase es o no exclusiva.

    - Disjunta: si una entidad puede ser miembro de una única subclase. Se representa con una "s" encerrada dentro del círculo que representa las especialización/generalización. La Figura 3.6 muestra un ejemplo de especialización disjunta, donde un animal puede ser un gato o un perro, pero no ambos a la vez.

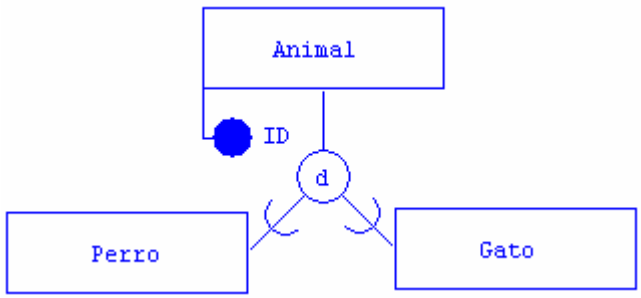


Figura 3.6: Especialización disjunta parcial

    - Solapada: si una entidad de la superclase puede pertenecer a varias subclases. Se representa con una "o" encerrada dentro del círculo que representa la especialización/generalización. La Figura 3.7 muestra un

ejemplo de especialización solapada, donde un informático puede ser programador, técnico, o realizar ambos trabajos al mismo tiempo.

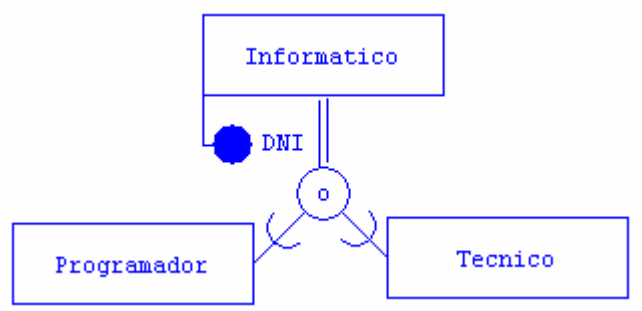


Figura 3.7: Especialización solapada total

- Restricción de Completitud: determina si todas las entidades deben pertenecer a alguna subclase o si por el contrario puede haber entidades que no pertenezcan a ninguna de ellas.

    - Total: cuando cada entidad de la superclase deber ser miembro de alguna/s subclases/s. Se representa con una línea doble uniendo la superclase con el círculo que representa la especialización/generalización. La Figura 3.8 muestra una especialización con restricción de completitud total, donde todo trabajador, o está activo, o ya se ha jubilado.

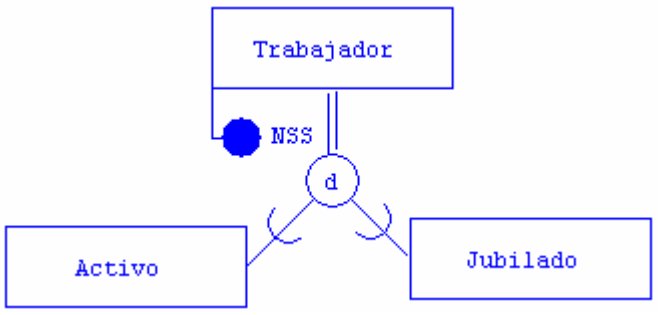


Figura 3.8: Especialización disjunta total

    - Parcial: cuando una entidad puede no pertenecer a ninguna de sus subclases. Se representa con una línea simple entre la superclase y el círculo que representa la especialización/generalización. La Figura 3.9 es un ejemplo con restricción de completitud parcial, pues un conductor puede

ser camionero, taxista o ninguno de ellos (conductor de autobús, por ejemplo).

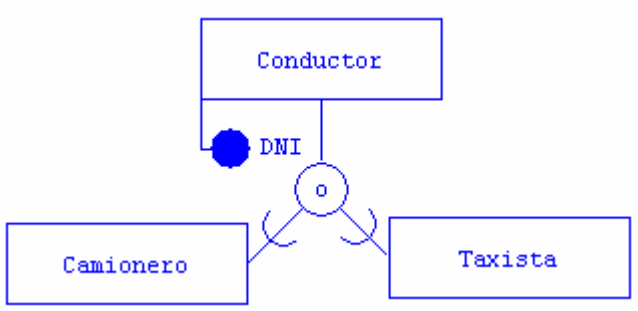


Figura 3.9: Especialización solapada parcial

Cuando una subclase pertenece a más de una superclase se dice que es una subclase compartida.

3.3.2 Tipos Unión o Categoría

Subclase que representa una colección de objetos, que son un subconjunto de la unión de distintos tipos de entidad. Siempre tiene dos o más superclases. Es similar a una subclase compartida, pero una subclase compartida debe pertenecer a todas sus superclases y hereda los atributos de todas ellas, mientras que los miembros de una categoría deber pertenecer a una de las superclases (no a todas) y heredan sólo los atributos de la superclase a las que pertenezca.

De forma análoga a la especialización, también tiene restricción de participación, que indica las superclases que pueden ser miembros de la categoría:

- Total: si todas las superclases de la categoría deben ser miembros de la categoría. En la Figura 3.10 se muestra un ejemplo de tipo unión con participación total.

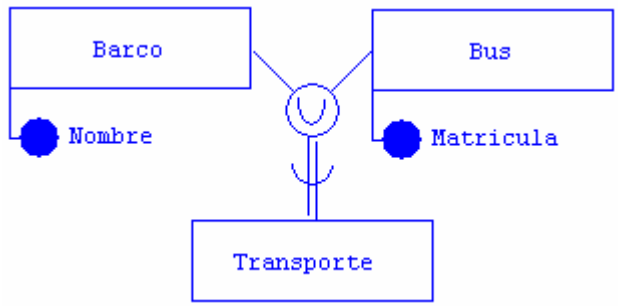


Figura 3.10: Ejemplo de Tipo Unión total

- Parcial: si no todas las superclases deben ser miembros de la categoría. En la Figura 3.11 se muestra un ejemplo de tipo unión con participación parcial.

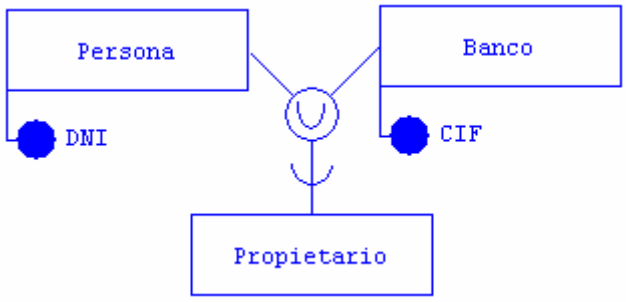


Figura 3.11: Ejemplo de Tipo Unión parcial

3.3.3 Subclases Compartidas

Una subclase compartida es una subclase de varias superclases que tienen la misma llave (si no, sería una categoría). Puede considerarse un tipo intersección, como se ha hecho en este proyecto, de manera que en Derytas las subclases compartidas se representan según se muestra en la Figura 3.12.

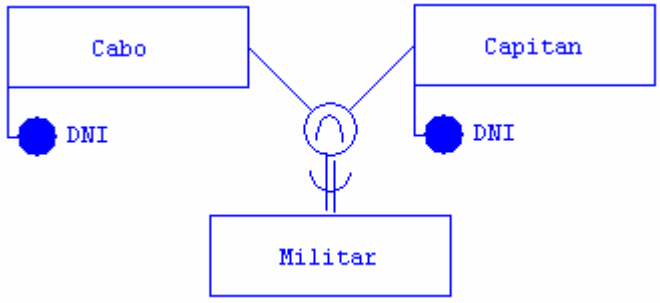


Figura 3.12: Subclase compartida (Tipo Intersección)

3.4 Traducción del Modelo EER en un Esquema de BDR

Una vez diseñado el **Esquema Conceptual** usando el **Modelo EER**, debemos traducirlo al **Esquema BDR**, para ello se utiliza el siguiente algoritmo, que consta de los siguientes ocho pasos [BDGA02]:

- **Paso 1**: Convertir cada entidad normal en una tabla que contenga todos los atributos simples de la entidad y los componentes simples de los atributos compuestos. Elegir uno (o varios) de los atributos llave de la entidad como llave primaria. Si la llave primaria es compuesta, el conjunto de atributos simples que la forman constituirá la llave primaria.

- **Paso 2**: Convertir cada entidad débil en una tabla que contenga todos los atributos simples de la entidad y los componentes simples de los atributos compuestos. También se incluyen como atributos las llaves primarias de las entidades propietarias. La llave primaria estará formada por la combinación de las llaves primarias de las entidades propietarias y la parcial de la entidad débil.

- **Paso 3**: Para cada relación R binaria 1:1 que relacione las entidades S y T, se escoge una de las entidades (por ejemplo S) y se incluyen en ella como llave externa los atributos de llave primaria de la otra. Es mejor que la participación en la relación de la entidad que desempeña el papel de S sea total. En S también se incluyen los atributos simples y los componentes simples de los atributos compuestos de la relación R.

- **Paso 4**: Para cada relación R binaria 1:N que relacione las entidades S y T, siendo S la que participa del lado N de la relación , incluir como llave externa en S la llave primaria de T. Incluir también en S los atributos simples y los componentes simples de los atributos compuestos de la relación R.

- **Paso 5**: Para cada relación R binaria N:M que relacione las entidades S y T, se crea una nueva tabla con los atributos que son llaves primarias de S y T. Juntos serán la llave primaria de la nueva tabla y por separado serán sendas llaves externas. En la

tabla también se incluyen los atributos simples y los componentes simples de los atributos compuestos de la relación R.

- **Paso 6**: Para cada atributo multivaluado se crea una nueva tabla que contiene un atributo correspondiente a dicho atributo más la llave primaria de la entidad a la que pertenece, que será llave externa. Si el atributo multivaluado es compuesto, se incluyen sus componentes simples. La llave primaria será la combinación de todos sus atributos.

- **Paso 7**: Para cada relación R de grado mayor que dos se crea una nueva tabla con los atributos que son llave primaria en las entidades participantes, que serán llaves externas en la nueva tabla. La combinación de estas llaves externas formarán la llave primaria de la nueva tabla. También se incluyen los atributos simples y los componentes simples de los atributos compuestos de la relación R.

- **Paso 8**: Convertir cada especialización en esquemas relacionales siguiendo una de las siguientes 4 opciones, en las que usamos la siguiente notación:

  a) Attrs(E): atributos de la entidad (o clase) E.
  b) PK(E): llave primaria de E.
  c) Sea **C** una superclase con atributos **Attrs(C) = {$\underline{k}$, a_1, a_2,…, a_n}** y con **m** subclases **{S_1, S_2,…, S_m}**

  - **OPCIÓN 8A:** Para cualquier tipo de especialización:
    → Crear una tabla **L** para **C**, con sus atributos.
    → Crear una tabla **L_i** para cada **subclase S_i**, con **$1 \leq i \leq m$**, tal que **$Attrs(L_i) = \{k\} \cup Attrs(S_i)$**. Además, **$PK(L_i) = k$**.

  - **OPCIÓN 8B:** Para una especialización **disjunta-total**:
    → Crear una tabla **L_i** para cada **subclase S_i**, con **$1 \leq i \leq m$**, tal que **$Attrs(L_i) = Attrs(C) \cup Attrs(S_i)$**. Además, **$PK(L_i) = k$**.

- **OPCIÓN 8C:** Para una especialización **disjunta** definida por el **atributo *t*** (si no es definida por atributo, se crea el atributo ***t***):
  → Crear una única tabla **L** con **PK(L) = k**.
  → **Attrs(L) = Attrs(C) ∪ Attrs(S_1) ∪ … ∪ Attrs(S_m) ∪ {t}**.

- **OPCIÓN 8D:** Para especializaciones **solapadas** (o **disjuntas**). Supondremos **m atributos lógicos (booleanos) t_i**, con , tal que cada **t_i** indica si la tupla pertenece o no a la subclase **S_i**.
  → Crear una única tabla **L** con **PK(L) = k**.
  → **Attrs(L) = Attrs(C) ∪ Attrs(S_1) ∪ … ∪ Attrs(Sm) ∪ { t_1, … t_m}**.

3.5 El Lenguaje SQL

SQL (Structured Query Language) es un lenguaje estandarizado de base de datos, el cual nos permite crear tablas, obtener datos y manipularlos de manera muy sencilla ya que es relativamente fácil de usar y aprender.

Es una evolución del lenguaje SEQUEL (Structured English QUEry Language: lenguaje estructurado de consultas en inglés) de Donald D. Chamberlin y Raymond F. Óbice [CHA74, CHA76] y fue implementado por primera vez por IBM un sistema experimental de bases de datos relacionales llamado SYSTEM R [ELM04].

Existen diferentes versiones del lenguaje SQL y está incluido en muchos SGBD, como Oracle (al que va enfocado el código que genera la aplicación [ORA02]), Sybase, Access, SQL Server…de forma que las mismas características sirven en distintos SGBD (la mayoría de las sentencias se construyen de forma similar, como `SELECT`, `UPDATE`, `DELETE`, `INSERT`, `WHERE` y otras…), aunque la mayoría del software para bases de datos tienen sus propias extensiones del lenguaje.

Es un lenguaje completo: tiene comandos que permiten crear y definir nuevas bases de datos (DDL) y comandos para manipular los datos de la base de datos (DML), comandos para dar permisos de acceso distintos para cada usuario, comandos para especificar control de transacciones (recuperación de errores, archivos históricos…), puede incrustarse en lenguajes de alto nivel (como C, C++, Pascal, COBOL,…) y permite operaciones tan complejas que lo hacen potente y expresivo.

A grandes rasgos, los tipos de datos utilizados por el lenguaje son Enteros, Reales,

Caracteres, Cadenas de bits y Fecha y Hora. Se pueden crear nuevos tipos de datos mediante la sentencia `CREATE DOMAIN <Nombre> AS <Tipo>` lo que permite hacer las definiciones más legibles y hacer más fáciles los cambios de dominio de ciertos atributos.

3.5.1 DDL de SQL

DDL son las siglas de Data Definition Language, y está formado por el conjunto de sentencias empleadas para la definición y creación de las bases de datos. En SQL, este conjunto está formado por las siguientes sentencias [BDGA02, GRO94, ORA02]:

- Sentencia para la creación de tablas

```
CREATE TABLE <NombreDeTabla>(
   <NombreDeAtributo1><TipoDeAtributo1><RestriccionesDeAtributo1>,
   … <RestriccionesDeTabla>);
```

En el Ejemplo 3.5 se crea una tabla llamada Persona que tiene dos atributos. El atributo Nombre es una cadena de 20 caracteres y el atributo DNI es una cadena de 9 caracteres que, además, es la llave de la tabla.

Ejemplo 3.5

```
CREATE TABLE Persona
(
   DNI VARCHAR2(9),
   Nombre VARCHAR2(20),
   CONSTRAINT PK_Persona PRIMARY KEY (DNI)
);
```

Sin exagerar, la sentencia DDL más importante es CREATE TABLE. Las tablas están formadas por columnas, que tienen asociadas un nombre y un tipo de datos SQL. Dentro de una tabla se pueden expresar restricciones y relaciones entre tablas (usando claves externas). Se pueden asignar valores por defecto para las columnas, que son utilizados si el usuario no indica explícitamente un valor para esa columna, por ejemplo, en una sentencia INSERT.

- Sentencia para borrar tablas

```
DROP TABLE <NombreDeTabla> [CASCADE | RESTRICT]
```

  - CASCADE: borra la tabla (y su contenido). Si hay referencias sobre ella, dichas referencias son borradas.
  - RESTRICT: borra la tabla si no hay referencias sobre ella.

En el Ejemplo 3.6 se borra la tabla `Persona` creada en el ejemplo anterior.

Ejemplo 3.6

```
DROP TABLE Persona;
```

- Sentencia para modificar tablas

```
ALTER TABLE <NombreDeTabla> <ACCIÓN>
```

Las acciones que pueden realizarse, por ejemplo, son las siguientes:

- Añadir una columna:

```
ADD (<NombreDeAtributo, Tipo, RestricciónDeColumna>);
```

En el Ejemplo 3.7 se añade la columna `Edad` a la tabla `Persona`.

Ejemplo 3.7

```
ALTER TABLE Persona
ADD Edad Number(2) CONSTRAINT MayorEdad CHECK (Edad > 18);
```

- Borrar una columna:

```
DROP COLUMN <NombreDeAtributo> [CASCADE | RESTRICT];
```

En el Ejemplo 3.8 se Elimina la columna `Edad` añadida anteriormente a la tabla `Persona`.

Ejemplo 3.8

```
ALTER TABLE Persona
DROP COLUMN Edad;
```

- Añadir restricción de atributo:

```
MODIFY (<NombreDeAtributo> [<Tipo>][DEFAULT <exp.>]
[[NOT] NULL]);
```

En el Ejemplo 3.9 se añade a la columna `Nombre` de la tabla `Persona` la restricción de que no puede ser nulo.

Ejemplo 3.9

```
ALTER TABLE Persona
MODIFY Nombre NOT NULL;
```

- Añadir restricción de tabla:

```
ADD (<RestricciónDeTabla>);
```

En el Ejemplo 3.10 se añade la restricción en la tabla `Persona` para que el atributo `Edad` sea mayor que 18.

Ejemplo 3.10

```
ALTER TABLE Persona
ADD CONSTRAINT Mayor_Edad CHECK (Edad > 18);
```

- Borrar restricción de tabla (la restricción debe tener un nombre):

```
DROP CONSTRAINT <NombreDeRestricción> CASCADE;
```

En el Ejemplo 3.11 se elimina la restricción de tabla añadida en el ejemplo anterior para `Edad`.

Ejemplo 3.11

```
ALTER TABLE Persona
DROP CONSTRAINT Mayor_Edad CASCADE;
```

Restricciones de Atributos: se declaran a continuación de la declaración de un atributo. Cada atributo puede tener más de una y cada una puede tener un nombre que la identifica.

- **NOT NULL :** si el atributo no puede estar sin valor
- **UNIQUE :** si el valor del atributo no puede repetirse
- **PRIMARY KEY :** si el atributo es llave primaria
- **CHECK (<Condición>) :** el atributo debe cumplir la condición especificada
- **REFERENCES :** si el atributo es llave externa
- **DEFAULT <Valor>:** se asigna el valor **<Valor>** si éste viene sin informar. No se trata de una restricción, pero se declara a continuación de la declaración del atributo.

En el Ejemplo 3.12 se crea la tabla Persona, en la que `DNI` es la llave y `Edad` es un atributo que no puede estar vacío y tiene que ser mayor que 18.

Ejemplo 3.12

```
CREATE TABLE Persona
(
   DNI VARCHAR2(9) CONSTRAINT PK_DNI PRIMARY KEY,
   Edad NUMBER(2)  CONSTRAINT N_Edad NOT NULL
                   CONSTRAINT Mayor_Edad CHECK (Edad > 18)
);
```

Restricciones de Tabla: se declaran a nivel de tabla, posteriormente a la declaración del atributo o atributos a los que se refiere. También pueden tener un nombre, que se asigna al principio.

```
CONSTRAINT <NombreDeRestricción>
| PRIMARY KEY
| UNIQUE
| FOREIGN KEY (<Llave Externa>) REFERENCES <Tabla> (<Atributos>)
      [ON DELETE    {CASCADE | SET NULL | SET DEFAULT}]
      [ON UPDATE {CASCADE | SET NULL | SET DEFAULT}]
| CHECK (<Condición>)
```

Tras las restricciones `PRIMARY KEY` y `UNIQUE` debe haber una lista de los atributos que la forman. Una cosa a tener en cuenta es que aunque todas las restricciones de

atributo pueden declararse como restricciones de tabla, las restricciones de tabla no pueden declararse como restricciones de atributo si involucran varios atributos.

Cabe destacar que, aunque la cláusula `ON UPDATE` es de SQL estándar, no es soportada por Oracle.

En el Ejemplo 3.13 se crea la tabla `Profesor` con la restricción de tabla `PK_Profesor` que indica que la columna `Dni` es la llave primaria de la tabla y la restricción `NombreUnicoProf` que indica que la combinación `Nombre-Apellido1-Apellido2` es única.

Ejemplo 3.13

```
CREATE TABLE Profesor (
   Nombre    VARCHAR2(20),
   Apellido1 VARCHAR2(30),
   Apellido2 VARCHAR2(30),
   Edad      NUMBER(2),
   Dni       VARCHAR2(9),
   CONSTRAINT PK_Profesor
              PRIMARY KEY (Dni),
   CONSTRAINT NombreUnicoProf
              UNIQUE(Nombre,Apellido1,Apellido2)
);
```

- Sentencias para añadir comentarios:

  - A una tabla

    ```
    COMMENT ON TABLE <NombreDeTabla>
    IS '<Comentario>';
    ```

    En el Ejemplo 3.14 se añade un comentario a la tabla `Profesor`.

    Ejemplo 3.14

    ```
    COMMENT ON TABLE Profesor
    IS 'Datos personales de profesores';
    ```

  - A una columna de la tabla

    ```
    COMMENT ON COLUMN < NombreDeTabla >.<NombreDeColumna>
    IS '<Comentario>';
    ```

En el Ejemplo 3.15 se añade un comentario a la columna.

Ejemplo 3.15

```
COMMENT ON COLUMN Profesor.Dni
IS 'Número y letra del DNI';
```

3.5.2 DML de SQL

DML son las siglas de Data Manipulation Language, y está formado por el conjunto de sentencias que permiten obtener los datos almacenados en la base de datos y para modificarlos o eliminarlos.

- **Sentencia para obtener información de la base de datos (consultar):**

```
SELECT <ListaDeExpresiones>
FROM <ListaDeTablas>
WHERE <Condición>
GROUP BY <AtributosParaAgrupar>
HAVING <CondiciónDeGrupo>
ORDER BY <AtributosParaOrdenar>
```

  - **SELECT:** va seguido de las expresiones a recuperar.
  - **FROM:** va seguido de las tablas de las que queremos recuperar los datos.
  - **WHERE:** va seguido de las condiciones que queremos que cumplan los datos que se recuperen.
  - **GROUP BY:** va seguido de los atributos por los que se agrupa el resultado, que son sobre los que se aplican las funciones de grupo o agregación explicadas a continuación en este mismo documento.
  - **HAVING:** va seguido de la condición sobre los grupos.
  - **ORDER BY:** va seguido de los atributos por los que ordenar, teniendo en cuenta que influye el orden en que se indican.

Hay que tener en cuenta que **WHERE**, **GROUP BY**, **HAVING** y **ORDER BY** son opcionales.

En el Ejemplo 3.16 se recupera el `Dni` y el `Nombre` de las personas que tengan más de 18 años:

Ejemplo 3.16

```
SELECT Dni, Nombre
FROM Persona
WHERE Edad > 18;
```

- **Sentencia para guardar información en la base de datos (insertar):**

INSERT INTO <Tabla> (<ListaDeAtributos>)
VALUES (<ValoresDeAtributos>)

Los valores de los atributos deben estar en el mismo orden en el que se especifican en `<ListaDeAtributos>` o en el orden en que fueron creados en la tabla si se omite esta lista.

En el ejemplo 3.17 se introduce una persona con `Dni = '74985633T'`, `Nombre = 'David'`, `Apellido1 = 'Quero'`, `Apellido2 = 'Sanchez'` y `Edad = 26` en la tabla `Persona`.

Ejemplo 3.17

```
INSERT INTO Profesor
VALUES('David', 'Quero', 'Sanchez', 26, '74985633T');
```

- **Sentencia para borrar información de la base de datos:**

DELETE [FROM] <Tabla> [<Alias>] WHERE <Condición>];

Borra los registros de la base de datos que cumplan la condición.
Si se omite `WHERE` y su condición la tabla quedará vacía (pero seguirá existiendo).

En el Ejemplo 3.18 se borra todo el contenido de la tabla `Profesor`, pero la tabla continúa existiendo.

Ejemplo 3.18

```
DELETE Profesor;
```

- **Sentencia para modificar la información de la base de datos:**

  UPDATE <Tabla> [<Alias>] SET <Actualizaciones> [WHERE <Cond>];

  <Actualizaciones> puede ser una lista de asignaciones del tipo *<Atributo> = <Expresión>* o una lista de atributos entre paréntesis a los que se asigna una subconsulta *(<A1>,..., <Am>) = (<Subconsulta>)*

  En el Ejemplo 3.19 se actualiza la `Edad` de la persona con `Dni = '74985633T'` de 26 a 27 años.

  Ejemplo 3.19

```
UPDATE Profesor
SET Edad = 27
WHERE Dni LIKE '74985633T';
```

- **Funciones de Grupo o de Agregación:**
  Son funciones que se aplican sobre un grupo de valores del mismo dominio. Se aplican sobre un conjunto de registros de la base de datos. Las más típicas son:

  - **COUNT**: cuenta el número de registros del grupo. En el siguiente ejemplo se cuenta el número de registros insertados en la tabla Profesor.

```
SELECT COUNT(*)
FROM Profesor;
```

  - **SUM**: calcula la suma de los valores. En el siguiente ejemplo se realiza la suma de la Edad.

```
SELECT SUM(Edad)
FROM Profesor;
```

  - **MAX**: calcula el valor mayor. En el siguiente ejemplo se busca la mayor Edad.

```
SELECT MAX(Edad)
FROM Profesor;
```

  - **MIN**: calcula el valor menor. En el Ejemplo 3.20 se busca la menor Edad.

Ejemplo 3.20

```
SELECT MIN(Edad)
FROM Profesor;
```

- **AVG**: calcula la media aritmética. En el Ejemplo 3.21 se calcula la media de la Edad.

Ejemplo 3.21

```
SELECT AVG(Edad)
FROM Profesor;
```

- **STDDEV**: calcula la desviación típica. En el Ejemplo 3.22 siguiente se calcula la desviación típica de la Edad.

Ejemplo 3.22

```
SELECT STDDEV(Edad)
FROM Profesor;
```

- **VARIANCE**: calcula la varianza. En el Ejemplo 3.23 se calcula la varianza de la Edad.

Ejemplo 3.23

```
SELECT VARIANCE(Edad)
FROM Profesor;
```

3.5.3 SQL:2003

SQL:2003 es el nuevo estándar que sustituye al existente hasta ahora SQL:1999 [W3ACM04]. SQL:2003 hace revisiones de todos los apartados de SQL:1999 y añade uno nuevo, el apartado 14: SQL/XML. El apartado 5 : SQL/Bindings ha sido eliminado en SQL:2003, combinando todo el material contenido en aquel apartado en el apartado 2 de SQL:2003: SQL/Foundation. Los nuevos rasgos introducidos en este apartado son los siguientes:

- Tipos de datos nuevos
- Mejoras a las rutinas SQL-invocadas
- Extensiones a la sentencia `CREATE TABLE`
- Nueva sentencia `MERGE`
- Un objeto nuevo del esquema: el generador de secuencias
- Dos tipos nuevos de columnas: columnas de identidad y columnas generadas

3.5.3.1 Tipos de Datos nuevos

SQL:2003 conserva los tipos de datos de SQL:1999 excepto los tipos de datos `BIT` y `BIT VARYING`. Han sido eliminados del estándar por su poca utilización en las bases de datos existentes y por las pocas expectativas de su uso en el futuro. SQL:2003 introduce tres tipos de datos nuevos: `BIGINT`, `MULTISET` y `XML`. Pueden ser utilizados en los mismo contextos que cualquier otro tipo de datos de SQL, como tipos de columnas, parámetros y tipos de retorno de las rutinas SQL-invocadas, etc.

El tipo de datos `BIGINT` es un valor numérico parecido a los tipos ya existentes `SMALLINT` e `INTEGER`, pero con mayor precisión. Aunque el estándar SQL no indica una precisión concreta para el tipo `INTEGER` (ni para cualquier otro tipo de datos numérico), en general representa valores enteros de 32 bits. Esas implementaciones, por lo general tienen valores `BIGINT` de 64 bits. El tipo `BIGINT` tiene las mismas operaciones aritméticas que el tipo `INTEGER`, por ejemplo, `+`, `−`, `ABS`, `MOD`, etc.

El tipo de datos `MULTISET` es una colección de datos parecido al tipo existente `ARRAY` pero sin un orden implícito. Un multiconjunto es una colección de elementos del mismo tipo, sin orden y permitiendo la existencia de elementos repetidos. El tipo de elemento puede ser cualquier otro tipo de SQL. Por ejemplo, `INTEGER MULTISET` indica el tipo de un valor del multiconjunto cuyo tipo de elemento es `INTEGER`. El tipo de elemento también podría ser otro tipo de colección, que permitiera colecciones anidadas. Un multiconjunto es una colección ilimitada, sin cardinalidad máxima definida. Esto no significa, sin embargo, que

el usuario puede insertar elementos en un multiconjunto sin límite, solamente que el estándar no indica que debería haber un límite. Esto es análogo a las tablas, que no tienen declarado un número máximo de filas. Los valores de un tipo `MULTISET` pueden crearse enumerando sus elementos o mediante una sentencia ; por ejemplo, `MULTISET[1, 2, 3, 4]` o `MULTISET(SELECT nivel FROM cursos)`. El tipo `MULTISET` tiene operaciones para convertir un multiconjunto en un array o en otro multiconjunto cuyos elementos sean de un tipo compatible, para eliminar duplicados del multiconjunto, para devolver el número de elementos de un multiconjunto dado, y para volver el elemento de un multiconjunto que sólo tiene un elemento. Además, también soporta la unión, intersección y diferencia entre multiconjuntos, así como tres nuevas funciones de agregación para crear un multiconjunto del valor del argumento en cada fila de un grupo (`COLLECT`), para crear la unión de multiconjuntos del valor del multiconjunto de cada fila de un grupo (`FUSION`) y para crear la intersección de multiconjuntos del valor del multiconjunto de cada fila de un grupo (`INTERSECTION`).

Ejemplo 3.24: Nuevas funciones de agregación de multiconjuntos

Dada la Tabla 3.2 :

FRIENDS	HOBBIES
'John'	MULTISET['READING', 'POPMUSIC', 'RUNNING']
'Susan'	MULTISET['MOVIES', 'OPERA', 'READING']
'James'	MULTISET['MOVIES', 'READING']

Tabla 3.2: Tabla con los datos de consulta

La siguiente consulta:

```
SELECT COLLECT(FRIEND) AS
          ALL_FRIENDS,
       FUSION(HOBBIES) AS
          ALL_HOBBIES,
       INTERSECTION(HOBBIES) AS
          COMMON_HOBBIES
FROM FRIENDS
```

Devuelve los datos expuestos en la Tabla 3.3:

ALL_FRIENDS	ALL_HOBBIES	COMMON_HOBBIES
MULTISET ['John', 'Susan', 'James']	MULTISET ['READING', 'READING', 'READING', 'POPMUSIC', 'RUNNING', 'OPERA', 'MOVIES', 'MOVIES']	MULTISET ['READING']

Tabla 3.3: Datos devueltos por l aconsulta

Un usuario podría almacenera un documento XML como un VARCHAR o un CLOB. Los documentos XML también pueden ser descompuestos y almacenados en columnas en una o varias tablas [W3ACM02]. La suma de un tipo de datos XML proporciona el potencial para una mayor capacidad y también para un rendimiento mayor. Utilizando este nuevo tipo de datos se pueden definir columnas, variables y parámetros. Los valores legales para este tipo de datos consisten en documentos, elementos, bosques de elementos, nodos de texto, y contenido mixto. Dentro de un elemento pueden existir atributos, pero no son valores legales XML por sí mismos. No están permitidos los comentarios ni las instrucciones dentro del tipo de datos XML. Se han proporcionado varios operadores que producen valores XML. Esos operadores tienen una sintaxis similar a la de las funciones, pero se diferencian de las funciones en algunos detalles. Esos operadores son XMLELEMENT (crea un elemento XML), XMLFOREST (crea un bosque de elementos), XMLGEN (es muy parecido a un constructor de consultas de elementos XML), XMLCONCAT (produce un bosque de elementos concatenando sus argumentos XML), y XMLAGG (es una función de agregación que produce un bosque de elementos a partir de una colección de elementos) [W3ACM02].

3.5.3.2 Funciones de Tabla

Las funciones de tabla son nuevas en SQL:2003, aunque muchos usuarios pueden estar familiarizados con ellas, ya que han estado disponibles en productos SQL bastante tiempo. Una función de tabla es una función SQL-invocada que devuelve una “tabla”. En realidad, el tipo devuelto es equivalente a un multiconjunto de filas (por ejemplo, un tipo MULTISET

cuyo tipo de elemento es un tipo ROW) y no una tabla de verdad, pero se puede consultar como si fuera una tabla. Los ejemplos 3 y 4 muestran las definiciones de una función de tabla externa y de una función de tabla en el cuerpo SQL, respectivamente. Las funciones de tabla externas permiten la incorporación de datos no almacenados en tablas y que vienen de fuera de la base de datos de consultas hechas en ella. El ejemplo 3 devuelve un conjunto de filas representando ciudades y sus condiciones meteorológicas. Se pueden especificar varias opciones para las funciones externas que gobiernan el comportamiento esperado de la función. Aquí, la función está escrita en `C` (indicado por la cláusula `LANGUAGE C`). Es importante saberlo para conocer el mecanismo utilizado para el paso de parámetros, la función por sí misma no llama al motor SQL para ejecutar sentencias SQL (indicado por `NO SQL`), lo que importante saber el tratamiento de las transacciones, distintas llamadas con los mismo valores de entrada devuelven resultados diferentes (indicado por `NOT DETERMINISTIC`), lo que es importante saber para optimizaciones potenciales, y cada parámetro de entrada y salida tiene asociado un indicador nulo (indicado por `PARAMETER STYLE SQL`).

Ejemplo 3.25: Definición de una función de tabla externa

```
CREATE FUNCTION weather()
     RETURNS TABLE (
          CITY VARCHAR(25),
          TEMP_IN_F INTEGER,
          HUMIDITY INTEGER,
          WIND VARCHAR(5),
          FORECAST CHAR(25) )
     NOT DETERMINISTIC
     NO SQL
     LANGUAGE C
     EXTERNAL
     PARAMETER STYLE SQL;
```

Las funciones de tabla en el cuerpo SQL, por otra parte, permiten las llamadas "vistas parametrizadas". En el Ejemplo 3.25, el único parámetro de entrada de la función – en virtud de ser utilizado en el predicado del `WHERE`– determina el conjunto de filas devueltas; por ejemplo, para distintos valores de entrada (números de departamentos) la función devuelve distintos conjuntos de filas (los empleados del departamento indicado por el número de departamento).

Ejemplo 3.26: Definición de una función de tabla en cuerpo SQL

```
CREATE FUNCTION DEPTEMPS
                (DEPTNO CHAR(3))
     RETURNS TABLE (
          EMPNO CHAR(6),
          LNAME VARCHAR(15),
          FNAME VARCHAR(12))
     LANGUAGE SQL
     READS SQL DATA
     DETERMINISTIC
     RETURN TABLE(
          SELECT EMPNO, LASTNAME, FIRSTNME
          FROM EMPLOYEE
          WHERE EMPLOYEE.WORKDEPT =
                DEPTEMPS.DEPTNO)
```

Otra vez, pueden ser especificadas las opciones que gobiernan el comportamiento de la función. LANGUAGE SQL indica que el cuerpo de la función está escrito ten SQL (en este ejemplo consiste en una única sentencia, la sentencia RETURN). READS SQL DATA indica que el acceso a los datos almacenados en la base de datos es sólo para lectura. DETERMINISTIC indica que la función devuelve los mismos resultados para los mismos valores de entrada y el mismo estado de la base de datos. Cuando una función de tabla es invocada en la cláusula FROM, va precedida por la palabra reservada TABLE, como otra más de las tablas, o como la única tabla de referencia.

Ejemplo 3.27: Llamada a una función de tabla

```
SELECT W.CITY, W.TEMP_IN_F, W.FORECAST
FROM TABLE(weather()) AS W
WHERE W.TEMP_IN_F > 65
```

El estándar SQL especifica las reglas exactas de cómo deben ser invocadas las funciones de tabla externas por el motor SQL. Simplificando, esto consiste en tres fases. En la primera fase, la función es invocada una vez con un valor especial para uno de los parámetros que indica que la función se llama por primera vez. Esto permite a la función inicializar las estructuras de datos que se necesitan para las siguientes llamadas. La segunda fase consiste en tantas invocaciones como sean necesarias para transmitir todas las filas al motor SQL. Sólo se transmite una fila por llamada. Cuando no hay más filas que suministrar, la función externa lo indica utilizando un valor especial para uno de los parámetros de retorno. La tercera y última fase consiste en una llamada de la función con un valor especial para uno de los parámetros

indicando a la función que el motor SQL reconoce que no hay más filas para traer y permitiendo que la función libere todos los recursos que le habían sido asignados para satisfacer las peticiones anteriores.

3.5.3.3 Extensiones de `CREATE TABLE LIKE`

En SQL:2003 es posible declarar una columna como una columna de identidad o como una columna generada. En ambos casos, el valor para la columna es generado y asignado de forma automática siempre que se inserta una fila nueva en la tabla. También son nuevas dos extensiones a la sentencia `CREATE TABLE` que son valoradas particularmente por los diseñadores y administradores cuando diseñan tablas nuevas que están basadas en tablas existentes o tienen una estructura similar. En SQL:1999 un usuario ya podía especificar que una tabla nueva sería como una o más tablas existentes. Esto se hacía con la cláusula `LIKE`, que permite copiar la estructura de una o más tablas existentes en la nueva tabla. Sin embargo, la copia estaba restringida al nombre de las columnas y al tipo de datos de las tablas existentes. El Ejemplo 3.28 muestra lo que era posible en SQL.1999. Dada la tabla T1 definida en el primer `CREATE TABLE`, la declaración de la tabla T2 es equivalente a la definición de la tabla T3.

Ejemplo 3.28 : Funcionalidad CREATE TABLE LIKE existente

```
CREATE TABLE T1 (
     C1 INTEGER GENERATED ALWAYS
          AS IDENTITY (START WITH 1, INCREMENT BY 2),
     C2 VARCHAR(100) NOT NULL DEFAULT 'test',
     C3 CHAR(30));

CREATE TABLE T2 (
     LIKE T1,
     C4 CHAR(50));

CREATE TABLE T3 (
     C1 INTEGER,
     C2 VARCHAR(100),
     C3 CHAR(30),
     C4 CHAR(50));
```

Como puede verse en el Ejemplo 3.28, la estructura de la tabla T1 contiene más información que la tabla T2; por ejemplo, la información de que C1 es una columna de identidad se pierde. Así, en SQL:2003 la opción adicional (opcional) para la cláusula `LIKE` es permitir la copia de más información (como las opciones de columna de identidad, las expresiones utilizadas para las columnas generadas y los valores por defecto). El Ejemplo 3.29 muestra las nuevas opciones para copiar la información de identidad y los valores por defecto. La tabla T4 tiene ahora la misma estructura que la la tabla T1.

Ejemplo 3.29: Nuevas opciones de CREATE TABLE LIKE

```
CREATE TABLE T4 (
     LIKE T1
     INCLUDING COLUMN DEFAULTS
     INCLUDING IDENTITY);
```

La sentencia `CREATE TABLE` no establece una dependencia entre la tabla nueva y la tabla o tablas usadas en la cláusula `LIKE`.

3.5.3.4 Extensiones de `CREATE TABLE AS`

Las extensiones CREATE TABLE LIKE expuestas anteriormente son útiles si el usuario quiere una tabla nueva copiando la estructura completa de una o más tablas existentes. Sin embargo, existen circunstancias en las que sería útil copiar sólo un subconjunto de la estructura de una o más tablas existentes o de forma más generalmente cualquier expresión de consulta. Para estos casos, la sentencia CREATE TABLE tiene una extensión (comúnmente llamada `CREATE TABLE AS`) que permite la creación de una tabla con la estructura de una expresión de consulta. Esta extensión también permite renombrar las columnas, si es necesario, e insertar las filas que produce esta expresión de consulta en la nueva tabla creada. El Ejemplo 3.30 muestra cómo crear una tabla nueva y llenarla al mismo tiempo.

Ejemplo 3.30: CREATE TABLE AS

```
CREATE TABLE T5 (D1, D2, D3, D4) AS
     (SELECT T1.C1, T1.C2, T2.C3, T2.C4
      FROM T1, T2
      WHERE T1.C2 = T2.C2) WITH DATA
```

El estándar de la sentencia CREATE TABLE AS no crea una dependencia de la tabla nueva con las de la expresión de consulta, tras la carga inicial, las actualizaciones de las tablas de la expresión de consulta no se reflejarán automáticamente en la tabla nueva.

3.5.3.5 Sentencia `MERGE`

Se trata de una nueva sentencia para actualizar la base de datos. Una exigencia frecuente que surge en la base de datos es la de ser capaz de transferir un conjunto de filas de una "tabla de transacción" a una tabla maestra. Por lo general, la tabla de transacción contiene actualizaciones de las filas existentes en la tabla maestra. El contenido de la tabla de transacción puede ser transferido a la tabla maestra en dos pasos separados, ejecutando una sentencia `UPDATE` para aquellas filas que tienen una ocurrencia en la tabla maestra y un `INSERT` para aquellas filas que no tienen ocurrencia en la tabla maestra. La sentencia `MERGE` de SQL:2003 combina los dos pasos en un solo proceso, haciendo que sea más eficiente, así como más fácil para especificar por el usuario.

3.5.3.6 Generadores de Secuencias

Un generador de secuencias es un nuevo tipo de objeto de la base de datos con un valor numérico que varía con el tiempo. Un generador de secuencias se crea cuando se ejecuta una sentencia `CREATE SEQUENCE`. Como parte de la sentencia `CREATE SEQUENCE`, el usuario puede especificar un valor mínimo, un valor máximo, un valor inicial, un incremento y una opción de iteración para el generador de secuencias que se está creando. El Ejemplo 3.31 se muestra la creación de un generador de secuencias llamado `PARTSEQ`:

Ejemplo 3.31

```
CREATE SEQUENCE PARTSEQ AS INTEGER
     START WITH 1
     INCREMENT BY 1
     MINVALUE 1
     MAXVALUE 10000
     NO CYCLE
```

Cuando se crea, el valor del generador de secuencias se establece al valor inicial. SQL:2003 proporciona una nueva función `NEXT VALUE FOR`, la cual, cuando se aplica a un

generador de secuencias, modifica su valor al siguiente de la secuencia. Cuando se crea un generador de secuencias, los usuarios pueden especificar CYCLE o NO CYCLE para ese generador de secuencias. Si se especifica NO CYCLE, se lanzará una excepción cuando la función NEXT VALUE FOR aplicada al generador de secuencias, trate de devolver un valor que no está en el intervalo limitado por los valores mínimo y máximo del generador de secuencias. Por otra parte, si se especifica CYCLE y al aplicar la función NEXT VALUE FOR al generador de secuencias intenta devolver un valor que no pertenece al intervalo limitado por los valores mínimo y máximo del generador de secuencias, el valor devuelto es el mínimo del generador de secuencias (si el incremento es un número positivo) o el valor máximo (si el incremento del generador de secuencias es un número negativo).

SQL también proporciona una sentencia ALTER SEQUENCE para modificar las propiedades de un generador de secuencias y una sentencia DROP SEQUENCE para eliminar un generador de secuencias. Utilizando una sentencia ALTER SEQUENCE, los usuarios pueden modificar el incremento, el valor máximo, el valor mínimo y la opción de ciclo de un generador de secuencias. Además, los usuarios pueden especificar un valor de reinicio utilizando la sentencia ALTER SEQUENCE ... RESTART WITH ... Este valor especificado es el devuelto por la función NEXT VALUE FOR cuando se aplica inmediatamente después de utilizar la sentencia ALTER SEQUENCE. Esto es útil cuando el usuario quiere que el generador de secuencias genere los siguientes valores a partir de un determinado valor y no de los valores obtenidos al aplicar la función NEXT VALUE FOR anteriormente.

3.5.3.7 Columnas de Identidad

Las columnas de identidad son columnas designadas con la palabra reservada IDENTITY IDENTIDAD, como se muestra en el Ejemplo 3.32:

Ejemplo 3.32

```
CREATE TABLE PARTS (
     PARTNUM INTEGER GENERATED ALWAYS
          AS IDENTITY (START WITH 1
                       INCREMENT BY 1
                       MINVALUE 1
                       MAXVALUE 10000
                       NO CYCLE),
     DESCRIPTION VARCHAR (100),
     QUANTITY INTEGER )
```

Como se ve en el Ejemplo 3.32, las columnas de identidad tienen los mismos atributos que los generadores de secuencias. Esto es porque un generador de secuencias que hereda los atributos de la columna de identidad se asocia conceptualmente a cada columna de identidad. El usuario no tiene que especificar un valor para una columna de identidad cuando se inserta una nueva fila en la tabla conteniendo esa columna de identidad. El valor para esa columna se genera automáticamente invocando la función `NEXT VALUE FOR` de forma implícita

3.5.3.8 Columnas Generadas

Es sabido que el rendimiento de las aplicaciones, particularmente en el área del almacenamiento de datos, se puede mejorar si las expresiones que más se usan son evaluadas una vez y sus resultados se almacenan para usos futuros. Con ese fin, SQL:2003 proporciona una nueva característica llamada columnas generadas. Los usuarios pueden designar una o varias columnas de la tabla como columnas generadas. Cada columna generada se asocia con una expresión escalar. Cuando se inserta una fila en una tabla que contiene una columna generada, se evalúa la expresión asociada con la columna generada se asigna como valor de esa columna.

Por ejemplo, considerar la siguiente sentencia `CREATE TABLE`:

```
CREATE TABLE EMPLOYEES (
     EMP_ID INTEGER,
     SALARY DECIMAL(7,2),
     BONUS DECIMAL(7,2),
     TOTAL_COMP GENERATED ALWAYS
     AS (SALARY + BONUS) )
```

`TOTAL_COMP` es una columna generada de la tabla `EMPLOYEES`. El tipo de datos de `TOTAL_COMP` es el tipo de datos de la expresión `(SALARY + BONUS)`. Opcionalmente, los usuarios pueden especificar un tipo de datos para una columna generada, caso en el que el tipo de datos debe coincidir con el tipo de datos de la expresión asociada.

La siguiente sentencia `INSERT`:

```
INSERT INTO EMPLOYEES (EMP_ID, SALARY, BONUS)
VALUES (501, 65000.00, 5000.00)
```

generaría automáticamente un valor para la columna TOTAL_COMP evaluando la expresión (SALARY + BONUS) e insertando la fila (501, 65000.00, 5000.00, 70000.00) en la tabla EMPLOYEES. Si se incluye una columna generada en la lista de una sentencia INSERT, la correspondiente entrada en la cláusula VALUES debe ser la palabra reservada DEFAULT, suponiendo un error cualquier otra cosa. Por ejemplo, la siguiente sentencia es legal:

```
INSERT INTO EMPLOYEES (EMP_ID, SALARY, BONUS, TOTAL_COMP)
VALUES (501, 65000.00, 5000.00, DEFAULT)
```

pero la siguiente sentencia no lo es:

```
INSERT INTO EMPLOYEES (EMP_ID, SALARY, BONUS, TOTAL_COMP)
VALUES (501, 65000.00, 5000.00, 100000.00)
```

Siempre que se actualiza el valor de cualquier columna referenciada en la expresión asociada con una columna generada, el valor de la columna generada es automáticamente reevaluado y el valor de la columna generada e asignado al nuevo valor. Se puede indicar una columna generada como objetivo de una sentencia UPDATE indicando la palabra DEFAULT; es erróneo indicar cualquier otra cosa. No se permite que las expresiones asociadas con columnas generadas hagan referencia a otras columnas generadas. Se pueden añadir columnas generadas a una tabla utilizando la sentencia ALTER TABLE ADD COLUMN de forma usual. El uso de columnas reservadas puede conducir a un mayor rendimiento no sólo la reducción de cómputo, sino también porque se permiten índices en tales columnas. Por ejemplo, si se pide con frecuencia mostrar una consulta con el resultado en orden descendente por TOTAL_COMP, el resultado podría aparecer en pantalla mucho más rápido si se hubiera creado un índice para la columna TOTAL_COMP.

Capítulo 4

Derytas - Programación de la Aplicación

En este capítulo se expone todo lo relacionado con la programación de la aplicación. Es importante destacar que se ha desarrollado siguiendo unas normas que cubren la mayor parte de los aspectos de la programación.

De la misma manera que es importante generar un código de calidad que sea fácil de mantener, junto con un compromiso con la eficiencia del programa, son importantes los pasos previos a la codificación. Es muy importante realizar una buena gestión de los requisitos de la aplicación para poder elaborar un buen documento funcional donde pueda plasmarse el funcionamiento de la aplicación y, a partir del mismo, poder realizar un documento técnico en el que se indique cómo implementar dicha funcionalidad con las herramientas que disponemos.

En este capítulo se comienza por ver las normas internas de programación adoptadas para, posteriormente, comentar los aspectos más relevantes de la programación de la aplicación.

4.1 Normas para la Nomenclatura

Se ha utilizado la nomenclatura presentada en este apartado para desarrollar las clases, los procedimientos y las funciones de la aplicación.

4.1.1 Variables

Dentro de la definición de variables interviene un factor del cual depende su nomenclatura: el tipo de variable. En la Tabla 4.1 se muestran los prefijos que se han utilizado en los nombres de las variables para especificar su tipo.

Prefijo	Tipo Variable	Ejemplo
n	Número entero	nVerAtributos
i	Número entero que será utilizado como índice para recorrer una estructura de datos	iIndice
s	Cadena de caracteres	sSuperclase
b	Valor booleano	bCerrar
v	Vector	vElementos
o	Objeto instancia de una clase	oMensajes
mi	Menu Item	miEliminar
img	Image	imgLogo
fra	Frame	fraVentana
pop	PopupMenu	popMenu
lbl	Label	lblNombre
txt	TextField	txtNombre
cgb	CheckboxGroup	
chb	Checkbox	chbCerrar
btn	Button	btnCrear
lst	List	lstAtributos
txa	TextArea	txaComentario

Tabla 4.3: Prefijos para identificar el tipo de variable

Para nombrar las variables, se ha elegido un nombre coherente con la función de la misma para proporcionar una mayor legibilidad del programa. El prefijo no se separa del nombre de la variable. Para expresar el tipo, se escribe el prefijo en minúsculas seguido de la/s palabra/s que definan la variable, sin separación y escribiendo la primera letra en mayúsculas, por ejemplo:

```
private boolean bVentanaAbierta;
```

En el nombre de la variable también se indica el ámbito de la misma. Este ámbito se representa con la letra **"g"** delante del prefijo del tipo para indicar que el ámbito de la variable es global. Si no se indica, se entiende que el ámbito de la variable es local.

Para las constantes, el nombre se escribe completamente en mayúsculas (menos el ámbito y el ámbito).

4.1.2 Funciones y Procedimientos

Para nombrar las funciones y los procedimientos se han utilizado nombres descriptivos. Al igual que las variables, contienen una o más palabras, comenzando cada una con letra mayúscula, continuando con letra minúscula y sin separación entre ellas. Para las funciones, se indica el tipo de datos que devuelven de la misma forma que se indica el tipo de las variables, mientras que el nombre de los procedimientos no tiene ningún prefijo, por ejemplo:

```
public int sFiltroNombre(String psNombre)
{
…
}

public void MostrarMensaje(String psMensaje)
{
…
}
```

El nombre de los parámetros de los métodos tiene como prefijo **"p"**, para indicar que se trata de un parámetro.

Para el nombre de las clases se utiliza el prefijo **"cls"**.

4.2 Documentación del Programa: Comentarios

Es muy importante que el código esté correctamente comentado, de manera que su mantenimiento resulte lo más cómodo posible. Se debe comentar de forma coherente, de manera que se aclaren las acciones del código que sean complejas de entender (sobre todo, los bucles `for`, `while`, etc., ya que en ellos se realiza la lógica más importante de los métodos), así como la declaración e inicialización de las variables. Estas aclaraciones se hacen sobre la

línea de código a comentar, con la única excepción de la declaración de variables, donde se hace en la misma línea de la declaración.

Para documentar las clases, los procedimientos y las funciones se utilizan un comentario que los precede. Las clases tienen un comentario que explica la funcionalidad de la clase y donde se muestran los métodos públicos de la clase. Todas las funciones y procedimientos contienen un comentario de descripción que explica qué hace y cuáles son los parámetros que se reciben. También se comenta de qué tipo es el retorno, en el caso de las funciones. A continuación, se muestran ejemplos de comentarios de cabeceras que ilustran el formato de los que se han utilizado para documentar las clases, las funciones y los procedimientos:

- Cabecera para clases:

```
////////////////////////////////////////////////////////////////
//                                                            //
// Clase: Nombre de la clase                                  //
// Métodos: Métodos de la clase                               //
// Descripción: Funcionalidad de la clase                     //
//                                                            //
////////////////////////////////////////////////////////////////
```

- Cabecera para funciones:

```
////////////////////////////////////////////////////////////////
//                                                            //
// Función: Nombre de la función                              //
// Parámetros: - NombreParámetro (Tipo): Descripción parámetro //
// Retorno: Valor devuelto por la función                     //
// Descripción: Funcionalidad de la función                   //
//                                                            //
////////////////////////////////////////////////////////////////
```

- Cabecera para procedimientos:

```
////////////////////////////////////////////////////////////////
//                                                            //
// Procedimiento: Nombre del procedimiento                    //
// Parámetros: - NombreParámetro (Tipo): Descripción parámetro //
// Descripción: Funcionalidad del procedimiento               //
//                                                            //
////////////////////////////////////////////////////////////////
```

  Debido a la sintaxis del lenguaje de programación utilizado, donde las sentencias se agrupan con llaves (‘{‘, ‘}’), también se comenta el final de cada bloque de código, indicando la clase, método o sentencia para el que fue abierto.

4.3 Estructura de la Aplicación

Derytas está formada por un conjunto de páginas HTML, que se muestran a continuación:

- Página de presentación: Simplemente muestra información sobre la aplicación y contiene enlaces a la página de la Universidad de Málaga, a la Escuela Técnica Superior de Ingeniería Informática y a Derytas a través de los logotipos correspondientes, tal y como se muestra en la Figura 4.1.

Figura 4.1: Página de presentación (index.html)

- Página de la aplicación (marco.html): Contiene 2 *frames* en los que muestra la página de botones a la izquierda y la página del esquema a la derecha (Figura 4.2). Éstas pueden comunicarse gracias a esta página, permitiendo que la página de botones haga las llamadas a las funciones JavaScript que implementa la página del esquema.

- Página de botones (botones.html): Contiene los botones a través de los que se realizan las acciones deseadas para construir el esquema (ver Figura 4.2, izquierda). Estas acciones se llevan a cabo mediante la llamada a funciones JavaScript de la página que muestra el esquema. El único botón que no llama a una función JavaScript es el de la ayuda, que la abre directamente en una nueva ventana del explorador. La ayuda está formada por un conjunto de páginas HTML, y se muestra en este mismo apartado.

- Página para dibujar el esquema (Derytas.html): Esta página constituye el motor de la aplicación (Figura 4.2, derecha), ya que implementa las funciones JavaScript que se

llaman desde la página de botones y contiene el *applet* Derytas, que contiene todos los métodos necesarios para dibujar el esquema y generar el *script* de generación de la base de datos. Las funciones JavaScript, a su vez, llaman a los métodos del *applet* para realizar la acción deseada.

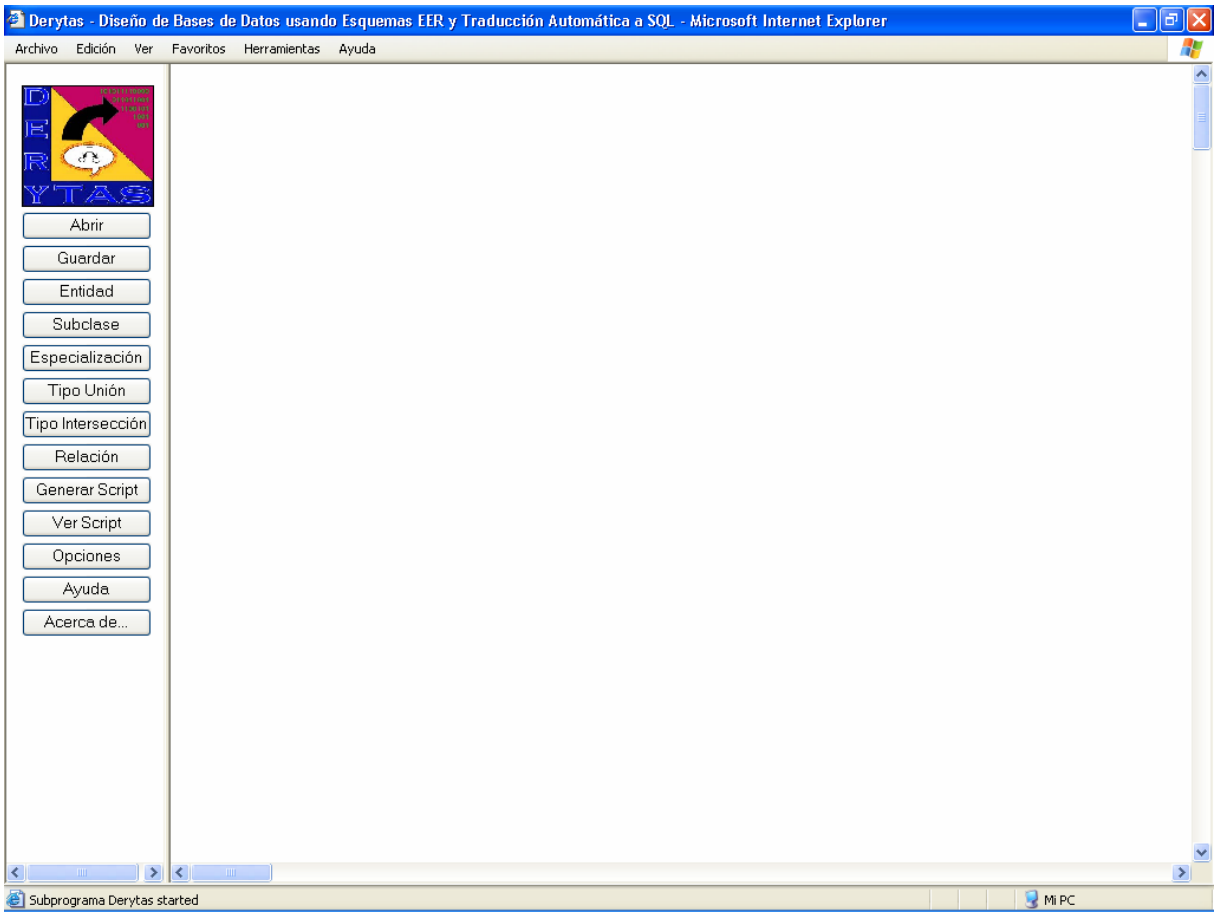


Figura 4.2: Página de la aplicación (marco.html)

- Página de ayuda: La ayuda está formada por un conjunto de 16 páginas, cuyo funcionamiento se explica a continuación.

  La página "Ayuda.html" (Figura 4.3) contiene 2 *frames*, mostrando en el *frame* izquierdo la página "enlaces.html" y en el derecho una de las 14 páginas correspondiente a uno de los 14 temas en los que se encuentra dividida la ayuda.

  La página "enlaces.html" muestra los temas en los que está divida la ayuda (el índice) que, a su vez, son enlaces a la página que contiene la información sobre dicho tema, y que se muestra en el *frame* derecho de la página de ayuda.

Figura 4.3: Página de ayuda (Ayuda.html)

4.4 Programación HTML/JavaScript: Portada, Interfaz con el *Applet* y con el Sistema de Ficheros

Como se ha comentado en el apartado anterior, el motor de la aplicación es la página "Derytas.html", que contiene el *applet* "Derytas.class" y las funciones JavaScript para acceder a los métodos de este *applet*.

Se ha decidido que la lógica resida en un *applet* por las ventajas que éstos aportan para dibujar gráficos, por la seguridad que aporta el lenguaje Java y por las facilidades para programarlos. Uno de los mayores problemas que se han tenido que resolver, derivado precisamente de la seguridad del lenguaje, ha sido el acceso a ficheros. Para que un *applet* pueda acceder al sistema de ficheros, debe estar firmado y poseer un certificado de seguridad. Para simplificar el desarrollo, incorporar nuevas herramientas y mostrar un ejemplo de la conexión JavaScript-Java, se ha decidido que sea JavaScript quien trate con los ficheros. Así,

cuando se quiere guardar en un fichero, una función JavaScript llama a una función del *applet* para obtener los datos que se desean guardar y los graba en el fichero. Cuando se desean leer datos de un fichero, una función JavaScript obtiene los datos y se los pasa a un método del *applet* para que los procese.

El diseño de "Derytas.html", que ofrece un conjunto de funciones en JavaScript para acceder al *applet* permite que cualquiera con conocimientos en páginas HTML pueda construir el interfaz gráfico que más le guste basándose en llamadas a estas funciones, que se exponen a continuación:

- `function Abrir()`

  Su utilidad es leer el contenido de un fichero que contiene un esquema. No recibe parámetros. Crea un objeto `ActiveXObject` para manejar el fichero, por lo que la primera vez que se ejecuta aparece un mensaje de seguridad (Figura 5.27). Si ocurre algún error creando este objeto, se llama al método `AvisoRecargar()` del *applet*, que muestra un mensaje indicando que se debe volver a cargar el *applet* (Figura 5.28). Obtiene el nombre del fichero y la ruta en la que se encuentra llamando a la función `sObtenerFicheroEsquema(0)` del *applet*, indicando en el argumento que se pasa que el fichero se quiere abrir para lectura. Lee el contenido del fichero y lo almacena en una cadena. Llama al método `Abrir(s)` al que le pasa esta cadena como parámetro y, a partir de ella, el método del *applet* construye la estructura de datos y pinta el esquema.

- `function Guardar()`

  Su utilidad es guardar en un fichero el contenido de un esquema. No recibe parámetros. Crea un objeto `ActiveXObject` para manejar el fichero, por lo que la primera vez que se ejecuta aparece un mensaje de seguridad (Figura 5.27). Si ocurre algún error creando este objeto, se llama al método `AvisoRecargar()` del *applet*, que muestra un mensaje indicando que se debe volver a cargar el *applet* (Figura 5.28). Obtiene el nombre del fichero y la ruta en la que se encuentra llamando a la función `sObtenerFicheroEsquema(1)` del *applet*, indicando en el argumento que se pasa que el fichero se quiere abrir para escritura. Llama a la función `sGuardar()`

del *applet*, que devuelve una cadena con los datos del esquema, y la guarda en el fichero indicado.

- `function AbrirEntidad()`

  Su utilidad es abrir la ventana que permite la creación de entidades. Para ello, llama al método `AbrirEntidad()` del *applet*, que es el que realmente muestra la ventana con la lógica necesaria para crear las entidades.

- `function AbrirSubclase()`

  Su utilidad es abrir la ventana que permite la creación de subclases. Para ello, llama al método `AbrirSubclase()` del *applet*, que es el que realmente muestra la ventana con la lógica necesaria para crear las subclases.

- `function AbrirEspecializacion()`

  Su utilidad es abrir la ventana que permite la creación de especializaciones. Para ello, llama al método `AbrirEspecializacion()` del *applet*, que es el que realmente muestra la ventana con la lógica necesaria para crear las especializaciones.

- `function AbrirTipoUnion()`

  Su utilidad es abrir la ventana que permite la creación de tipos unión. Para ello, llama al método `AbrirTipoUnion()` del *applet*, que es el que realmente muestra la ventana con la lógica necesaria para crear los tipos unión.

- `function AbrirTipoInterseccion()`

  Su utilidad es abrir la ventana que permite la creación de tipos intersección. Para ello, llama al método `AbrirTipoInterseccion()` del *applet*, que es el que realmente muestra la ventana con la lógica necesaria para crear los tipos intersección.

- `function AbrirRelacion()`

  Su utilidad es abrir la ventana que permite la creación relaciones. Para ello, llama al método `AbrirRelacion()` del *applet*, que es el que realmente muestra la ventana con la lógica necesaria para crear las relaciones.

- `function GenerarScript()`

  Su utilidad es guardar en un fichero el *script* de generación de la base de datos. No recibe parámetros. Crea un objeto `ActiveXObject` para manejar el fichero, por lo que la primera vez que se ejecuta aparece un mensaje de seguridad (Figura 5.27). Si ocurre algún error creando este objeto, se llama al método `AvisoRecargar()` del *applet*, que muestra un mensaje indicando que se debe volver a cargar el *applet* (Figura 5.28). Obtiene el nombre del fichero y la ruta en la que se encuentra llamando a la función `sObtenerFicheroScript()` del *applet*. Llama a la función `sLeerScript()` del *applet*, que devuelve una cadena con el *script* y lo guarda en el fichero indicado.

- `function VisualizarScript()`

  Su utilidad es mostrar una ventana con el *script* de generación de la base de datos del esquema que se está creando. Para ello, llama al método `VisualizarScript()` del *applet*, que es el que muestra la ventana, genera el *script* y lo muestra en la ventana.

- `function AbrirConfig()`

  Su utilidad es mostrar la ventana de configuración de la aplicación. Para ello, llama al método `AbrirConfig()` del *applet*, que es el que muestra la ventana de configuración.

- `function AcercaDe()`

  Su utilidad es mostrar una ventana con información relativa a la aplicación. Para ello, llama al método `AcercaDe()` del *applet*, que es el que muestra dicha ventana.

4.5 Programación Java: Gráficos del Esquema EER y Generación del *Script*

La parte Java soporta la mayor parte del peso de la aplicación y desarrolla la tarea más importante: pintar el esquema y generar el *script*, los pilares del proyecto.

Uno de los objetivos que se ha intentado conseguir durante la programación, es tener un código estructurado, aplicando la técnica de "divide y vencerás", de manera que se obtenga un conjunto de métodos que nos permitan utilizar las clases sin que sea necesario saber cómo funcionan esos métodos. Por ello, la parte Java se divide en 19 clases, encapsulando en cada una de ellas una funcionalidad específica. A continuación se muestran las clases, comentando sus variables y métodos más importantes (aunque todos se encuentran debidamente comentados en el código), así como el diagrama UML de algunas clases cuyo tamaño lo permita:

4.5.1 Clase `clsConstantes.java`

El objetivo de esta clase es proporcionar un conjunto de constantes a utilizar en el desarrollo de la aplicación. Esto evita el "*hardcode"*, una de las buenas prácticas de programación que se intenta seguir en este desarrollo. Al tener todas las constantes definidas en una misma clase, el mantenimiento de éstas es más sencillo que si se definieran en cada clase, ya que seguro estarían repetidas y un cambio en alguna de ellas implicaría un trabajo mayor.

4.5.2 Clase `clsMensajes.java`

Esta clase construye una ventana (un `Frame`) en la que se muestra el mensaje que se indique al usuario. Proporciona los siguientes métodos para mostrar el mensaje:

- `public void MostrarMensaje(String psMensaje,`
  `String psParam,`
  `Frame pfraLlamante)`

  Muestra la ventana creada por la clase, en la que se lee el mensaje con texto `psMensaje` que tiene un parámetro que se sustituye por `psParam`. El parámetro `pfraLlamante` es una referencia al `Frame` donde se ha producido algún error al realizar la validación de un elemento y que llama a este método de esta clase. Esta referencia se pasa para ocultar el `Frame` al mostrar el mensaje.

- `public void MostrarMensaje(String psMensaje,`
  `Frame pfraLlamante)`

  Muestra la ventana creada por la clase, en la que se lee el mensaje con texto `psMensaje`. El parámetro `pfraLlamante` es una referencia al `Frame` donde se ha producido algún error al realizar la validación de un elemento y que llama a este método de esta clase. Esta referencia se pasa para ocultar el `Frame` al mostrar el mensaje.

- `public void MostrarMensaje(String psMensaje)`

  Muestra la ventana creada por la clase, en la que se lee el mensaje con texto `psMensaje`.

4.5.3 Clases `clsEntidad.java`, `clsSubclase.java`, `clsEspecializacion.java`, `clsRelacion.java`, `clsAtributo.java`, `clsTipoUnion.java` y `clsTipoInterseccion.java`

El objetivo de cada una de estas clases es crear y/o modificar entidades, subclases, especializaciones, relaciones, atributos, tipos unión y tipos intersección respectivamente. Estas clases descienden de `Frame`, heredando sus métodos y propiedades. Cada una muestra

una ventana distinta donde se introduce la información necesaria para crear estos elementos. Cuando se desea modificar, la ventana muestra la información que se introdujo en el momento de su creación para que pueda ser modificada. Cada clase controla que los datos para la creación o modificación respeten las restricciones a la que está sujeto cada elemento que compone el esquema.

4.5.4 Clase `clsEntRel.java`

Clase para contener una entidad participante en una relación. Esta clase no posee métodos, ya que simplemente almacena el nombre de una entidad participante en una relación y las cardinalidades mínima y máxima con las que participa en ella. En la Figura 4.4 se muestra el diagrama UML de la clase.

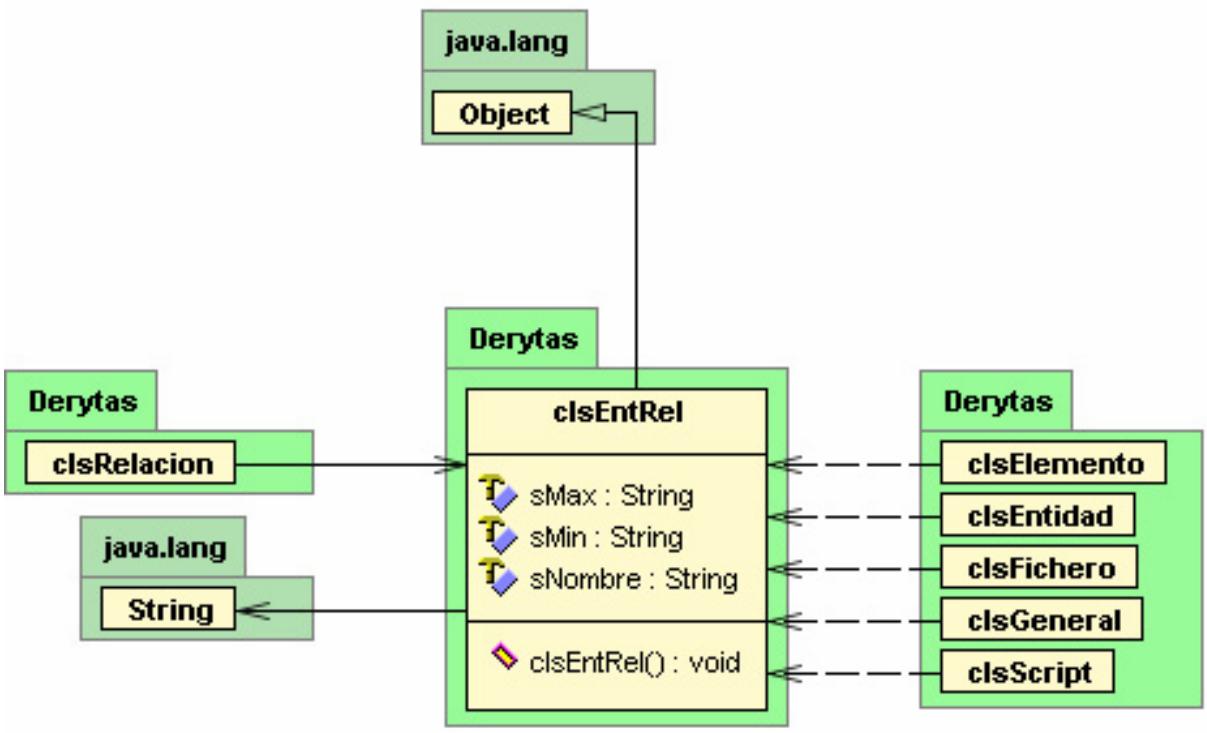


Figura 4.4: Diagrama UML de la clase clsEntRel

El diagrama muestra las variables de clase (`sMin`, `sMax` y `sNombre`) y el único método que posee, `clsEntRel`, el constructor. Muestra las referencias de esta clase, como son `String` y `Object` de `java.lang`. La primera de estas referencias se debe a que las variables de la clase son de tipo `String`, y la segunda es debida a que hereda directamente

de la clase Object. Indica que en la clase clsRelacion se crea una instancia de la clase y que en las clases clsElemento, clsEntidad, clsFichero, clsGeneral y clsScript se utilizan objetos que son instancias de clsEntRel.

4.5.5 Clases clsEliminarElemento.java y clsEliminarComoSubclase.java

La función de estas clases es eliminar elementos del esquema y la relación superclase-subclase entre entidades, respectivamente. La primera de estas clases muestra un mensaje esperando la confirmación del usuario para eliminar el elemento del esquema seleccionado (entidades, relaciones, especializaciones, tipos unión o tipos intersección).

La segunda clase, elimina la relación superclase-subclase entre dos entidades, previa confirmación por parte del usuario para realizar dicha acción.

La existencia de estas dos clases por separado se debe a que la primera elimina un elemento del esquema como tal, es decir, un objeto de la clase clsElemento, mientras que la segunda no elimina un elemento del esquema, sino una relación existente entre dos entidades del esquema. Esta relación superclase-subclase entre dos entidades, no se ve reflejada como un objeto de la clase clsElemento, sino que las entidades implicadas almacenan el nombre de la entidad de la que es superclase y el nombre de la entidad de la que es subclase, respectivamente.

En la Figura 4.5 se muestra el diagrama UML de la clase clsEliminarComoSubclase. En éste se muestran la variable de clase (oGeneral) y los métodos que posee: clsEliminarComoSubclase () (constructor), Mostrar(), y Ocultar(). También muestra los componentes utilizados para construir la ventana de opciones de configuración.

La clase hereda directamente de Frame, y hace referencia a los componentes utilizados, como son Button, y Label. Muestra los objetos que referenciados indirectamente, es decir, que no se crea explícitamente una instancia de ellos, de java.awt (para los gráficos) y de java.awt.event (para los eventos gráficos). La referencia a Object es debida a que todos los objetos heredan, directa o indirectamente de Object. El diagrama también muestra una referencia a la clase clsGeneral, debido a que es el

parámetro del constructor. La flecha entrante procedente de la clase `Derytas` se debe a que esa clase crea una instancia de `clsEliminarComoSubclase`.

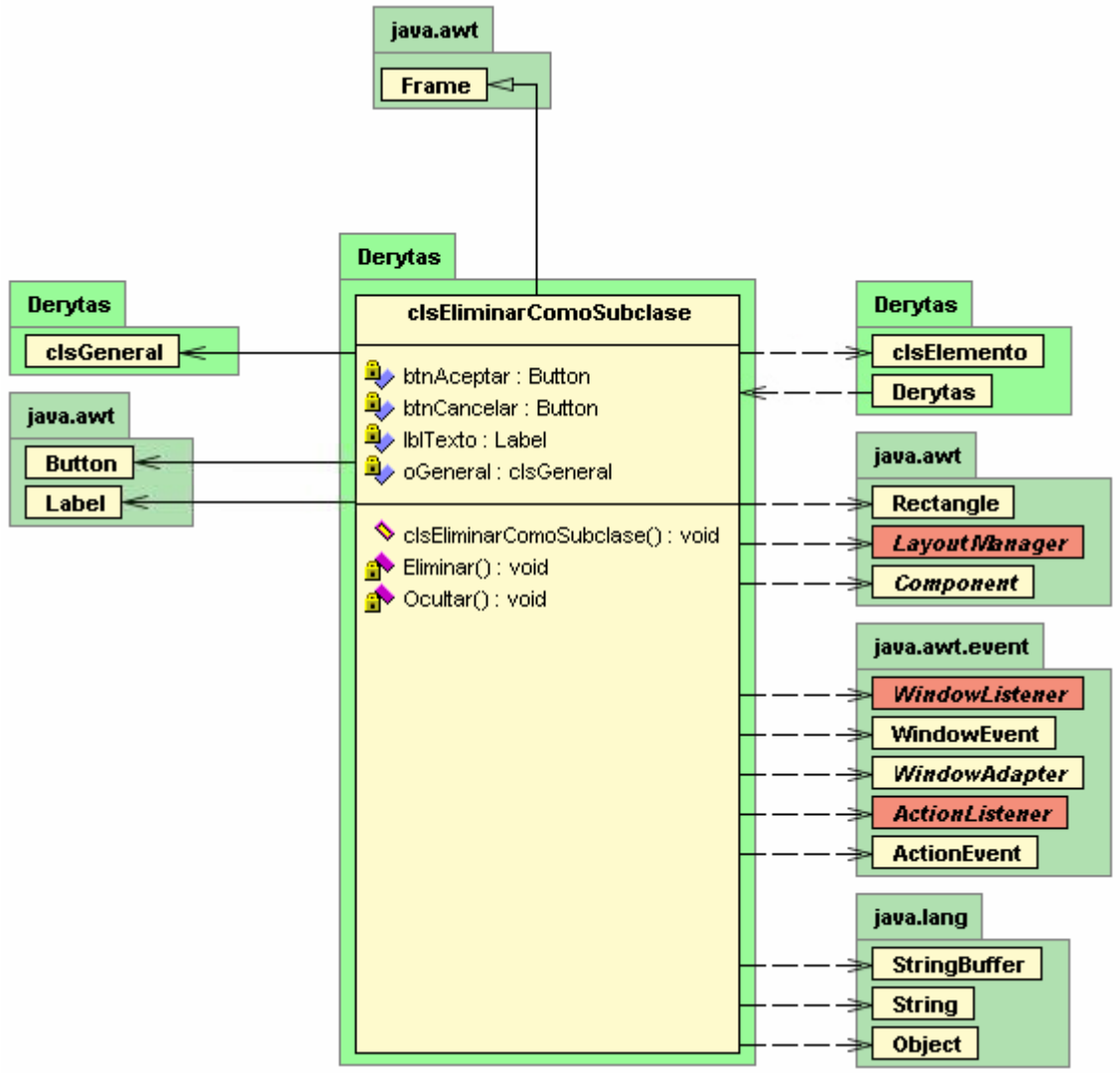


Figura 4.5: Diagrama UML de la clase clsEliminarComoSubclase

4.5.6 Clase `clsFichero.java`

Clase para guardar y leer esquemas EER. Gracias a esta clase se generan los datos necesarios para guardar el esquema en un fichero (con extensión ".eer").

También gracias a esta clase se crea la estructura de datos del esquema con los datos leídos del fichero que contiene el esquema y se dibuja el mismo en el *applet*.

Los métodos que proporciona esta clase son los siguientes:

- `public String sGuardar()`

Esta función, que no tiene argumentos, devuelve una variable de tipo `String` con los elementos del esquema, construida a partir de la variable `vElementos`. Esta cadena de caracteres tiene el formato adecuado para ser guardada en un fichero y poder recuperar el esquema EER posteriormente a partir de este fichero. El proceso para construir la cadena con los datos del esquema, se expone a continuación:

En primer lugar, llama a la función `sGuardarConfiguracion()`, que se comenta a continuación:

- `private String sGuardarConfiguracion()`

  Esta función, que no tiene argumentos, devuelve una cadena de caracteres con los valores de las variables de configuración de la clase `clsConfig`.

La cadena devuelta por la función anterior se almacena en la variable de tipo `String` que se devuelve al final de esta función. A continuación, recorre la variable `vElementos` que contiene los elementos del esquema y, para cada uno, llama a la función `sGuardarElemento()`, que se comenta a continuación:

- `private String sGuardarElemento(clsElemento oElemento)`

  Esta función recibe un objeto de la clase `clsElemento`, `oElemento` , y devuelve una cadena de caracteres con los datos de ese elemento para ser guardados en un fichero.

  En primer lugar, identifica el tipo de elemento, para llamar a una función específica para cada tipo:

  o Si es una entidad o una entidad débil, llama a la función `sGuardarEntidad()`, a la que se le pasa un objeto de la clase `clsElemento` con el elemento en cuestión, y que devuelve una cadena de caracteres con los datos del elemento, como se muestra a continuación:

- `private String sGuardarEntidad(clsElemento oElemento)`

  Devuelve una cadena de caracteres que contiene el tipo de elemento, el nombre, la posición en pantalla, el número de atributos. En caso de que el número de atributos sea mayor que cero, recorre el vector de atributos de la entidad, llamando a la función `sGuardarAtributo()` para cada uno (se comenta al final de esta sección). Esta función devuelve una cadena de caracteres con los datos para guardar el atributo en cuestión en un fichero. Esta cadena de caracteres se concatena con la cadena que contiene los datos de la entidad.

  A continuación, obtiene el número de relaciones en los que participa la entidad, que también se guarda en la cadena con los datos de la entidad. En caso de que este valor sea mayor cero, recorre el vector que tiene la entidad con el nombre de las relaciones en las que participa para guardarlos en la cadena de caracteres que contiene los datos. Este comportamiento se repite, de forma análoga, para las especializaciones, tipos unión y tipos intersección en los que interviene la entidad. Seguidamente, comprueba si es superclase o subclase de otra entidad (directamente, no a través de una especialización, tipo unión o tipo intersección). Concatena un valor que indica si es o no subclase o superclase y, en caso afirmativo, también concatena el nombre de la entidad de la que es superclase o subclase. A continuación almacena si los atributos se pintarán en la parte izquierda o derecha de la entidad. Por último, se guarda un indicador para la presencia de comentario y, en caso de que exista comentario, se guarda el texto correspondiente al mismo. La última acción de la función es devolver la cadena de caracteres construida con los datos de la entidad.

o Si es una relación o una relación identificativa, llama a la función `sGuardarRelacion()`, a la que se le pasa un objeto de la clase `clsElemento` con el elemento en cuestión, y que devuelve una cadena de caracteres con los datos del elemento.

- `private String sGuardarRelacion (clsElemento oElemento)`

  Devuelve una cadena de caracteres que contiene el tipo de elemento, el nombre, la posición en pantalla, el número de atributos. En caso de que el número de atributos sea mayor que cero, recorre el vector de atributos de la relación, llamando a la función `sGuardarAtributo()` de igual forma que se hace en la función `sGuardarEntidad()`.

  A continuación, obtiene el número de entidades que participan en la relación, que también se guarda en la cadena con los datos de la entidad. En caso de que este valor sea mayor cero, recorre el vector que tiene la relación con el nombre de las entidades participantes para guardarlos en la cadena de caracteres que contiene los datos. La última acción de la función es devolver la cadena de caracteres construida con los datos de la relación.

o Si es una especialización de cualquier tipo, llama a la función `sGuardarEspecializacion()`, a la que se le pasa un objeto de la clase `clsElemento` con el elemento en cuestión, y que devuelve una cadena de caracteres con los datos del elemento.

- `private String sGuardarEspecializacion (clsElemento oElemento)`

  Devuelve una cadena de caracteres que contiene el tipo de elemento, el nombre, la posición en pantalla y el nombre de la superclase de la especialización. A continuación, concatena el número de subclases de la especialización y recorre el vector que tiene la especialización con el nombre de las subclases para guardarlos en la cadena de caracteres que contiene los datos. Por último, comprueba si se ha indicado un nombre para el atributo de la especialización, concatenando este nombre o un indicador de su ausencia. La última acción de la función es devolver la cadena de caracteres construida con los datos de la especialización.

o Si se trata de un tipo unión, llama a la función `sGuardarTipoUnion()`,a la que se le pasa un objeto de la clase `clsElemento` con el elemento en cuestión, y que devuelve una cadena de caracteres con los datos del elemento.

- `private String sGuardarTipoUnion (clsElemento oElemento)`

  Devuelve una cadena de caracteres que contiene el tipo de elemento, el nombre, la posición en pantalla y el nombre de la subclase del tipo unión. A continuación, concatena el número de superclases del tipo unión y recorre el vector que tiene el tipo unión con el nombre de las superclases para guardarlos en la cadena de caracteres que contiene los datos. Por último concatena un valor para indicar si las llaves de las superclases son iguales o no. La última acción de la función es devolver la cadena de caracteres construida con los datos del tipo unión.

o Si se trata de un tipo intersección, llama a la función `sGuardarTipoInterseccion()`,a la que se le pasa un objeto de la clase `clsElemento` con el elemento en cuestión, y que devuelve una cadena de caracteres con los datos del elemento.

- `private String sGuardarTipoInterseccion (clsElemento oElemento)`

  Devuelve una cadena de caracteres que contiene el tipo de elemento, el nombre, la posición en pantalla y el nombre de la subclase del tipo unión. A continuación, concatena el número de superclases del tipo unión y recorre el vector que tiene el tipo unión con el nombre de las superclases para guardarlos en la cadena de caracteres que contiene los datos. La última acción de la función es devolver la cadena de caracteres construida con los datos del tipo intersección.

A continuación se comenta la función `sGuardarAtributo()` que se invoca en las funciones `sGuardarEntidad()` y `sGuardarRelacion()`:

- `private String sGuardarAtributo (clsElemento oElemento)`

  Devuelve una cadena de caracteres que contiene el tipo de elemento, el nombre y el dominio del atributo. A continuación, se guardan las restricciones y el valor por defecto del atributo (si existen). Por último, guarda los atributos del atributo y el comentario del atributo del mismo modo explicado para las entidades. Finalmente, devuelve la cadena de caracteres construida con los datos del atributo.

- `public void Abrir(String psDatos)`

Este procedimiento recibe una cadena con todo el contenido de un fichero que contiene un esquema EER, y construye la estructura de datos necesaria para contener dicho esquema, pero no lo pinta en pantalla.

En primer lugar, divide la cadena recibida, `psDatos` , utilizando como carácter para dividirla el espacio en blanco que separa el valor de las propiedades de los elementos. El resultado lo almacena en una cadena de cadenas de caracteres. Posteriormente, inicializa una variable de tipo `int` que utiliza como índice para recorrerla. A continuación, llama a la función `nLeerConfiguracion()`, a la que le pasa este índice como parámetro y que se comenta a continuación:

- `private int nLeerConfiguracion (int piIndice)`

  Esta función recibe el índice actual de la cadena con las propiedades de los elementos del esquema y actualiza con ellos los valores de las variables de configuración de la clase `clsConfig`. Devuelve el índice de la primera propiedad del siguiente elemento del esquema a crear.

A continuación, mientras haya datos que leer de la cadena, se llama a la función `nLeerElemento()`, a la que le pasa el índice como parámetro y que se comenta a continuación:

- `private int nLeerElemento (int piIndice)`

  Esta función recibe el índice actual de la cadena con las propiedades de los elementos del esquema. En primer lugar, lee el tipo de elemento que se está recuperando para llamar a la función que construye ese tipo de elemento con los valores obtenidos del fichero. Estas funciones son `nLeerEntidad()`, si el tipo de elemento es una entidad o una entidad débil, `nLeerRelacion()`, si el tipo de elemento es una relación o una relación identificativa, `nLeerEspecializacion()`, si el tipo de elemento es una especialización, `nLeerTipoUnion()`, si el tipo de elemento es un tipo unión y `nLeerTipoInterseccion()`, si el tipo de elementos es un tipo intersección. Estas funciones reciben el índice de la primera propiedad del elemento en la cadena de cadenas de caracteres, y devuelven el índice de la primera propiedad del siguiente elemento a leer. Cada función crea un objeto de la clase `clsElemento` y lee las propiedades de los elementos de forma análoga a como se crean las cadenas con las propiedades de los mismos y que se ha comentado en este mismo apartado. Los valores del objeto creado para el elemento se van actualizando a medida que se van leyendo las propiedades. Por último, este objeto se añade a la variable que contiene todos los elementos del esquema, `vElementos`.

- `public String sObtenerFicheroEsquema(int pnOpcion)`

  Esta función muestra una ventana de diálogo para indicar un fichero del que leer un esquema o donde guardar el esquema actual (según la opción indicada a través del parámetro `pnOpcion`) y devuelve una cadena con la ruta completa de este fichero.

- `public String sObtenerFicheroScript()`

  Esta función muestra una ventana de diálogo para indicar un fichero donde guardar el *script* que genera la base de datos que se corresponde con el esquema actual. Devuelve una cadena de caracteres con la ruta completa de este fichero para que la utilice el método que guarda el *script* en un fichero.

En la Figura 4.6 se muestra el diagrama UML de la clase. El diagrama muestra las variables de clase (`gsDatos`, `gsPath`, `oConfig`, `oGeneral` y `oMensajes`) y los métodos que posee y que se han comentado anteriormente. Esta clase hereda de `Object`, y hace referencia a los componentes utilizados indirectamente, como son `Frame` y `FrameDialog`. El diagrama también muestra referencias a las clases `clsGeneral`, `clsConfig y clsMensajes`, debido a que la clase utiliza objeto que son instancias de estas clases para utilizar los métodos que proporcionan. Las flechas entrante con origen en la clase Derytas se debe a que esa clase crea un objeto que es instancia de la clase `clsFichero`.

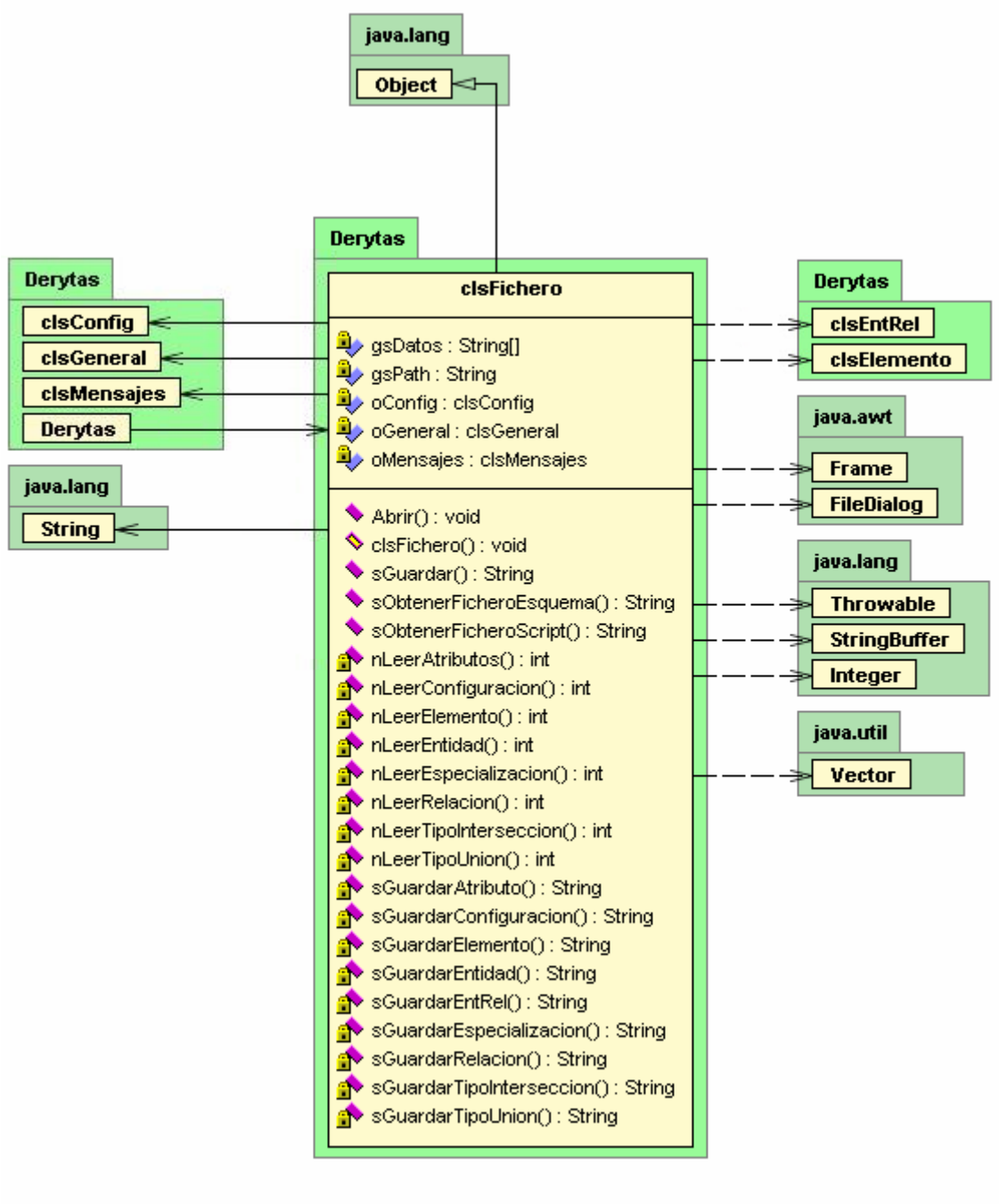


Figura 4.6: Diagrama UML de la clase clsFichero

4.5.7 Clase `clsGraficos.java`

Esta clase contiene los métodos necesarios para pintar los elementos del esquema en el *applet*. A continuación se exponen algunos de los métodos más importantes de la clase. En el código se incluyen comentarios aclarando todos los métodos.

- `private double dAngulo(int pnXOrigen,`
  `int pnYOrigen,`
  `int pnXDestino,`
  `int pnYDestino)`

  Esta función devuelve el ángulo, en grados, formado por la recta que va del punto indicado por (`pnXOrigen, pnYOrigen`) al punto (`pnXDestino, pnYDestino`) con la horizontal.

- `public void TrazarLinea(Graphics g,`
  `int pnXOrigen,`
  `int pnYOrigen,`
  `int pnAnchoOrigen,`
  `int pnAltoOrigen,`
  `int pnXDestino,`
  `int pnYDestino,`
  `int pnAnchoDestino,`
  `int pnAltoDestino,`
  `boolean pbInclusion)`

  Este procedimiento dibuja una recta entre un elemento origen y otro elemento destino del esquema, utilizando para ello una variable de tipo `Graphics`. El procedimiento recibe las coordenadas de la posición superior izquierda, la anchura y la altura de cada elemento. Con el parámetro `pbInclusion` se indica si se dibuja el símbolo de inclusión sobre esta línea.

- ```
  public void TrazarLinea(Graphics g,
                          int pnXOrigen,
                          int pnYOrigen,
                          int pnAnchoOrigen,
                          int pnAltoOrigen,
                          int pnXDestino,
                          int pnYDestino,
                          int pnAnchoDestino,
                          int pnAltoDestino,
                          String psMin,
                          String psMax)
  ```

  Este procedimiento dibuja una recta entre un elemento origen y otro elemento destino del esquema, utilizando para ello una variable de tipo `Graphics`. El procedimiento recibe las coordenadas de la posición superior izquierda, la anchura y la altura de cada elemento. Este procedimiento se utiliza para pintar rectas entre relaciones y entidades participantes en la relación, por lo que muestra sobre la línea recta el valor de las cadenas `psMin` y `psMax`, que son las caridinalidades mínima y máxima con las que participa la entidad en la relación.

- ```
  public void PintarEntidad(Graphics g,
                            Color c,
                            int pnX,
                            int pnY,
                            String psNombre,
                            int pnVerAtributos,
                            Vector pvAtributos,
                            boolean pbAtribIzq)
  ```

  Este procedimiento dibuja una entidad, así como los atributos de la misma (si tiene atributos), utilizando para ello una variable de tipo `Graphics`. El procedimiento recibe el color con el que se va a pintar la entidad, las coordenadas de la posición superior izquierda (`pnX` y `pnY`), el nombre de la entidad (`psNombre`), la opción de configuración elegida para visualizar los atributos (`pnVerAtributos`), un

parámetro de tipo `Vector` que contiene los atributos de la entidad (`pvAtributos`) y un parámetro booleano (`pbAtribIzq`) que indica si los atributos se pintan a partir de la parte izquierda de la entidad (cuando vale `true`) o a partir de la parte derecha de la entidad (cuando vale `false`).

4.5.8 Clase `clsGeneral.java`

Esta clase contiene métodos y propiedades que son utilizados por las otras clases de la aplicación. No responden a una funcionalidad común, por lo que contiene procedimientos y funciones con distintos propósitos. A continuación se muestran las propiedades y métodos más importantes:

- `vElementos`: Variable de tipo `Vector` para almacenar objetos de tipo `clsElemento` y contener los elementos del esquema. Su tamaño es variable en tiempo de ejecución, y existe límite para el número de objetos que puede contener

- `bVentanaAbierta`: Variable de tipo `boolean` que indica si se está mostrando alguna ventana. Con esta variable se controla que sólo haya una ventana visualizándose en cada momento, de manera que cuando se abre una ventana esta variable toma el valor `true`. Cada ventana es responsable de asignar el valor `false` a esta variable cuando se cierra. Cuando se quiere abrir una ventana, en primer lugar se comprueba el valor de esta variable, y, si vale `true`, la ventana no se abre.

- `nXInicial` y `nYInicial`: Variables de tipo `int` con la posición horizontal y vertical, respectivamente, donde se van a pintar los elementos al crearlos. Esta posición será aquella donde se haya hecho *click* por última vez con el ratón (siempre que no haya ya otro elemento en esa posición). Si no se ha hecho *click* con el ratón, se asume que esa posición es la superior izquierda del área de dibujo.

- `oElementoActual`: Objeto de la clase `clsElemento` con el elemento del esquema sobre el que se ha pulsado el ratón.

- `public boolean bEnElemento(int pnX,`
  `int pnY,`
  `boolean pbReferencia)`

  Función que recibe la posición horizontal y vertical de la última pulsación del ratón y devuelve un valor lógico indicando si se ha hecho *click* sobre un elemento del esquema. También recibe una variable de tipo `boolean (pbReferencia)` que indica si debe actualizar la variable `oElementoActual` con el elemento encontrado.

- `public boolean bExisteElemento(String psNombre,`
  `int pnTipo,`
  `Vector pvElementos)`

  Función que recibe una cadena de cartacteres, `psNombre`, que representa el nombre de un elemento, un valor numérico, `pnTipo`, que indica uno de los tipos del esquema (entidad, entidad débil, relación, relación identificativa, especialización, tipo unión, tipo intersección o atributo) y un `Vector` que contiene elementos del esquema. Devuelve un valor lógico indicando si alguno de los elementos que contiene el `Vector` tiene el mismo nombre y es del mismo tipo que los que ha recibido como parámetro. Si el tipo de elemento es un atributo, también se busca en el vector de atributos de dicho elemento.

- `public clsElemento elmElemento(String psNombre)`

  Función que recibe una cadena de caracteres, `psNombre`, que representa el nombre de un elemento y devuelve el objeto de la clase `clsElemento` almacenado en `vElementos` con ese nombre. Dado que la función `bExisteElemento()`, que se invoca cada vez que se crea un elemento del esquema, no permite que se repita el mismo nombre para dos elementos del esquema, basta con indicar el nombre para obtener el elemento deseado.

- `public void Repintar()`

  Procedimiento que llama al método `repaint()` del *applet* con el fin de actualizar el esquema dibujado cuando se modifica o elimina algún elemento del mismo.

- `public void LimpiarPantalla()`

  Este procedimiento elimina todos los elementos de `vElementos` y llama al método `Repintar()` para actualizar el esquema que se muestra en pantalla.

- `public String sFiltrarNombre(String psNombre)`

  Función que recibe una cadena de caracteres, `psNombre`, que representa un nombre y devuelve la misma cadena pero eliminando caracteres no deseados y sustituyendo unos por otros para que no se produzcan errores de sintaxis cuando sean utilizados para crear la base de datos.

4.5.9 Clase `clsConfig.java`

Esta clase permite configurar, mediante una ventana, el comportamiento de la aplicación y diversos parámetros que se pueden ajustar a la hora de crear el *script* de generación de la base de datos, como se muestran a continuación:

- `nVerAtributos`: indica el nivel de la jerarquía de atributos que se van a visualizar en el esquema, de manera que se muestran todos, ninguno o todos los atributos exceptuando los atributos compuestos, de los que sólo se muestra el primero de ellos.

- `nOpcionEsp`: indica la opción que se va a utilizar para generar el código de las especializaciones. Ver apartado 3.4.

- `bCerrar`: indica si las ventanas se cierran o permanecen abiertas después de crear o modificar un elemento del esquema.

- `bComentScript`: indica si se añadirán comentarios en el *script* de generación de la base de datos.

- `bComentDic`: indica si el *script* de generación de la base de datos incorporará el código necesario para insertar comentarios en el diccionario de la base de datos, utilizando la orden `COMMENT` explicada en la sección 3.5.1.

- `private void GuardarConfiguracion()`

  Procedimiento privado que actualiza las propiedades de la clase con los valores indicados en la ventana de opciones de configuración de la aplicación.

En la Figura 4.7 se muestra el diagrama UML de la clase. El diagrama muestra las variables de clase (`bCerrar`, `bComentDic`, `bComentScript`, `nOpcionEsp` y `nVerAtributos`) y los métodos que posee: `clsConfig()` (constructor), `Mostrar()`, `GuardarConfiguracion()` y `Ocultar()`. También muestra los componentes utilizados para construir la ventana de opciones de configuración y variables privadas utilizadas por la clase.

La clase hereda directamente de `Frame`, y hace referencia a los componentes utilizados, como son `Button`, `CheckBox`, `CheckboxGroup` y `Label`. Muestra los objetos que referenciados indirectamente, es decir, que no se crea explícitamente una instancia de ellos, de `java.awt` (para los gráficos) y de `java.awt.event` (para los eventos gráficos). La referencia a `Object` es debida a que todas los objetos heredan, directa o indirectamente de `Object`. El diagrama también muestra una referencia a la clase `clsGeneral`, debido a que es el parámetro del constructor. Las flechas entrantes en la clase indican que las clases de origen, o bien crean una instancia, o reciben un objeto de `clsConfig` como parámetro de su constructor.

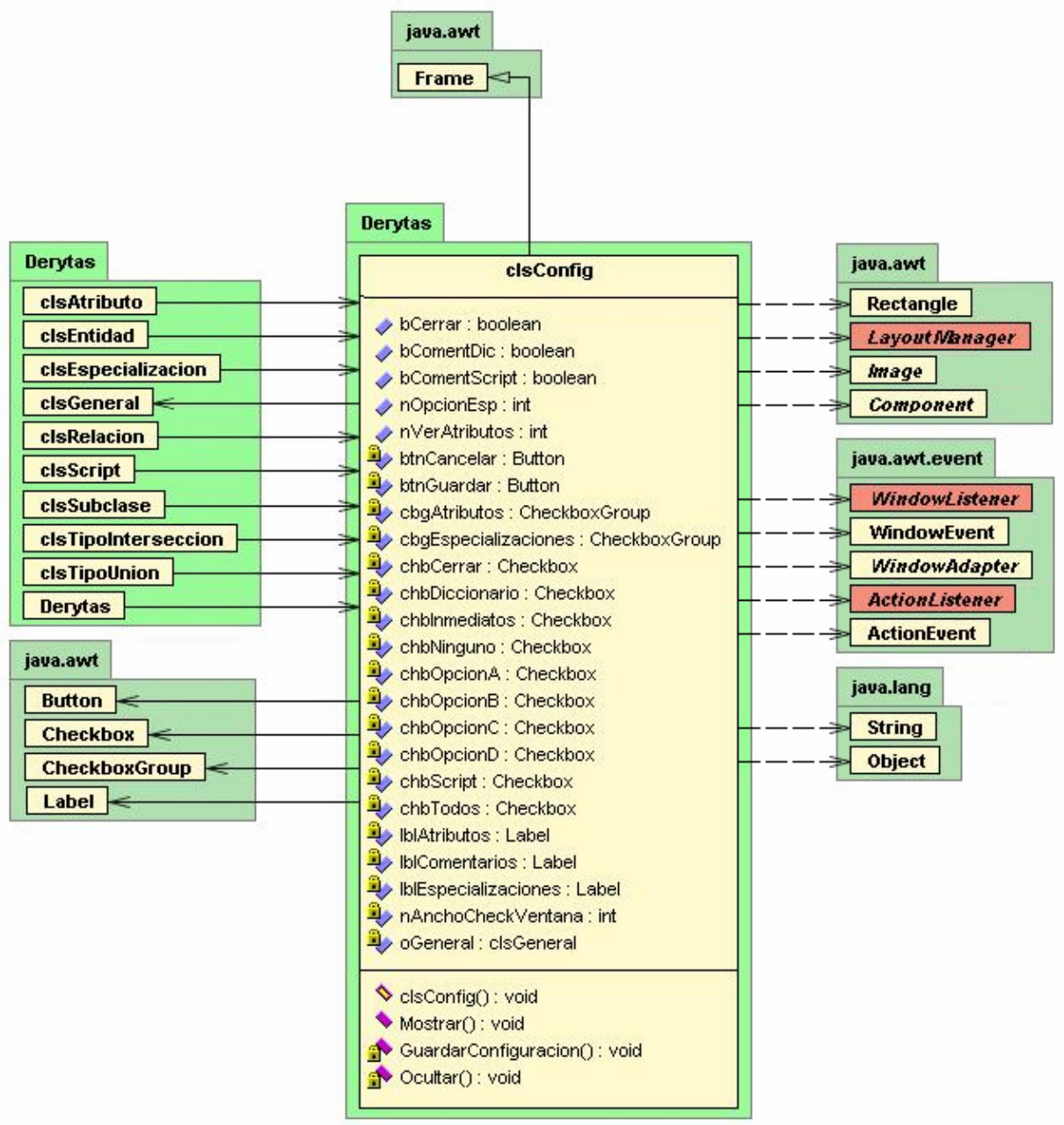


Figura 4.7: Diagrama UML de la clase clsConfig

4.5.10 Clase `clsScript.java`

Clase para crear el *script* de generación de la base de datos. Contiene una implementación del algoritmo que transforma un esquema entidad-relación extendido en un conjunto de tablas (ver el apartado 3.4). Además, genera el código en SQL de Oracle para crear las tablas y las relaciones entre ellas.

Contiene dos variables de tipo `String` para guardar el código del *script*: `gsTablas` y `gsReferencias`. La primera de ellas contiene el código correspondiente a las sentencias `CREATE TABLE` para crear las tablas de la base de datos. La segunda contiene las sentencias `ALTER TABLE` para crear las restricciones de clave externa en cada tabla. Esto se hace así para asegurar que existen todas las tablas cuando se crean las referencias a dichas tablas al crear las restricciones de llave externa.

El único método público de la clase, a partir del cual devuelve la cadena con el *script*, es la función `sGenerarScript()`, cuyo funcionamiento se expone a continuación:

- `public String sGenerarScript()`

  La función inicializa las variables de la clase `gsTablas` y `gsReferencias` asignándoles el valor "". Posteriormente, llama al procedimiento `GenerarScript()` (que se expone a continuación) que asigna el código correspondiente a las variables de la clase para, finalmente devolver la concatenación de dichas variables.

- `private void GenerarScript()`

  En primer lugar, llama al procedimiento `GenerarExtendidos()`, que genera el código para especializaciones, tipos unión, tipos intersección y subclases. A continuación, recorre el vector de elementos, `vElementos`, comprobando el tipo de elemento para llamar al método correspondiente. Si se trata de una entidad que no pertenece a una especialización, o a un tipo unión, o a un tipo intersección y que no es superclase ni subclase de otra entidad, llama al procedimiento `GenerarEntidad()`. Para relaciones no identificativas con cardinalidad dos, llama al procedimiento `GenerarRelacion_NM()`. Para relaciones con cardinalidad mayor que dos, llama al procedimiento `GenerarRelacionMasDos()`. Para relaciones identificativas llama al procedimiento `GenerarRelacionPropietaria()`. Por último, para cada atributo multivaluado, se llama al método `GenerarAtributoMultivaluado()`.

- `private void GenerarExtendidos()`

  Este procedimiento recorre el vector de elementos, `vElementos`, comprobando el tipo de elemento para llamar al método correspondiente. Si se trata de una entidad que no pertenece a una especialización, o a un tipo unión, o a un tipo intersección pero que es superclase o es subclase de otra entidad, llama al procedimiento `GenerarSubclase()`. Para cualquier tipo de especialización llama al procedimiento `GenerarEspecializacion()`, para cualquier tipo unión llama al procedimiento `GenerarUnión()` y, finalmente, para cualquier tipo intersección, llama al procedimiento `GenerarIntersección()`.

Todos los procedimientos que se llaman en `GenerarScript()` y `GenerarExtendidos()` reciben como argumento un objeto de tipo `clsElemento`, que es el objeto para el que se quiere generar el código. El funcionamiento de estos procedimientos se basa en un sistema de plantillas (representadas como constantes en la clase `clsConstantes`). Así, cuando según el algoritmo de traducción (ver apartado 3.4), se debe crear una tabla con el comando `CREATE TABLE`, se añade a la variable `gsTablas` la constante correspondiente para el elemento que se está tratando. De forma análoga, cuando en una tabla hay un atributo que es clave externa, se añade a la variable `gsReferencias` la constante correspondiente para modificar la tabla y añadir la restricción `FOREIGN KEY`. Estas plantillas tienen unas marcas que, una vez que se han añadido a las variables de la clase, se van sustituyendo por los valores correspondientes del elemento que se está tratando, o incluso por otras plantillas, como ocurre para el caso de los atributos o la restricción de clave primaria de las entidades.

4.5.11 Clase `clsElemento.java`

Clase para contener un elemento del esquema. Es una de las clases más importantes de toda la aplicación, pues contiene lo necesario para contener cualquier elemento del esquema y los métodos necesarios para gestionarlos.

Hay propiedades que son comunes a todos los elementos del esquema, pero otras son características de un elemento concreto. En principio se pensó crear un tipo de datos distinto

para cada elemento del esquema, pero como eso complicaría la gestión de los mismos, se optó por crear un único tipo representado mediante esta clase.

Las propiedades comunes a todos los elementos son las siguientes:

- `sNombre`: Variable de tipo `String` para guardar el nombre del elemento.

- `nTipo`: Variable de tipo `int` para guardar el tipo de elemento (entidad, entidad débil, relación, relación identificativa, especialización disjunta parcial, especialización disjunta total, especialización solapada parcial, especialización solapada total, tipo unión parcial, tipo unión total, tipo intersección parcial o tipo intersección total).

- `nX` y `nY`: Variables de tipo `int` para guardar la posición horizontal y vertical, del punto superior izquierdo del elemento, respectivamente. Estas variables se actualizan con la posición donde se arrastra el elemento con el ratón.

- `nAncho` y `nAlto`: Variables de tipo `int` para guardar el ancho y el alto del elemento, respectivamente.

A continuación se exponen los métodos de la clase:

- ```
  public void CambiarPos(int pnX,
                         int pnY)
  ```

  Este procedimiento actualiza el valor de las coordenadas de la posición del elemento. Actualiza el valor de `nXAntigua` y `nYAntigua` con los valores de `nX` y `nY` respectivamente, para asignar posteriormente a éstas últimas los dos valores de tipo `int` recibidos como parámetros.

- ```
  public void Pintar(Graphics g)
  ```

  Pinta la representación del elemento en el esquema a través de la llamada al método correspondiente de la clase `clsGraficos`.

- `public void PintarLineas(Graphics g)`

  Pinta las líneas existentes entre el elemento llamante y aquellos con los que tiene algún tipo de relación.

4.5.12 Clase `Derytas.java`

Clase para contener el esquema Esta clase es el *applet* de la aplicación, donde se dibuja el esquema. Supone el verdadero motor de la aplicación, ya que es el punto de conexión entre todas las demás clases y la página HTML con las funciones JavaScript.

Tiene las siguientes propiedades:

- `nXRatonAnt` y `nYRatonAnt`: Variables de tipo `int` que contienen la última posición donde se pulsó el botón izquierdo del ratón.

- `nXRatonAct` y `nYRatonAct`: Variables de tipo `int` que contiene la posición actual del ratón.
- popMenu: Variable de tipo PopupMenu. Es el menú contextual de la aplicación que nos permite realizar ciertas acciones según el elemento sobre el que se muestra.

Con los objetos `oConfig`, `oGeneral`, `oMensajes`, `oFichero` y `oScript` se tienen instancias de las clases `clsConfig`, `clsGeneral`, `clsMensajes`, `clsFichero` y `clsScript` respectivamente, de manera que se pueden utilizar sus métodos y sus propiedades cuando sea necesario.

Para mostrar la ventana de configuración, la clase proporciona el siguiente método:

- `public void AbrirConfig()`

  Este procedimiento llama al método `Mostrar()` del objeto `oConfig` y que muestra la ventana de configuración de la aplicación.

Para mostrar un esquema guardado en fichero y para guardar en un fichero el esquema que se está visualizando, la clase proporciona los siguientes métodos, respectivamente:

- `public void Abrir(String psDatos)`

  Recibe una cadena de caracteres, `psDatos`, con el contenido del fichero que almacena un esquema EER. Este procedimiento limpia la estructura de datos mediante la llamada al procedimiento `LimpiarPantalla()` del objeto `oGeneral`. Posteriormente llama al método `Abrir()` del objeto `oFichero`, al que le pasa la cadena recibida con el contenido del fichero para que cree la estructura de datos del esquema. Por último, invoca al método *repaitn()* del *applet*, para que actualice la pantalla con los datos del esquema que se han leído del fichero.

- `public String sGuardar()`

  Esta función devuelve la cadena de caracteres que obtiene tras llamar a la función `sGuardar()` del objeto `oFichero` que devuelve una cadena de caracteres con el contenido del esquema con el formato adecuado para ser guardado en fichero.

Para poder crear los elementos del esquema, la clase proporciona los siguientes métodos:

- `public void AbrirEntidad()`
- `public void AbrirSubclase()`
- `public void AbrirEspecializacion()`
- `public void AbrirTipoUnion()`
- `public void AbrirTipoInterseccion()`

- `public void AbrirRelacion()`

Cada uno de estos métodos crea una instancia de la clase correspondiente al elemento que se quiere crear, comprobando antes si ya hay una ventana abierta, caso en el que se muestra dicha ventana para que el usuario tenga conocimiento de ello.

Con el fin de modificar los elementos del esquema, la clase dispone del siguiente método:

- `public void Modificar()`

  Este procedimiento muestra la ventana correspondiente al elemento del esquema EER que se quiere modificar. Si ya hay abierta una ventana, no crea la instancia para mostrar la ventana del elemento que se quiere modificar, sino que muestra la ventana que ya esté abierta para que el usuario tenga conocimiento de ello.

La clase permite que se eliminen los elementos del esquema gracias a los métodos `EliminarElemento()` y `EliminarComoSubclase()`, que se muestran a continuación:

- `public void EliminarElemento()`

  Este procedimiento se utiliza para eliminar cualquier elemento del esquema (menos los atributos, que no se permite eliminarlos directamente, sino a través del elemento al que pertenecen). Crea una instancia de la clase `clsEliminarElemento`, que pide confirmación al usuario, y contiene la lógica para eliminar el elemento del esquema.

- `public void EliminarComoSubclase()`

  Este procedimiento se utiliza cuando se quiere romper la relación superclase-subclase entre dos entidades. Crea una instancia de la clase

`clsEliminarSubclase`, que pide confirmación al usuario, y contiene la lógica para romper esta relación entre entidades.

Por último, destacar que dentro de `Derytas.java` está implementada la clase `clsRaton`, cuya finalidad es la de gestionar los eventos del ratón que nos interesan para el funcionamiento de la aplicación. Por ejemplo, habilita y deshabilita las opciones del menú contextual, según si el ratón se ha pulsado o no sobre un elemento del esquema (y según qué tipo de elemento). También actualiza la posición donde se posicionará un nuevo elemento e indica si se está moviendo algún elemento del esquema.

Capítulo 5

Derytas - Manual de Usuario

En este capítulo se muestra toda la información necesaria para el uso de Derytas, tanto para diseñar el esquema EER de una base de datos, como para generar el conjunto de sentencias en SQL (en adelante, *script*), a partir del cuál se crea la base de datos físicamente cuando éste se ejecuta en un sistema gestor de bases de datos (en adelante, SGBD). Se empieza con una visión general, continuando por la creación y conexión entre los elementos del esquema para terminar en la generación del *script*.

5.1 Requisitos Técnicos

El único requisito para utilizar Derytas es tener un ordenador conectado a Internet en el que se tenga instalada la Máquina Virtual de Java (en inglés, Java Virtual Machine (JVM)). Las versiones anteriores a Windows XP ya la tenían instalada, pero debido a problemas legales Microsoft tuvo que retirarla y debe instalarse aparte. Los usuarios de Windows XP podrán encontrarla en la página de Sun Microsystems [W3SUN] (la descarga es gratuita). Aunque no se trata de un requerimiento, se aconseja utilizar Internet Explorer 6.0 y una configuración de pantalla de 1024x768.

5.2 Elementos de Derytas

Una vez que Derytas se ha cargado en el navegador, se dispone de 13 botones y un menú contextual (al que se accede pulsando el botón derecho del ratón) para explotar toda la funcionalidad que nos ofrece la aplicación. En la Figura 5.1 se muestra Derytas cuando ya se ha cargado en el navegador.

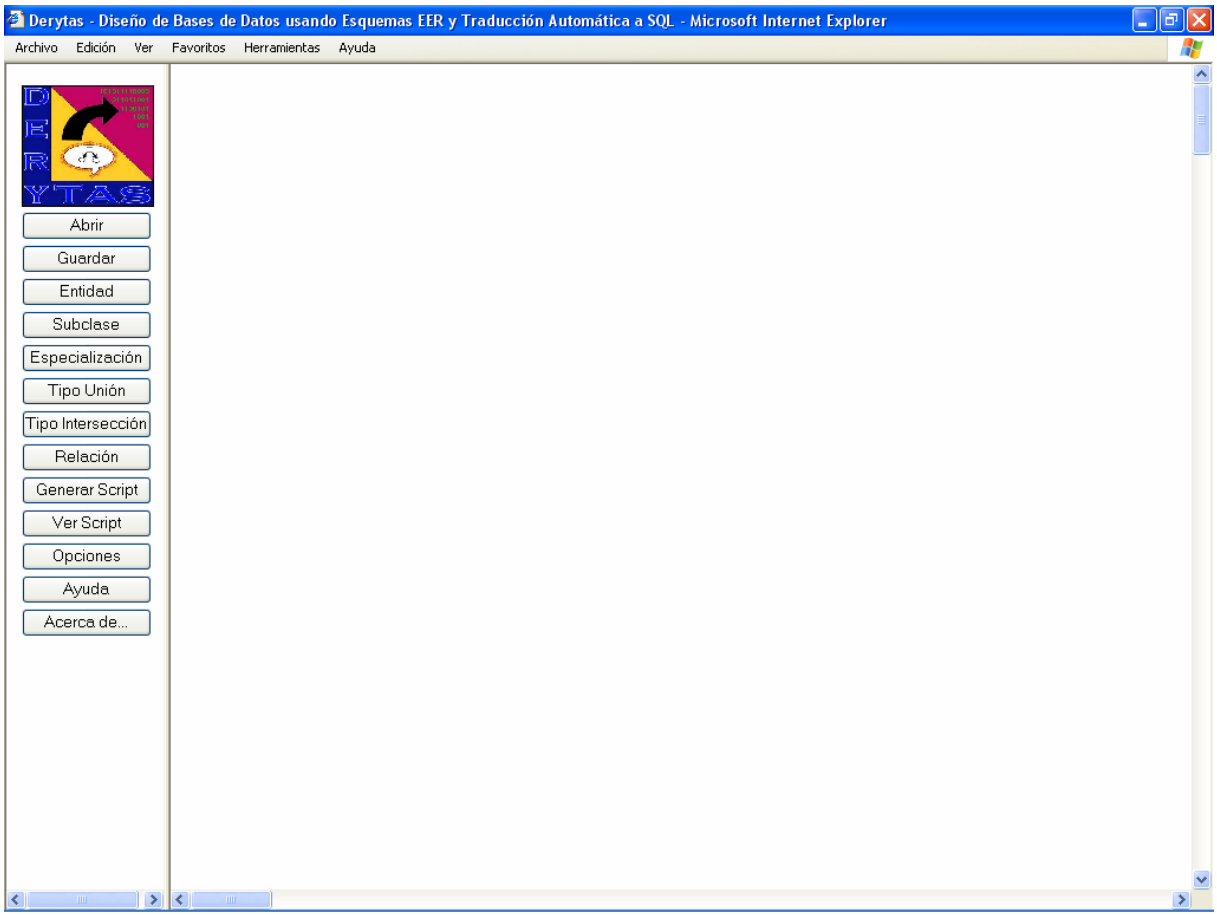


Figura 5.1: Derytas cargado en Internet Explorer

A continuación, se expone brevemente la funcionalidad que nos ofrece cada uno de los botones y cada una de las opciones del menú contextual, que estarán habilitadas o deshabilitadas, según el elemento sobre el que se ha hecho emerger el menú (o en una región vacía del esquema).

- **Botones** (Figura 5.1):
    - **Abrir** (apartado 5.13): permite abrir un esquema guardado anteriormente.
    - **Guardar** (apartado 5.12): permite guardar el esquema que se está creando.
    - **Entidad** (apartado 5.3): permite crear una nueva entidad para el esquema.
    - **Subclase** (apartado 5.4): permite crear una única subclase para una entidad del esquema.

- **Especialización** (apartado 5.5): permite crear una nueva especialización para el esquema.

- **Tipo Unión** (apartado 5.6): permite crear un nuevo tipo unión para el esquema.

- **Tipo Intersección** (apartado 5.7): permite crear un nuevo tipo intersección para el esquema, que es como Derytas modela las subclases compartidas.

- **Relación** (apartado 5.8): permite crear una nueva relación para el esquema.

- **Generar Script** (apartado 5.14): genera un fichero con el *script* para crear la base de datos.

- **Ver Script** (apartado 5.14): abre una ventana con el *script* que la aplicación va a generar para crear la base de datos.

- **Opciones** (apartado 5.15): abre la ventana de configuración de la aplicación.

- **Ayuda** (apartado 5.17): abre la ayuda de la aplicación.

- **Acerca de…** (apartado 5.16): abre una ventana que muestra información relacionada con Derytas.

- **Menú Contextual** (Figura 5.2):

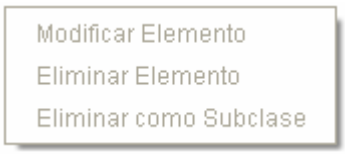


Figura 5.2: Menú contextual de Derytas

- **Modificar Elemento** (apartado 5.10): permite modificar el elemento sobre el que se ha pulsado el ratón para que aparezca el menú contextual. Sólo los

atributos de una entidad, de una relación, o de otro atributo (para el caso de los atributos compuestos) no pueden ser modificados.

- **Eliminar Elemento** (apartado 5.10.1): permite eliminar el elemento sobre el que se pulsado el botón derecho del ratón. Sólo los atributos de una entidad, de una relación, o de otro atributo (para el caso de los atributos compuestos) no pueden ser eliminados con esta opción del menú contextual.

- **Eliminar como Subclase** (apartado 5.10.1): permite eliminar la relación superclase-subclase entre dos entidades.

5.3 Creación de Entidades

Para crear entidades, se debe pulsar el botón **Entidad** (Figura 5.1), con lo que se abre la ventana de la Figura 5.3:

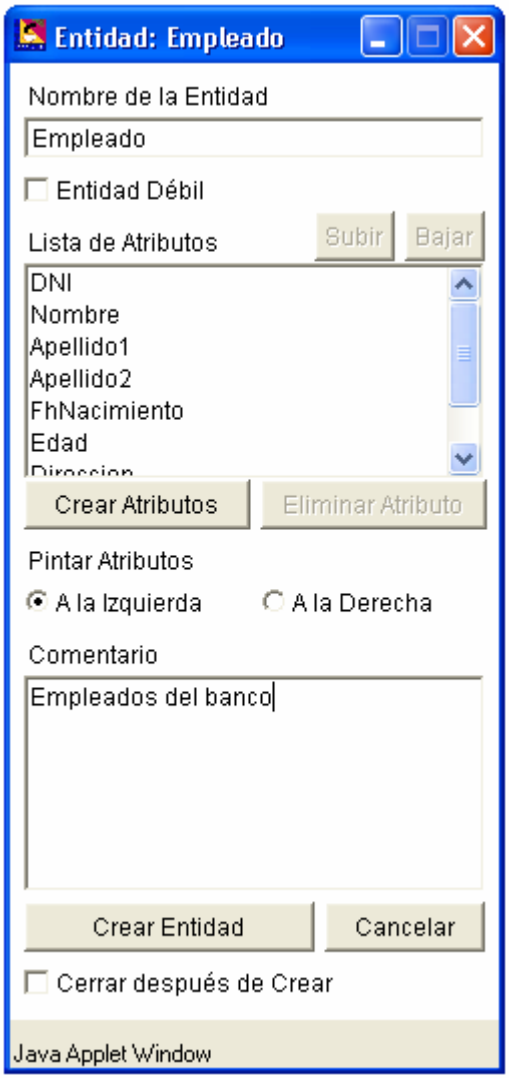


Figura 5.3: Ventana Nueva Entidad

- Campo de texto "Nombre de la Entidad": este campo se utiliza para asignar el nombre de la entidad. Como máximo puede tener 15 caracteres y en el título de la ventana se observa cómo Derytas lo va formateando (Figura 5.4), sustituyendo las vocales acentuadas por las mismas sin acentuar, y los espacios y los caracteres no alfanuméricos por "_", para evitar errores en los nombre de las tablas y de las columnas al ejecutar el *script* de creación de la base de datos. Cuando se abre una ventana, se ha limpiado tras crear una entidad o se borra el campo de texto del nombre, el título de la ventana es "Nueva Entidad, mientras que al escribir en el campo de texto del nombre, el título de la ventana cambia a "Entidad: *nombre formateado*". Este *nombre formateado* es el que Derytas va a dar a la entidad en el momento de su creación.

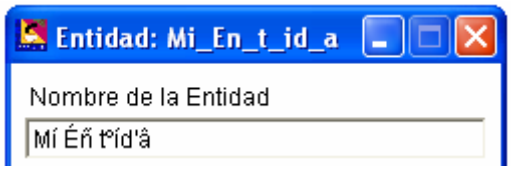


Figura 5.4: Filtrado del nombre de la entidad

- *Check* "Entidad Débil": se debe marcar para indicar que estamos creando una entidad débil y se tengan en cuenta las restricciones de las mismas al crear esta entidad.

- Botón "Crear Atributos": permite definir los atributos para la entidad que estamos creando. Para ello, se abre la ventana de la Figura 5.22, cuyo funcionamiento se explica en el apartado 5.9. Los atributos que vayamos definiendo para esta entidad van apareciendo en la lista que está sobre este botón, como se muestra en la Figura 5.3. El botón se habilita cuando se escribe el nombre de la entidad, permaneciendo deshabilitado si el nombre está vacío.

- Botón "Eliminar Atributo": se habilita cuando se selecciona un atributo de la lista de atributos (Figura 5.5), de manera que el atributo seleccionado es eliminado de la lista de atributos de la entidad. Si no se ha seleccionado ningún atributo, este botón permanece deshabilitado, como se puede observar en la Figura 5.3.

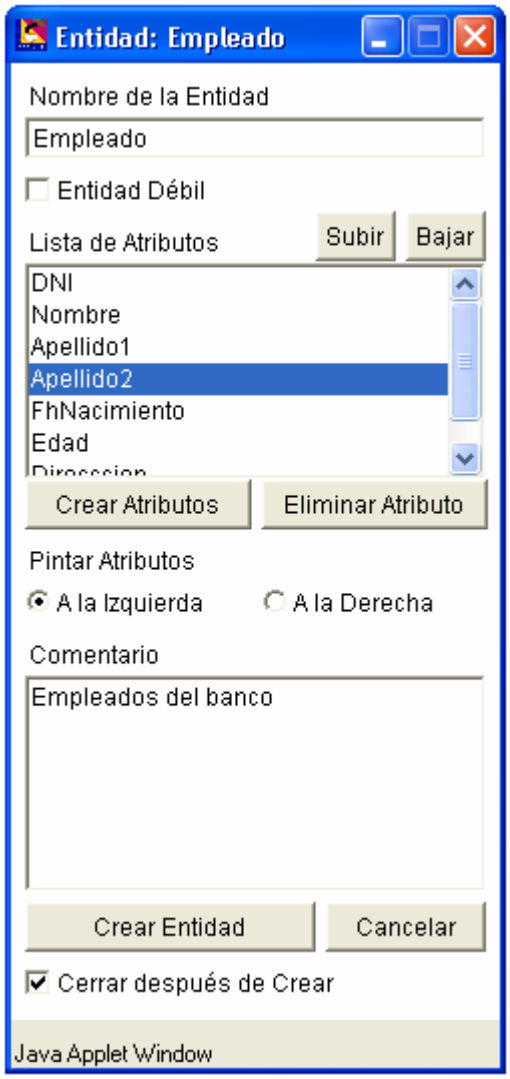


Figura 5.5: Botón Eliminar Atributo habilitado

- Lista "Lista de Atributos": contiene la lista de atributos de la entidad, como se puede ver en la Figura 5.5. Esta lista la podemos ordenar con los botones "Subir" y "Bajar" que se activan o desactivan al seleccionar un elemento de esta lista.

- Botón "Subir": se utiliza para reordenar la lista de atributos, subiendo el atributo seleccionado de la misma. El atributo indicado se mantiene seleccionado mientras no se seleccione otro, y el botón sigue habilitado mientras el atributo seleccionado no alcance la primera posición o no se seleccione el primer atributo de la lista.

- Botón "Bajar": se utiliza para reordenar la lista de atributos, bajando el atributo seleccionado de la misma. El atributo indicado se mantiene seleccionado mientras no

se seleccione otro, y el botón sigue habilitado mientras el atributo no alcance la última posición o no se seleccione el último atributo de la lista.

- Botones de radio “Pintar Atributos”: marcando “A la Izquierda”, se indica que los atributos se mostrarán a la izquierda de la entidad (Figura 5.6), mientras que marcando “A la Derecha”, los atributos se mostrarán a la derecha de la entidad (Figura 5.7).

- Campo de texto “Comentario”: este espacio está reservado para los comentarios acerca de la entidad que se está definiendo. Según la configuración elegida (apartado 5.15), este comentario aparecerá (o no) como un comentario del *script* y/o en una sentencia `COMMENT` (se explica en el apartado 3.5.1 del capítulo 3) para ser introducido en el diccionario de la base de datos.

- Botón “Crear Entidad”: crea una nueva entidad con todos sus atributos, de manera que ésta aparece ya en el esquema (Figuras 5.6 y 5.7) representada como un rectángulo con el nombre de la entidad en el interior y de donde cuelgan, en forma de árbol, los atributos de la misma. El botón no está habilitado mientras que no se escriba un nombre para la entidad y no se haya creado al menos un atributo para la entidad. Si la entidad está mal definida y no puede crearse, Derytas nos informa con un mensaje de error, indicando la causa del mismo (sección 5.10.1). Cuando se está modificando una entidad, este botón aparece como “Modificar Entidad”. La posición donde se colocará la entidad, en el momento de su creación, viene indicada por la posición de la última pulsación del botón izquierdo del ratón en la zona de dibujo (si no se ha pulsado ninguna vez, esta posición será la esquina superior izquierda de la zona de dibujo). Si en esa posición hay algún elemento, la entidad se desplazará hacia abajo y hacia la derecha, a partir de esa posición, hasta encontrar una zona sin ningún elemento.

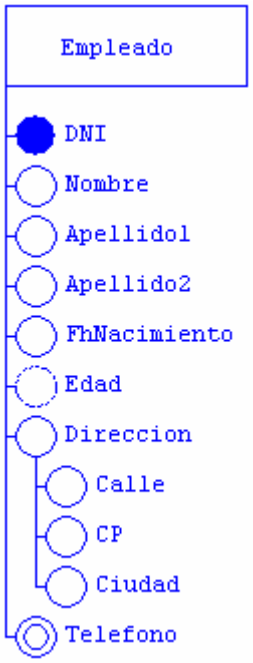


Figura 5.6: Ejemplo de entidad con atributos a la izquierda

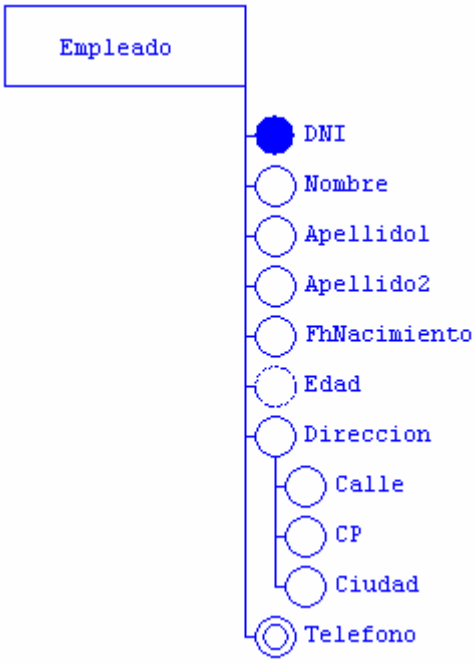


Figura 5.7: Ejemplo de entidad con atributos a la derecha

- Botón "Cancelar": cancela la creación de la entidad y cierra la pantalla.

- *Check* "Cerrar después de Crear": si está seleccionada, la ventana se cerrará después de dar de crear la entidad, en caso contrario, la ventana se limpia y se queda esperando la creación de otra entidad. Esto es útil por si se quieren crear varios elementos del mismo tipo, de forma que no haya que elegir la opción de creación cada vez. Cuando se está modificando una entidad, el título de esta *check* es "Cerrar después de Modificar".

  El valor de esta *check* se queda guardado en las variables de configuración, que es común a todas las ventanas de la aplicación, por lo que al modificar su estado (marcada o sin marcar) en una de las ventanas, todas las demás ventanas se ven afectadas por este cambio.

5.4 Creación de Subclases

En este apartado se explica cómo, a partir de dos subclases (una con llave primaria y la otra no) se puede hacer que una sea la subclase de otra (y por tanto, que una sea superclase de la otra). Un requisito impuesto por Derytas es que estas entidades no estén en una especialización, tipo unión, tipo intersección o ya sean superclase o subclase de otra entidad.

Derytas sólo permite una subclase para cada superclase y una entidad sólo puede ser superclase de una subclase.

Para crear una subclase se debe pulsar el botón **Subclase** (Figura 5.1), con lo que se abre la ventana de la Figura 5.8:

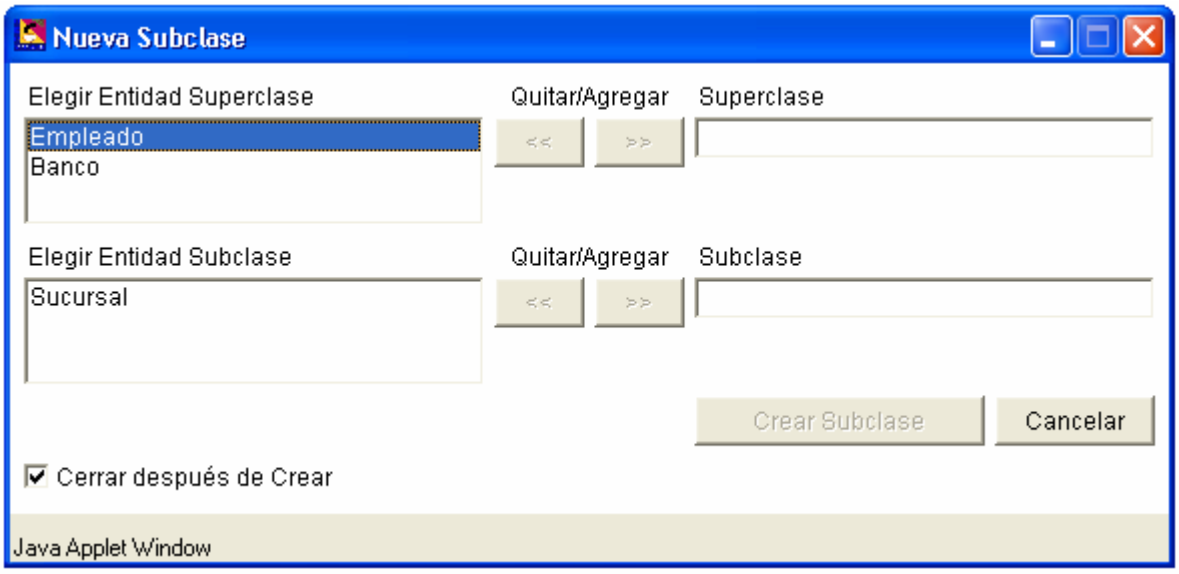


Figura 5.8: Ventana Nueva Subclase

- Lista "Elegir Entidad Superclase": en esta lista se muestran las entidades ya creadas, que tengan llave y que están disponibles (es decir, que no son ya superclases) para ser superclase de la entidad que indiquemos.

- Lista "Elegir Entidad Subclase": en esta lista se muestran las entidades ya creadas, sin llave y que están disponibles (es decir, que no son subclase de otra entidad) para ser la subclase de la entidad que se indique que va a ser la superclase.

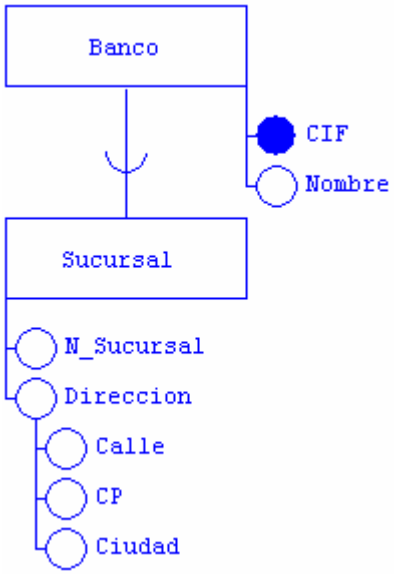
- Botones "Quitar/Agregar": Los botones ">>" se utiliza para agregar la entidad seleccionada de la lista "Elegir Entidad Superclase" o "Elegir Entidad Subclase" a la caja de texto "Superclase" o "Subclase". Con los botones "<<" se quita la entidad de la caja de texto "Superclase" o "Subclase" y se devuelve a la lista de la que procede. Los botones ">>" estarán habilitados mientras no se haya seleccionado la "Superclase" o la "Subclase" correspondientes, pasando a estar deshabilitados cuando se informen. Los botones "<<" estarán deshabilitados mientras no se haya

seleccionado la “Superclase” o la “Subclase” correspondientes, pasando a estar habilitados cuando se informen.

- Campo de texto “Superclase”: contiene la entidad elegida para ser superclase. Es un campo protegido, para que su contenido esté gestionado mediante los botones “Quitar/Agregar”.

- Campo de texto “Subclase”: contiene la entidad elegida para ser subclase. Es un campo protegido, para que su contenido esté gestionado mediante los botones “Quitar/Agregar”.

- Botón “Crear Subclase”: establece la relación superclase-subclase entre las entidades indicadas, representada en el esquema mediante una línea que las une con el símbolo de inclusión señalando la superclase (Figura 5.9). El botón no está habilitado mientras los campos “Superclase” y “Subclase” no estén informados.



5.9: Ejemplo de subclase

- Botón “Cancelar”: cancela la creación de la subclase y cierra la pantalla.

- *Check* “Cerrar después de Crear”: Su comportamiento es análogo al que se ha explicado en el apartado 5.3.

5.5 Creación de Especializaciones

Para crear especializaciones se debe pulsar el botón **Especialización** (Figura 5.1) con lo que se abre la ventana de la Figura 5.10:

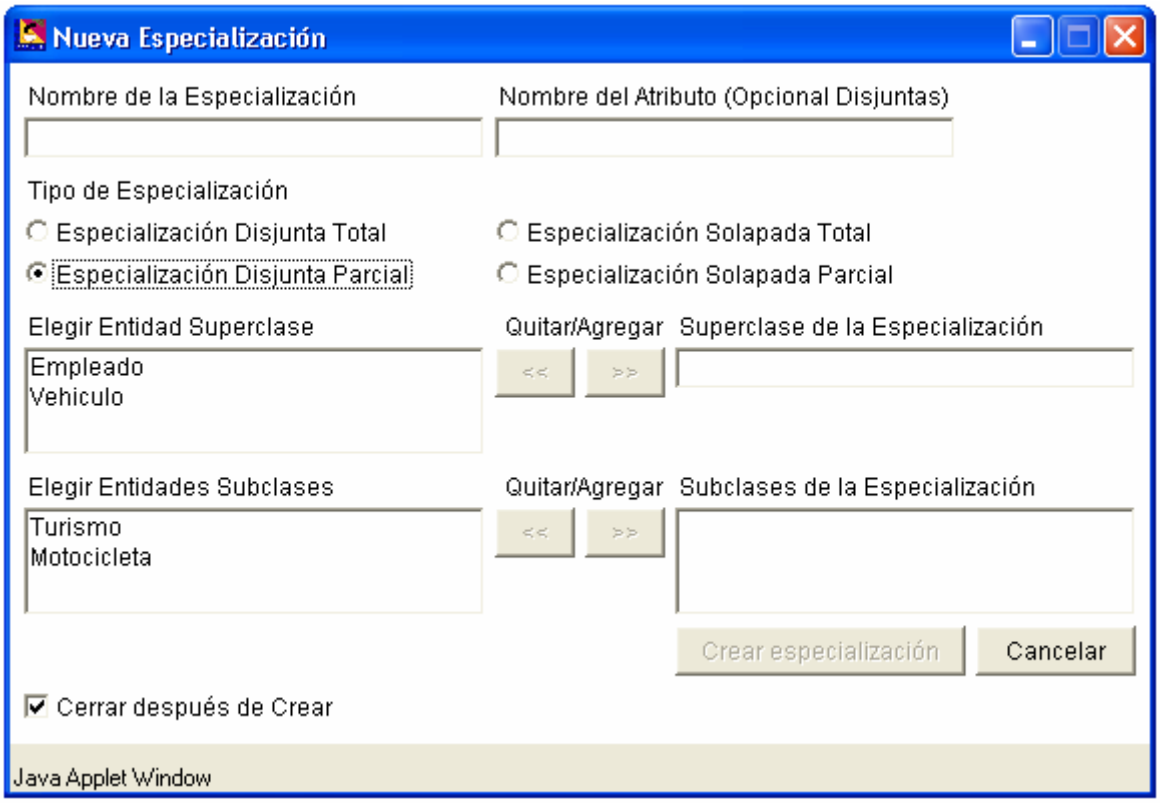


Figura 5.10: Ventana Nueva Especialización

- Campo de texto "Nombre de la Especialización": este campo se utiliza para asignar el nombre de la especialización. Como máximo puede tener 15 caracteres y en el título de la ventana se observa cómo Derytas lo va formateando (Figura 5.11), sustituyendo las vocales acentuadas por las mismas sin acentuar, y los espacios y los caracteres no alfanuméricos por "_", para evitar errores en los nombre de las tablas y de las columnas al ejecutar el *script* de creación de la base de datos. Cuando la se abre una ventana (o se ha limpiado tras crear una especialización) o se borra el campo de texto del nombre, el título de la ventana es "Nueva Especialización", mientras que al escribir en el campo de texto del nombre, el título de la ventana cambia a "Especialización: *nombre formateado*". Este *nombre formateado* es el nombre que Derytas utilizará internamente cuando sea necesario.

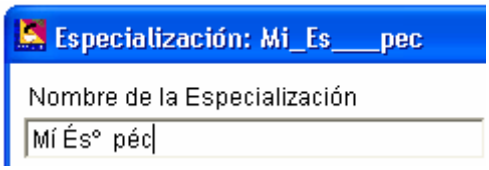


Figura 5.11: Filtrado del nombre de la especialización

- "Tipo de Especialización": mediante este grupo de botones de radio se indica el tipo de especialización que se quiere crear: Disjunta Total, Disjunta Parcial, Solapada Total o Solapada Parcial. En la sección 3.3.1 del capítulo 3 se explican las restricciones de las especializaciones.

- Campo de texto "Nombre del Atributo (Opcional Disjuntas)": en este campo se indica el nombre del atributo para las especializaciones definidas por atributo. Es un campo opcional y sólo disponible para especializaciones disjuntas (Figura 5.10), de manera que, cuando se genere el *script* utilizando la opción D para especializaciones, se creará una columna con el nombre indicado en este atributo, con dominio VARCHAR2(15). El motivo para que el dominio de este atributo sea VARCHAR2(15), es que se utiliza para indicar a qué subclase de la especialización disjunta pertenece cada elemento de la superclase (cuando exista tal pertenencia).

- Lista "Elegir Entidad Superclase": en esta lista se muestran las entidades ya creadas, que tengan llave y que están disponibles (es decir, que no son ya superclases) para ser superclase de la especialización.

- Lista "Elegir Entidades Subclases": en esta lista se muestran las entidades ya creadas, sin llave y que están disponibles (es decir, que no son subclases) para ser subclase de la especialización.

- Botones "Quitar/Agregar": Los botones ">>" se utilizan para agregar la entidad seleccionada de la lista "Elegir Entidad Superclase" o "Elegir Entidades Subclases" a la caja de texto "Superclase de la Especialización" o a la lista "Subclases de la Especialización". Con los botones "<<" se quita la entidad de la caja de texto "Superclase de la Especialización" o la entidad seleccionada en la lista "Subclases de la especialización", devolviéndola a la lista de la que procede. Los botones "<<"

estarán deshabilitados mientras no se haya seleccionado la “Superclase” o alguna “Subclase”, pasando a estar habilitados cuando se informen. El botón “>>” para superclase se habilita cada vez que se selecciona una entidad de la lista de entidades disponibles para superclases siempre que no se haya informado ésta, y el botón “>>” para subclases se habilita cada vez que se selecciona una entidad de la lista de entidades disponibles para subclases.

- Campo de texto “Superclase de la Especialización”: contiene la entidad elegida para ser superclase. Es un campo protegido, para que su contenido esté gestionado mediante los botones “Quitar/Agregar” correspondientes, que mueven las entidades entre este campo y la lista “Elegir Entidad Superclase”.

- Lista “Subclases de la Especialización”: contiene las entidades elegidas para ser subclase. El contenido de esta lista se controla mediante los botones “Quitar/Agregar” correspondientes, que mueven las entidades entre esta lista y la lista “Elegir Entidades Subclases”.

- Botón “Crear Especialización”: crea una nueva especialización con las entidades que pertenecen a ella, de manera que ésta aparece ya en el esquema (Figura 5.12) como un círculo del que parten líneas hacia la superclase y hacia las subclases. Si la especialización es disjunta, el círculo encierra una “d” (*disjoint*, en inglés), y si es solapada, una “o” (*overlapping*, en inglés). Para el caso de especializaciones definidas por atributo, éste aparece a la derecha de la representación de la especialización indicando su nombre. El botón no se habilita mientras no se haya indicado la superclase, más de una subclase y un nombre para la especialización. Si la especialización está mal definida y no puede crearse, Derytas nos informa con un mensaje de error (sección 5.10.2). Cuando se está modificando una especialización, este botón aparece como “Modificar Especialización”. La posición donde se colocará la especialización, en el momento de su creación, viene indicada por la posición de la última pulsación del botón izquierdo del ratón en la zona de dibujo (si no se ha pulsado ninguna vez, esta posición será la esquina superior izquierda de la zona de dibujo). Si en esa posición hay algún elemento, la especialización se desplazará hacia abajo y hacia la derecha, a partir de esa posición, hasta encontrar una zona sin ningún elemento.

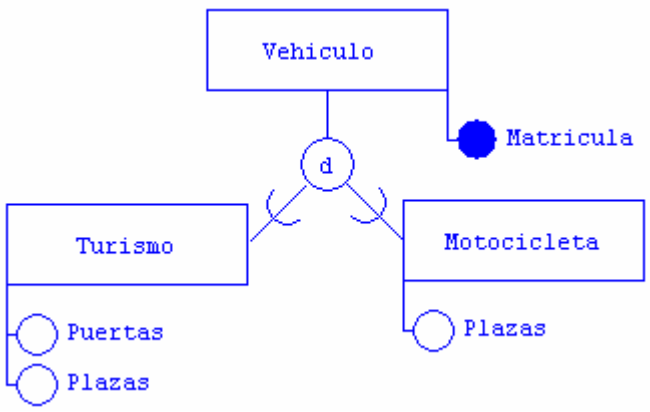


Figura 5.12: Ejemplo de especialización

- Botón "Cancelar": cancela la creación de la especialización y cierra la ventana.

- *Check* "Cerrar después de Crear": Su comportamiento es análogo al que se ha explicado en el apartado 5.3.

5.6 Creación de Tipos Unión

Para crear tipos unión se debe pulsar botón **Tipo Unión** (Figura 5.1), con lo que se abre la ventana de la Figura 5.13:

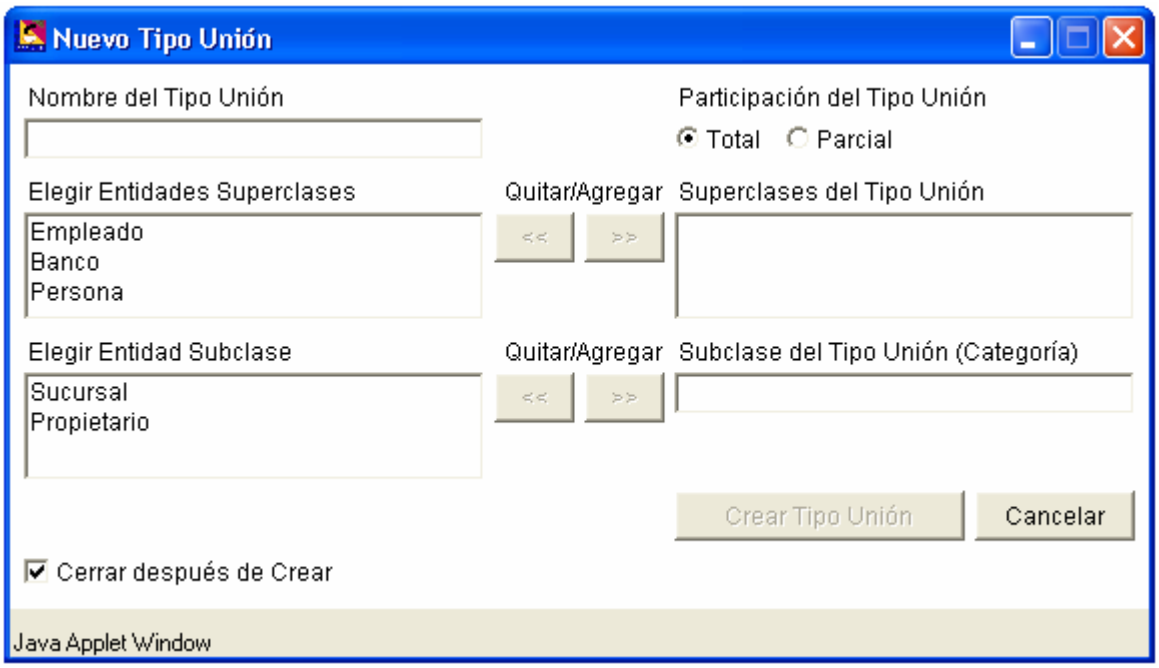


Figura 5.13: Ventana Nuevo Tipo Unión

- Campo de texto "Nombre del Tipo Unión": este campo se utilizar para asignar un nombre al tipo unión. Como máximo puede tener 15 caracteres y en el título de la ventana se observa cómo Derytas lo va formateando (Figura 5.14), sustituyendo las vocales acentuadas por las mismas sin acentuar, y los espacios y los caracteres no alfanuméricos por "_", para evitar errores en los nombre de las tablas y de las columnas al ejecutar el *script* de creación de la base de datos. Cuando se abre una ventana (o se ha limpiado tras crear una categoría) o se borra el campo de texto del nombre, el título de la ventana es "Nuevo Tipo Unión", mientras que al escribir en el campo de texto del nombre, el título de la ventana cambia a "Tipo Unión: *nombre formateado*".

Figura 5.14: Filtrado del nombre del tipo unión

- "Participación del Tipo Unión": mediante este grupo de botones de radio se indica la participación del tipo unión que se quiere crear: "Total" para indicar que cualquier elemento de una superclase debe pertenecer a la subclase, o "Parcial" para indicar que puede haber elementos de una superclase que no pertenezcan a la subclase.

- Lista "Elegir Entidades Superclases": en esta lista se muestran las entidades ya creadas, que tengan llave y que están disponibles (es decir, que no son ya superclases) para ser superclase en el tipo unión que se está creando.

- Lista "Elegir Entidad Subclase": en esta lista se muestran las entidades ya creadas, sin llave y que están disponibles (es decir, que no son subclase de otra entidad) para ser la subclase del tipo unión que se está creando.

- Botones "Quitar/Agregar": Los botones ">>" se utilizan para agregar la entidad seleccionada en la lista "Elegir Entidades Superclases" o "Elegir Entidad Subclase" a la lista "Superclases del Tipo Unión" o a la caja de texto "Subclase del Tipo Unión (Categoría)". Con los botones ">>" se quita la entidad seleccionada en la lista

"Superclases del Tipo Unión" o la entidad de la caja de texto "Subclase del Tipo Unión (Categoría)", devolviéndola a la lista de la que procede. Los botones "<<" estarán deshabilitados mientras no se haya informado alguna "Superclase" o la "Subclase" correspondientes, pasando a estar habilitados cuando se informen. El botón ">>" para superclase se habilita cada vez que se selecciona una entidad de la lista de entidades disponibles para superclases, y el botón ">>" para subclases se habilita cada vez que se selecciona una entidad de la lista de entidades disponibles para subclases siempre que no se haya informado ya ésta.

- Lista "Superclases del Tipo Unión": contiene las entidades elegidas para ser superclases del tipo unión. El contenido de esta lista se controla mediante los botones "Quitar/Agregar" correspondientes, que mueven las entidades entre esta lista y la lista "Elegir Entidades Superclases".

- Campo de texto "Subclase del Tipo Unión (Categoría)": contiene la entidad elegida para ser subclase del tipo unión (categoría). Es un campo protegido, para que su contenido esté gestionado mediante los botones "Quitar/Agregar".

- Botón "Crear Tipo Unión": crea un nuevo Tipo Unión con las entidades que pertenecen a ella, de manera que ésta aparece ya en el esquema en pantalla (Figura 5.15) representada como un círculo con el símbolo de unión en su interior y con líneas que salen hacia las superclases y hacia la subclase. El botón no se habilita mientras que no se haya indicado más de una superclase, la subclase y el nombre del tipo unión. Si el tipo unión está mal definido y no puede crearse, Derytas nos informa con un mensaje de error, indicando la causa del mismo (sección 5.10.3). Cuando se está modificando un tipo unión, este botón aparece como "Modificar Tipo Unión". La posición donde se colocará el tipo unión, en el momento de su creación, viene indicada por la posición de la última pulsación del botón izquierdo del ratón en la zona de dibujo (si no se ha pulsado ninguna vez, esta posición será la esquina superior izquierda de la zona de dibujo). Si en esa posición hay algún elemento, el tipo unión se desplazará hacia abajo y hacia la derecha, a partir de esa posición, hasta encontrar una zona sin ningún elemento.

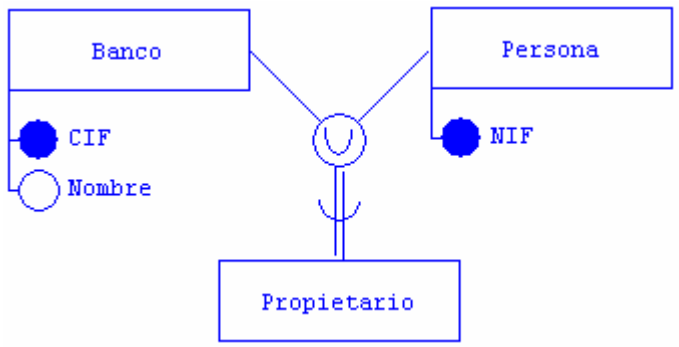


Figura 5.15: Ejemplo de tipo unión

- Botón "Cancelar": cancela la creación del tipo unión y cierra la ventana.

- *Check* "Cerrar después de Crear": Su comportamiento es análogo al que se ha explicado en el apartado 5.3.

5.7 Creación de Tipos Intersección

Para crear tipos intersección se debe pulsar el botón **Tipo Intersección** (Figura 5.1), con lo que se abre la ventana de la Figura 5.16:

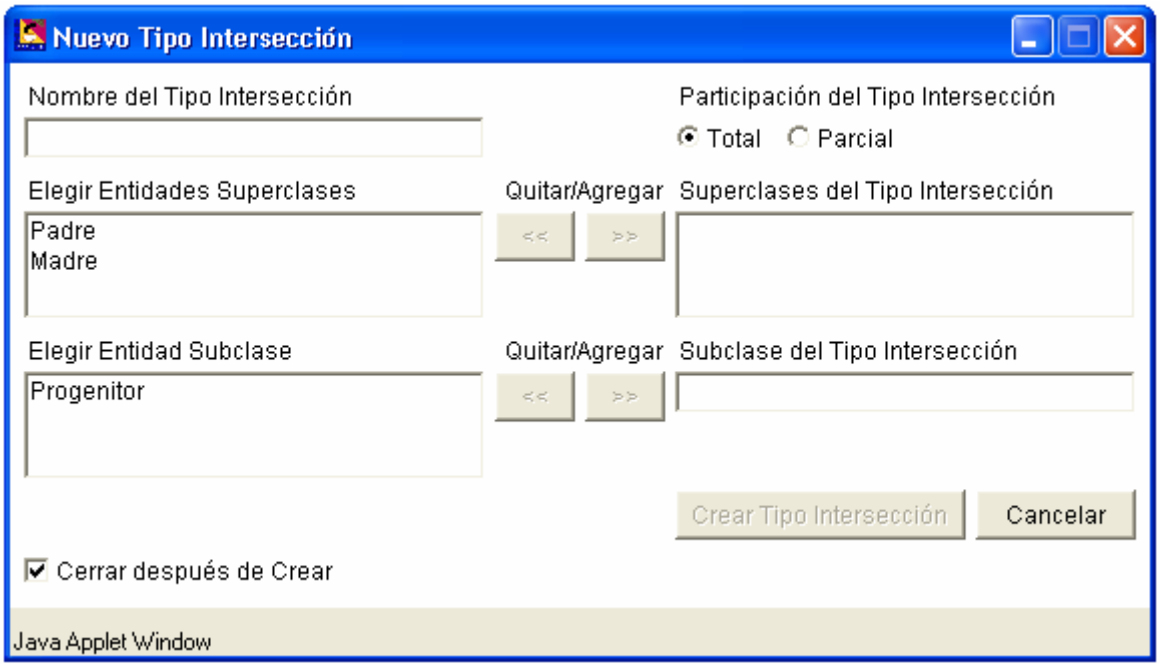


Figura 5.16: Ventana Nuevo Tipo Intersección

- Campo de texto "Nombre del Tipo Intersección": este campo se utilizar para asignar un nombre al tipo intersección. Como máximo puede tener 15 caracteres y en el título de la ventana se observa cómo Derytas lo va formateando (Figura 5.17), sustituyendo las vocales acentuadas por las mismas sin acentuar, y los espacios y los caracteres no alfanuméricos por "_", para evitar errores en los nombre de las tablas y de las columnas al ejecutar el *script* de creación de la base de datos. Cuando se abre una ventana (o se ha limpiado tras crear una categoría) o se borra el campo de texto del nombre, el título de la ventana es "Nuevo Tipo Intersección", mientras que al escribir en el campo de texto del nombre, el título de la ventana cambia a "Tipo Intersección: *nombre formateado*".

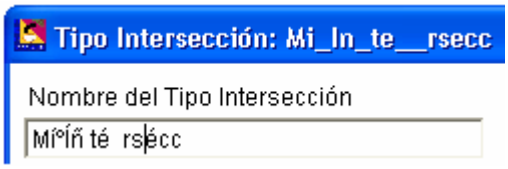


Figura 5.17: Filtrado del nombre del tipo intersección

- "Participación del Tipo Intersección": mediante este grupo de botones de radio se indica la participación del tipo intersección que se quiere crear: "Total" para indicar que cualquier elemento de una superclase debe pertenecer a la subclase, o "Parcial" para indicar que puede haber elementos de una superclase que no pertenezcan a la subclase.

- Lista "Elegir Entidades Superclases": en esta lista se muestran las entidades ya creadas, que tengan llave y que están disponibles (es decir, que no son ya superclases) para ser superclase en el tipo intersección que se está creando.

- Lista "Elegir Entidad Subclase": en esta lista se muestran las entidades ya creadas, sin llave y que están disponibles (es decir, que no son subclase de otra entidad) para ser la subclase del tipo intersección que se está creando.

- Botones "Quitar/Agregar": Se comportan de forma análoga a lo explicado en el apartado 5.6.

- Lista “Superclases del Tipo Intersección”: contiene las entidades elegidas para ser superclases del tipo intersección. El contenido de esta lista se controla mediante los botones “Quitar/Agregar” correspondientes, que mueven las entidades entre esta lista y la lista “Elegir Entidades Superclases”.

- Campo de texto “Subclase del Tipo Intersección (Subclase Compartida)”: contiene la entidad elegida para ser subclase del tipo intersección (subclase compartida). Es un campo protegido, para que su contenido esté gestionado mediante los botones “Quitar/Agregar”.

- Botón “Crear Tipo Intersección”: crea un nuevo tipo intersección con las entidades que pertenecen a ella, de manera que ésta aparece ya en el esquema (Figura 5.18) representada como un círculo con el símbolo de intersección en su interior y con líneas que salen hacia las superclases y hacia la subclase. Una restricción muy importante para crear un tipo intersección es que todas las entidades superclases deben tener la misma llave, en caso contrario se nos informa de este hecho con un mensaje de error (sección 5.10.4). El botón no se habilita mientras que no se haya indicado más de una superclase, la subclase y el nombre del tipo intersección. Si el tipo intersección está mal definido y no puede crearse, Derytas nos informa con un mensaje de error (sección 5.10.4). Cuando se está modificando un tipo intersección, este botón aparece como “Modificar Tipo Intersección”. La posición donde se colocará el tipo intersección, en el momento de su creación, viene indicada por la posición de la última pulsación del botón izquierdo del ratón en la zona de dibujo (si no se ha pulsado ninguna vez, esta posición será la esquina superior izquierda de la zona de dibujo). Si en esa posición hay algún elemento, el tipo intersección se desplazará hacia abajo y hacia la derecha, a partir de esa posición, hasta encontrar una zona sin ningún elemento.

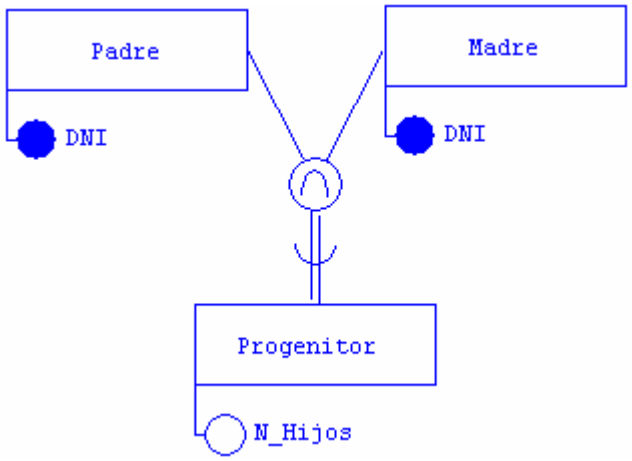


Figura 5.18: Ejemplo de tipo intersección

- Botón "Cancelar": cancela la creación del Tipo Intersección y cierra la ventana.

- *Check* "Cerrar después de Crear": Su comportamiento es análogo al que se ha explicado en el apartado 5.3.

5.8 Creación de Relaciones

Para crear relaciones se debe pulsar el botón **Relación** (Figura 5.1), con lo que se abre la ventana de la Figura 5.19:

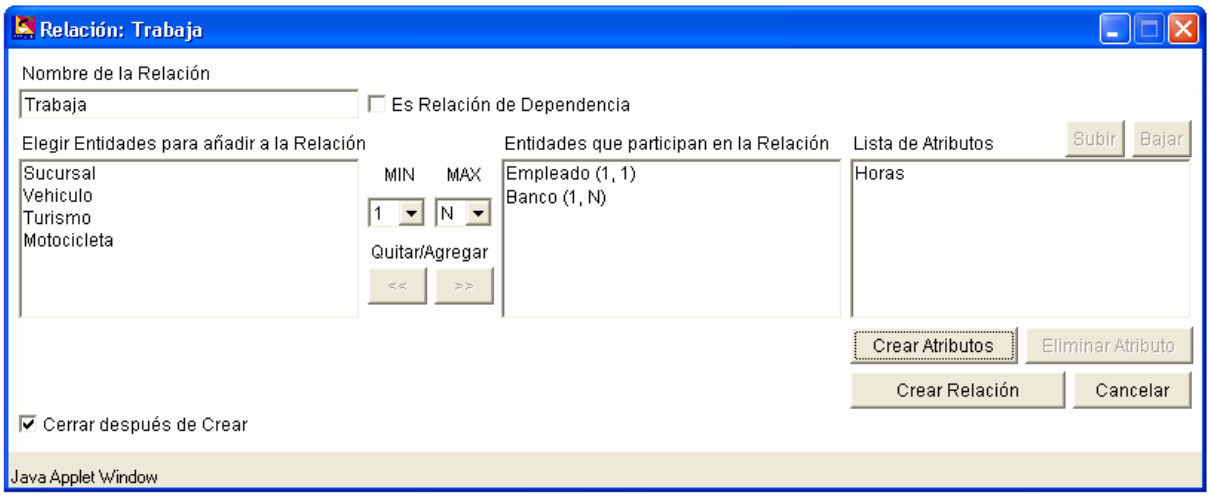


Figura 5.19: Ventana Nueva Relación

- Campo de texto "Nombre de la Relación": este campo se utiliza para asignar un nombre a la relación. Como máximo puede tener 15 caracteres y en el título de la ventana se observa cómo Derytas lo va formateando (Figura 5.19), sustituyendo las vocales acentuadas por las mismas sin acentuar, y los espacios y los caracteres no alfanuméricos por "_", para evitar errores en los nombre de las tablas y de las columnas al ejecutar el *script* de creación de la base de datos. Cuando la se abre una ventana (o se ha limpiado tras crear una relación) o se borra el campo de texto del nombre, el título de la ventana es "Nueva Relación", mientras que al escribir en el campo de texto del nombre, el título de la ventana cambia a "Relación: *nombre formateado*". Este *nombre formateado* es el que Derytas va a dar a la relación en el momento de su creación.

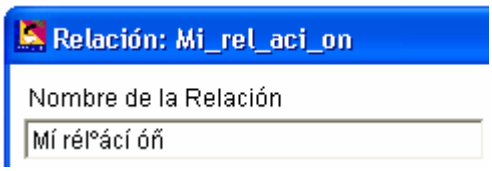


Figura 5.20: Filtrado del nombre de la relación

- *Check* "Es Relación de Dependencia": indica que se trata de una relación para identificar una entidad débil y se tengan en cuenta las restricciones que tienen éstas. Para ello se informa al usuario con los mensajes de error que se comentan en la sección 5.10.5.

- Lista "Elegir entidades para añadir a la Relación": lista de entidades disponibles para participar en la relación. Las entidades que son elegidas para participar en la relación van desapareciendo de esta lista.

- Lista desplegable "MIN": indica la cardinalidad mínima con la que participa en la relación la entidad que se añade a la misma. Los valores que puede tomar son 0, 1, N y M. Estos dos últimos valores representan cardinalidades mayores que 1.

- Lista desplegable "MAX": indica la cardinalidad máxima con la que participa en la relación la entidad que se añade a la misma. Los valores que puede tomar son 1, N y M. Estos dos últimos valores representan cardinalidades mayores que 1.

- Lista “Entidades pertenecientes a la Relación”: muestra las entidades que están participando en la relación, seguidas de la cardinalidad con la que participan (Figura 5.19).

- Botones “Quitar/Agregar”: El botón “>>” se utiliza para agregar a la relación la entidad que se haya seleccionado en la lista “Elegir Entidades para añadir a la Relación” a la lista “Entidades que participan en la Relación”, de manera que dicha entidad desaparece de la primera lista y aparece en la otra. Con el botón “<<” se quita la entidad seleccionada en la lista “Entidades que participan en la Relación” para devolverla a la lista “Elegir Entidades para añadir a la Relación”. El botón “<<” se habilita cuando se selecciona alguna entidad de la lista de entidades participantes en la relación, y el botón “>>” se habilita cuando se selecciona alguna entidad de la lista de entidades disponibles para participar en la relación.

- Lista “Lista de Atributos”: contiene la lista de atributos de la relación (Figura 5.19).

- Botones “Subir” y “Bajar”: funcionan de forma análoga a lo explicado para las entidades en el apartado 5.3.

- Botón “Crear Atributos”: permite definir los atributos para la relación que estamos creando. Para ello, se abre la ventana de la Figura 5.22, cuyo funcionamiento se explica en el apartado 5.9. Los atributos que vayamos definiendo para esta relación van apareciendo en la lista que está sobre este botón, como se muestra en la Figura 5.19. El botón se habilita cuando se ha escrito un nombre para la relación, en otro caso permanece inhabilitado.

- Botón “Eliminar Atributo”: se habilita cuando se selecciona un atributo de la “Lista de Atributos”, de manera que el atributo seleccionado es eliminado de la lista de atributos de la relación.

- Botón “Crear Relación”: crea una nueva relación con todos sus atributos, y las entidades que participan en ella, de manera que ésta aparece ya en el esquema (Figura 5.21), representada como un rombo del que salen líneas hacia las entidades

participantes y sobre las cuáles se encuentran las cardinalidades con las que participan estas entidades en la relación. En el caso de que la participación sea total, estas líneas son dobles, en el caso en el que la participación no sea total, las líneas son simples. El botón no se habilita mientras que no se haya indicado más de una entidad participante y un nombre para la relación. Si la relación está mal definida y no puede crearse, Derytas nos informa con un mensaje de error, indicando la causa del mismo (sección 5.10.5). Cuando se está modificando una relación, este botón aparece como "Modificar Relación". La posición donde se colocará la relación, en el momento de su creación, viene indicada por la posición de la última pulsación del botón izquierdo del ratón en la zona de dibujo (si no se ha pulsado ninguna vez, esta posición será la esquina superior izquierda de la zona de dibujo). Si en esa posición hay algún elemento, la relación se desplazará hacia abajo y hacia la derecha, a partir de esa posición, hasta encontrar una zona sin ningún elemento.

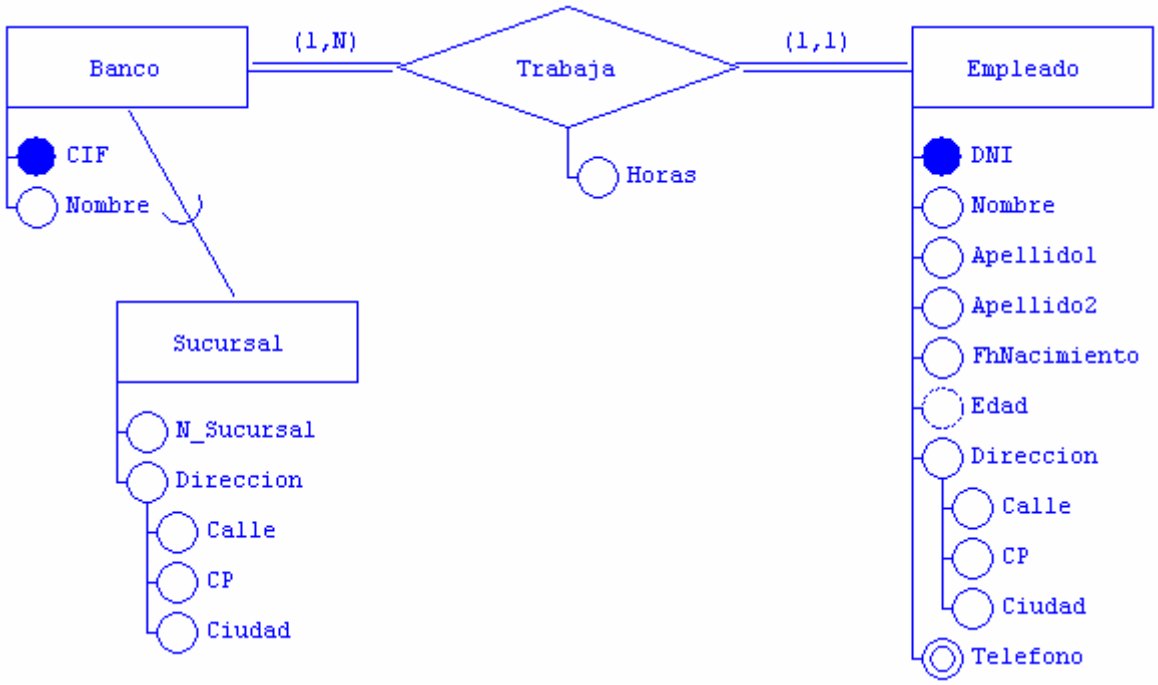


Figura 5.21: Ejemplo de relación

- Botón "Cancelar": cancela la creación de la relación y cierra la ventana.

- *Check* "Cerrar después de Crear": Su comportamiento es análogo al que se ha explicado en el apartado 5.3.

5.9 Creación de Atributos

Sólo se pueden crear atributos para las entidades, las relaciones y otros atributos (caso de atributo compuesto), por tanto, para que estos elementos del esquema tengan atributos hay que pulsar el botón **Crear Atributos** de sus ventanas correspondientes (Figura 5.3, Figura 5.19 y Figura 5.22). Al pulsar el botón en cualquiera de las ventanas citadas anteriormente nos aparece la ventana de la Figura 5.22:

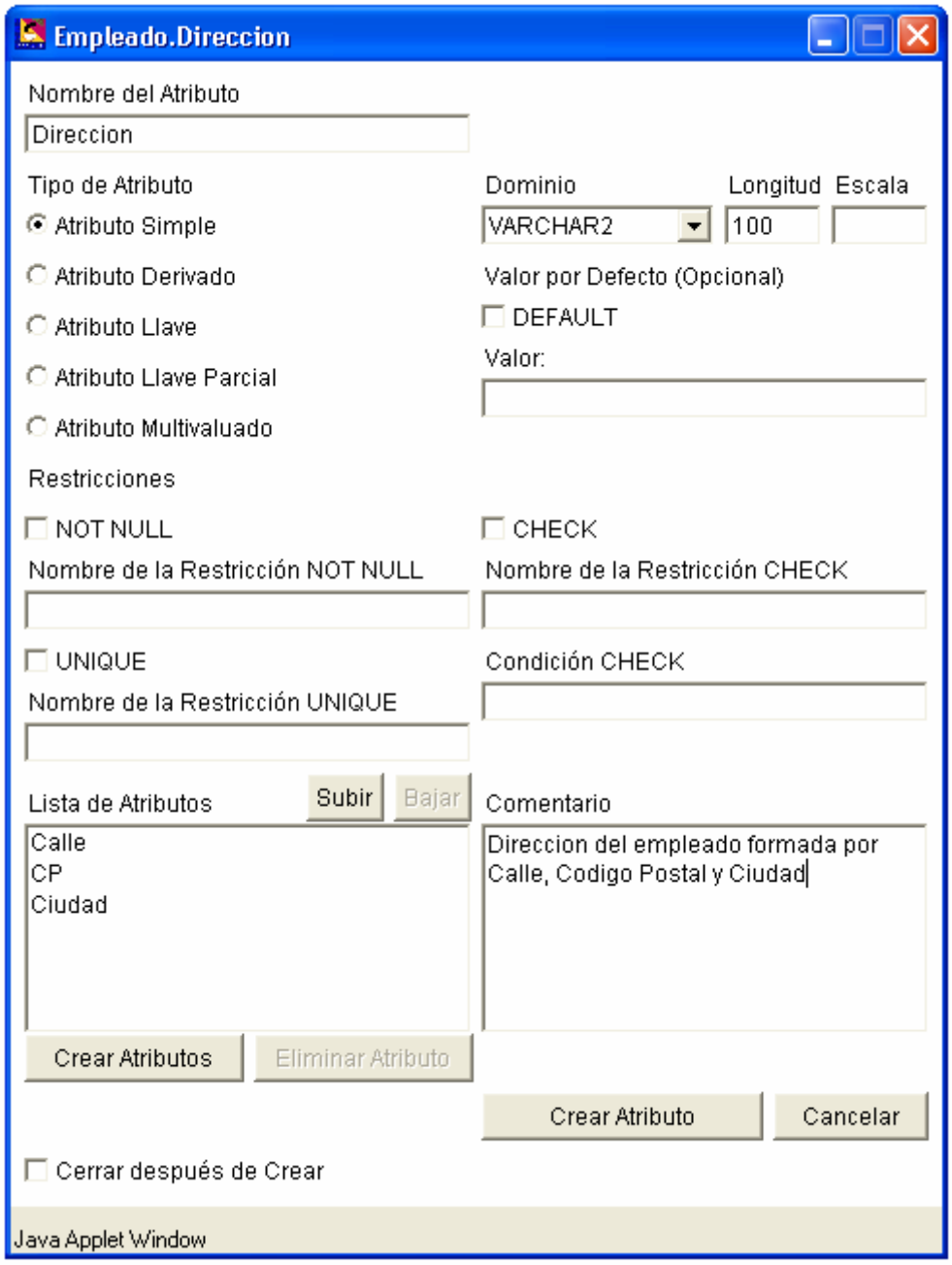


Figura 5.22: Ventana Nuevo Atributo

- Campo de texto “Nombre del Atributo”: este campo se utiliza para asignar el nombre del atributo. Como máximo puede tener 15 caracteres y en el título de la ventana se observa cómo Derytas lo va formateando (Figura 5.23), sustituyendo las vocales acentuadas por las mismas sin acentuar, y los espacios y los caracteres no alfanuméricos por “_”, para evitar errores en los nombre de las tablas y de las columnas al ejecutar el *script* de creación de la base de datos. Cuando se abre una ventana (o se ha limpiado tras crear un atributo) o se borra el campo de texto del nombre, el título de la ventana es “Nuevo Atributo”, teniendo como prefijo las entidades, relaciones o atributos de quien va a ser atributo separados con “.”, como puede observarse en la Figura 5.22, mientras que al escribir en el campo de texto del nombre, el título de la ventana cambia a “*Elemento.nombre formateado*”, como puede verse en la Figura 5.22. Este *Elemento* es el nombre de la entidad, relación o atributo al que pertenece el atributo, y el *nombre formateado* es el nombre que Derytas va a dar al atributo en el momento de su creación.

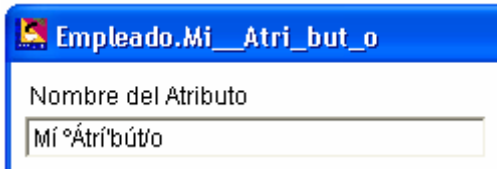


Figura 5.23: Filtrado del nombre del atributo

- Botones de radio de “Tipo de Atributo”: indican el tipo de atributo que se está definiendo, que puede ser “Atributo Simple”, “Atributo Derivado”, “Atributo Llave”, “Atributo Llave Parcial” o “Atributo Multivaluado” (explicado en la sección 3.2.3 del capítulo 3). Según la opción elegida, y si se trata de un atributo de una entidad (o de otro atributo de una entidad) o de un atributo de una relación (o de otro atributo de una relación) se establecerán una serie de restricciones a la hora de su creación (secciones 5.10.1 y 5.10.5).

- Lista desplegable “Dominio”: esta lista permite elegir el dominio al que va a pertenecer el atributo que se está definiendo. Los dominios que se ofrecen son los siguientes [GRO94]: "NUMBER", "LONG", "CHAR", "VARCHAR2", "NCHAR", "NVARCHAR2", "RAW", "LONG RAW", "DATE", "BLOB", "CLOB", "BFILE", "ROWID" y "UROWID".

Asociados a este campo, están los campos “Longitud” y “Escala”. El campo “Longitud” indica el número de caracteres que tendrá el campo, y el campo “Escala” indica, para atributos con dominio numérico, cuántos caracteres de los indicados en “Longitud” son decimales.

- *Check* “Default”: si está seleccionada, indica que el atributo va a tener un valor por defecto, el cuál debe especificarse en la caja de texto “Valor” que, tras marcar esta *check*, pasa a estar habilitada.

- Caja de texto “Valor”: contiene el valor por defecto para el atributo. Sólo se habilita cuando la *check* “Default” está marcada.

- *Check* “NOT NULL”: si está seleccionada, indica que el atributo no va admitir valores nulos.

- Caja de texto “Nombre de la Restricción NOT NULL”: es un campo opcional para escribir el nombre de la restricción NOT NULL cuando se ha marcado la *check* NOT NULL. Si no se indica nombre para la restricción, se le asigna uno con el formato N*_NombreEntidad_NombreAtributo*. Este campo sólo se habilita cuando la *check* NOT NULL está marcada.

- *Check* “UNIQUE”: si está seleccionada, indica que el valor del atributo va a ser único.

- Caja de texto “Nombre de la Restricción UNIQUE”: es un campo opcional para escribir el nombre de la restricción UNIQUE cuando se ha marcado la *check* UNIQUE. Si no se indica nombre para la restricción, ésta se crea a nivel de atributo, y se le asigna uno con el formato U*_NombreEntidad_NombreAtributo*. Si se indica un nombre, el atributo formará parte de la restricción de tabla UNIQUE con ese mismo nombre, de manera que la combinación de los atributos que la forman es única. Este campo sólo se habilita cuando la *check* UNIQUE está marcada.

- *Check* “CHECK”: si está seleccionada, indica que los valores para el atributo deben cumplir la condición especificada en la caja de texto “Condición CHECK”.

- Caja de texto “Condición CHECK”: contiene la condición de la restricción CHECK, que debe ser sintácticamente correcta. Derytas incluye automáticamente el paréntesis inicial y el paréntesis final de las restricciones CHECK cuando genera el *script*.

- Caja de texto “Nombre de la Restricción CHECK”: es un campo opcional para escribir el nombre de la restricción CHECK cuando se ha marcado la *check* CHECK. Si no se indica nombre para la restricción, se le asigna uno con el formato C_*NombreEntidad_NombreAtributo*. Este campo sólo se habilita cuando la *check* CHECK está marcada.

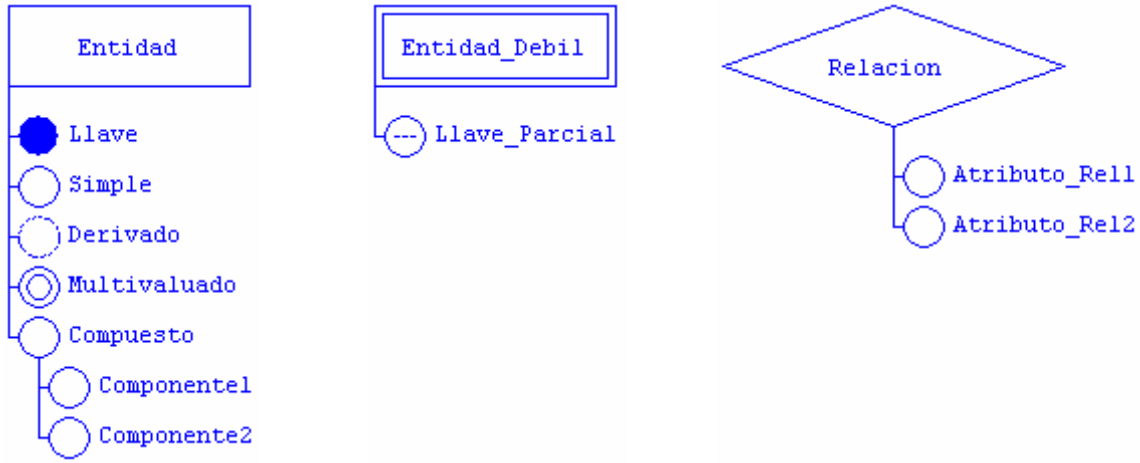
- Botón “Crear Atributos”: permite definir los atributos para el atributo que estamos creando. Para ello, se abre la ventana de la Figura 5.22. Los atributos que vayamos definiendo para este atributo van apareciendo en la lista que está sobre este botón, como se muestra en la Figura 5.22. El botón se habilita cuando se informa el nombre del atributo (Figura 5.22), permaneciendo deshabilitado cuando el nombre no está informado.

- Botón “Eliminar Atributo”: se habilita cuando se selecciona un atributo de la lista de atributos, de manera que el atributo seleccionado es eliminado de la lista de atributos del atributo.

- Lista “Lista de Atributos”: contiene la lista de atributos de la entidad, como se puede ver en la Figura 5.22. Esta lista la podemos ordenar con los botones “Subir” y “Bajar” que se activan o desactivan al seleccionar un elemento de esta lista, y cuyo funcionamiento es análogo al explicado en el apartado 5.3.

- Campo de texto “Comentario”: este espacio está reservado para los comentarios acerca del atributo que se está definiendo. Según la configuración elegida (apartado 5.15), este comentario aparecerá (o no) como un comentario del *script* y/o en una sentencia `COMMENT` (se explica en el apartado 3.5.1 del capítulo 3) para ser introducido en el diccionario de la base de datos.

- Botón “Crear Atributo”: crea una nuevo atributo con todos sus atributos para la entidad, relación o atributo al que pertenezca. Cuando se cree la entidad o la relación, el atributo aparece en el esquema (Figura 5.24) representado como un círculo seguido de su nombre. Según el tipo de atributo, éste se representa de una forma característica El botón no está habilitado mientras que no se escriba un nombre para el atributo que se está definiendo. Si el atributo está mal definido y no puede crearse, Derytas nos informa con un mensaje de error, indicando la causa del mismo (sección 5.10.6).



a) Entidad con atributos **b) Entidad débil con llave parcial** **c) Relación con atributos**

Figura 5.24: Entidades y relación con atributos

- Botón “Cancelar”: cancela la creación del atributo y cierra la pantalla.

- *Check* “Cerrar después de Crear”: Su comportamiento es análogo al que se ha explicado en el apartado 5.3, con la particularidad de que el valor de la *check* no se queda almacenado en la variable de configuración.

5.10 Modificar el Esquema

Para modificar cualquier elemento (entidades, relaciones, especializaciones, tipos unión y tipos intersección), debe hacerse “click” con el botón derecho sobre el elemento que se desea modificar. En ese momento, aparece el menú contextual con todas las opciones inhabilitadas, excepto “Modificar” y “Eliminar”. Eligiendo la opción de “Modificar”, nos aparece la misma ventana que cuando lo creamos con los datos correspondientes y nos permite modificarlos de

igual forma que fueron creados. Al modificar, podemos recibir mensajes de error debido a las validaciones que se realizan para mantener la coherencia del esquema (apartado 5.11).

5.10.1 Eliminar Elementos del Esquema

Cuando se pulsa el botón derecho del ratón sobre una entidad, una relación, una especialización, un tipo unión o un tipo intersección, una de las opciones del menú contextual que aparece habilitada es “Eliminar Elemento”. Eligiendo esta opción el elemento es eliminado del esquema, desapareciendo del mismo. Cuando esta operación se realiza sobre una entidad, la opción “Eliminar Elemento” no estará habilitada si ésta participa en una relación o es superclase o subclase en una especialización, en un tipo unión o en un tipo intersección. Si la entidad seleccionada es superclase o subclase de otra, mediante el botón “Subclase”, aparecerá habilitada la opción del menú contextual “Eliminar como Subclase”. Mediante esta opción eliminamos la relación superclase-subclase entre las dos entidades.

Esta opción de menú contextual no permite eliminar los atributos de la entidades, de las relaciones o de otros atributos. Para ello hay que abrir la ventana de modificación de entidades y relaciones y eliminar el atributo de la lista de atributos de ese elemento. Debido a que los atributos no se pueden modificar, se debe eliminar el atributo entero para eliminar los atributos no deseados de un atributo.

5.11 Mensajes de Error

En el proceso de definición de los elementos de nuestro esquema, podemos recibir mensajes de error por múltiples causas. Cuando se produce un error por el cuál no se puede crear el elemento que acabamos de definir aparece un mensaje de error informándonos de este hecho y, en algunos casos, las causas por las que este error ha tenido lugar. A continuación se exponen los mensajes de error que podemos recibir en la creación de cada tipo de elemento que componen un esquema.

5.11.1 Mensajes de Error acerca de las Entidades

En esta sección se muestran los mensajes de error que podemos obtener al tratar de crear o modificar una entidad:

- “El elemento ‘*nombre entidad’* ya existe”. Este mensaje nos informa de un elemento existente con el mismo nombre de la entidad que estamos intentando crear y que, por tanto, Derytas no puede crearla.
- “Sólo las entidades débiles pueden tener llave parcial”. Este mensaje nos indica que se está intentando crear una entidad no débil en la que se ha definido un atributo como llave parcial.

- “Sólo las entidades no débiles pueden tener llaves primarias”. Este mensaje indica que se intenta crear una entidad débil con una llave que no es parcial.

- “No se puede eliminar la única llave de una superclase”. Este mensaje indica que se ha intentado modificar una entidad que es superclase y se ha intentando eliminar el único atributo llave que tenía.

- “Las subclases no pueden tener llave”. Este mensaje indica que se ha intentado modificar una entidad que es subclase y se ha intentado añadirle un atributo llave.

5.11.2 Mensajes de Error acerca de las Especializaciones

En esta sección se muestran los mensajes de error que podemos obtener al tratar de crear una especialización:

- “El elemento ‘*nombre especialización’* ya existe”. Este mensaje nos informa de un elemento existente con el mismo nombre de la especialización que estamos intentando crear y que, por tanto, Derytas no puede crearla.

5.11.3 Mensajes de Error acerca de los Tipos Unión

En esta sección se muestran los mensajes de error que podemos obtener al tratar de crear un tipo unión:

- “El elemento ‘*nombre tipo unión’* ya existe”. Este mensaje nos informa de un elemento existente con el mismo nombre del tipo unión que estamos intentando crear y que, por tanto, Derytas no puede crearlo.

5.11.4 Mensajes de Error acerca de los Tipos Intersección

En esta sección se muestran los mensajes de error que podemos obtener al tratar de crear un tipo intersección:

- "El elemento '*nombre tipo intersección'* ya existe". Este mensaje nos informa de un elemento existente con el mismo nombre del tipo intesección que estamos intentando crear y que, por tanto, Derytas no puede crearlo.
- "Las superclases del tipo intersección deben tener la misma llave". Este mensaje nos informa de la restricción que tienen los tipos intersección sobre la llave de las superclases, que deben tener todas la misma.

5.11.5 Mensajes de Error acerca de las Relaciones

En esta sección se muestran los mensajes de error que podemos obtener al tratar de crear una relación:

- "El elemento '*nombre relación'* ya existe". Este mensaje nos informa de un elemento existente con el mismo nombre de la relación que estamos intentando crear y que, por tanto, Derytas no puede crearla.

- "Sólo las entidades débiles pueden tener llave parcial". Este mensaje nos indica que se está intentando crear una relación en la que se ha definido un atributo como llave parcial.

- "Sólo las entidades no débiles pueden tener llaves primarias". Este mensaje indica que se intenta crear una relación débil con una llave primaria.

- "Entidad débil '*nombre entidad*' incorrecta, debe tener cardinalidad (1,1)". Este mensaje aparece cuando se está creando una relación de dependencia en la que la entidad débil no está participando como (1,1), que es el valor obligatorio que debe tener.

5.11.6 Mensajes de error acerca de los Atributos

En esta sección se muestran los mensajes de error que podemos obtener al tratar de crear un atributo:

- "El atributo '*nombre atributo'* ya existe". Este mensaje nos informa de un atributo existente en la entidad o relación a la que éste pertenece, con el mismo nombre del que estamos intentando crear y que, por tanto, Derytas no puede crearlo.

- "Las llaves sólo pueden estar compuestas por atributos simples". Con este mensaje se evita que creemos un atributo que es llave con componentes que no sean simples, puesto que Derytas no lo permite.

- "'Longitud' no puede ser vacía". Para los atributos con dominio VARCHAR2, NCHAR, NVARCHAR2, RAW y UROWID, el campo "Longitud" es obligatorio, de lo que se nos informa con este mensaje.

- "Debe introducir un valor numérico para 'Longitud'". Este mensaje nos indica que el valor introducido para "Longitud" no es correcto, ya que debe ser un número entero positivo.

- "Debe introducir un valor numérico para 'Escala'". Este mensaje nos indica que el valor introducido para "Escala" no es correcto, ya que debe ser un número entero positivo.

- "Debe introducir un valor numérico para Longitud si indica Escala". Este error se produce cuando se indica la "Escala" sin haber informado la "Longitud".

- "Debe introducir el valor por defecto". Este mensaje evita que no se indique un valor por defecto para el atributo cuando se haya indicado que debe tenerlo.

- "Debe introducir la condición de la restricción CHECK". Este mensaje indica que se ha indicado que el atributo tiene una condición que cumplir pero que ésta no ha sido informada.

5.11.7 Mensajes de Error Abriendo Esquemas

En esta sección se muestran los mensajes de error que podemos obtener al tratar de abrir un esquema:

- "El fichero con el esquema está dañado o tiene formato incorrecto". Este mensaje nos informa que el fichero que contiene el esquema que se está intentando leer está corrupto o no se trata de un fichero generado con el formato de Derytas.

5.12 Guardar Esquemas

Para guardar un esquema hay que pulsar el botón **Guardar**. En ese momento, se abre una ventana de diálogo (Figura 5.25) para elegir la localización y el nombre del archivo donde se va a guardar el esquema que actualmente se está visualizando. Los archivos que contienen los esquemas deben tener extensión ".eer" y, si ésta no se indica, Derytas la pone automáticamente.

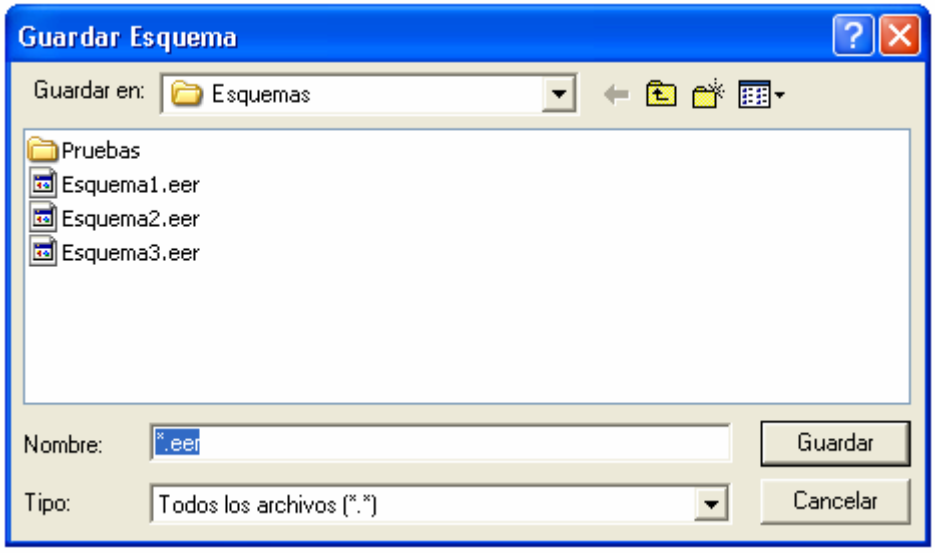


Figura 5.25: Diálogo para guardar un esquema

Si se elige un archivo existente su contenido es reemplazado con el contenido del nuevo esquema, por lo que un mensaje nos pregunta si deseamos reemplazar el archivo antiguo (Figura 5.26).

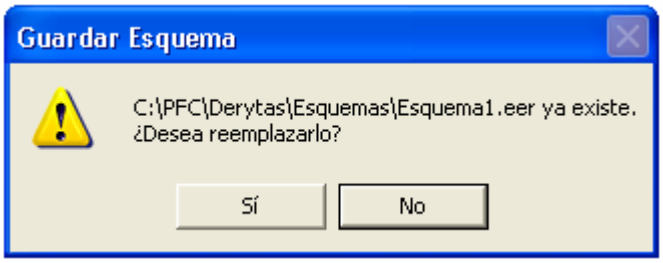


Figura 5.26: Advertencia de reemplazo de archivo

Debido a las restricciones de seguridad de JavaScript, se nos advierte que vamos a crear un objeto ActiveX (Figura 5.27), damos a "Sí" para que aparezca el diálogo de la Figura 5.25.

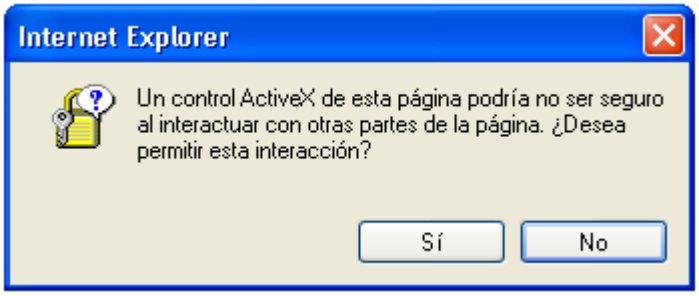


Figura 5.27: Advertencia de seguridad de JavaScript

Si se elige "No", se impide que Derytas pueda leer o guardar ficheros y aparece el mensaje de la Figura 5.28, siendo necesario cargar la página de nuevo para poder utilizar toda su funcionalidad.

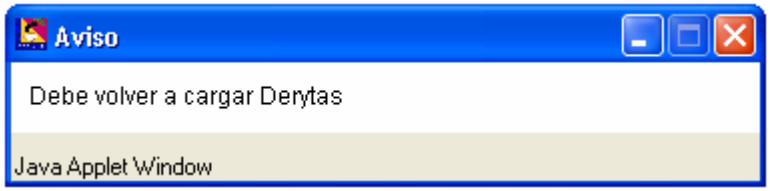


Figura 5.28: Aviso de recarga de Derytas

Algunos antivirus pueden alertar al usuario que se está ejecutando un *script* para decidir si se permite o no su ejecución (Figura 5.29), en este caso se da permiso para que se ejecute y se graba el esquema en la ubicación indicada.

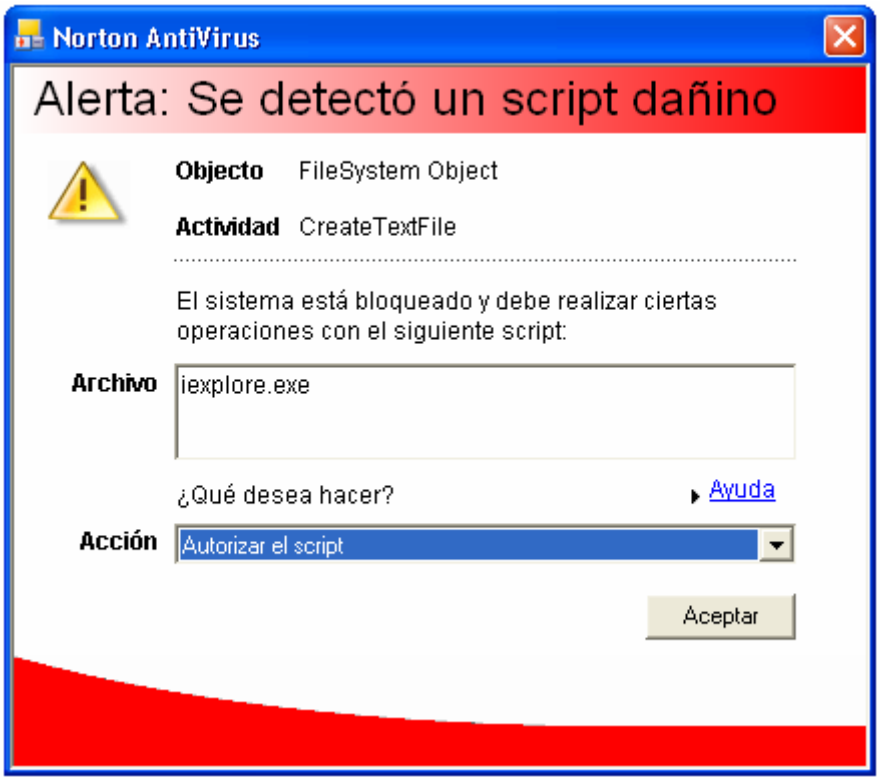


Figura 5.29: Advertencia de seguridad de Norton Antivirus al guardar el archivo

5.13 Abrir Esquemas

Para abrir un esquema hay que pulsar el botón "Abrir Esquema". En ese momento, se abre una ventana de diálogo (Figura 5.30) para elegir la localización y el nombre del archivo de donde se quiere leer el esquema.

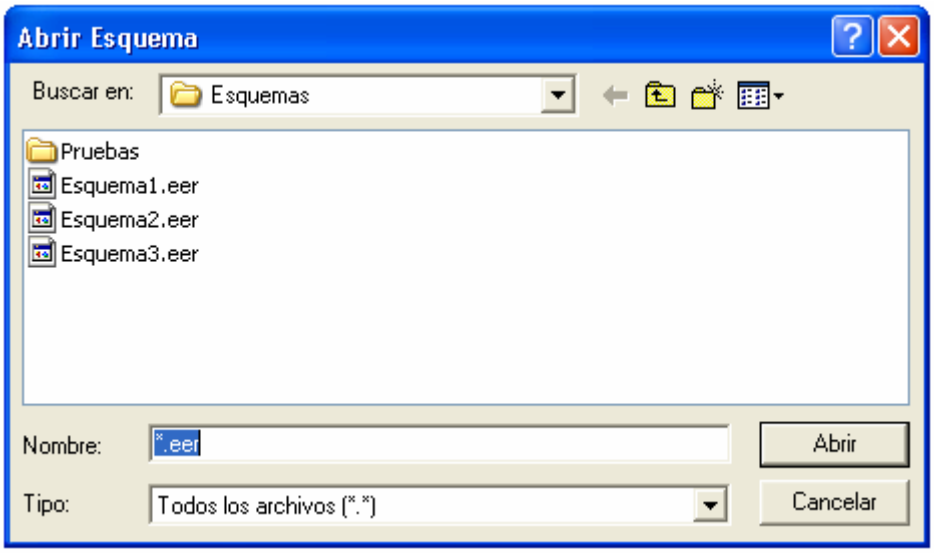


Figura 5.30: Diálogo para abrir un esquema

Debido a las restricciones de seguridad de JavaScript, se nos advierte que vamos a crear un objeto ActiveX (Figura 5.27), damos a "Sí" para que aparezca el diálogo de la Figura 5.30. Si se elige "No", se impide que Derytas pueda leer o guardar ficheros y aparece el mensaje de la Figura 5.28, siendo necesario cargar la página de nuevo para poder utilizar toda su funcionalidad.

Algunos antivirus pueden alertar al usuario que se está ejecutando un *script* para decidir si se permite o no su ejecución (Figura 5.29), en este caso se da permiso para que se ejecute y se abra el esquema.

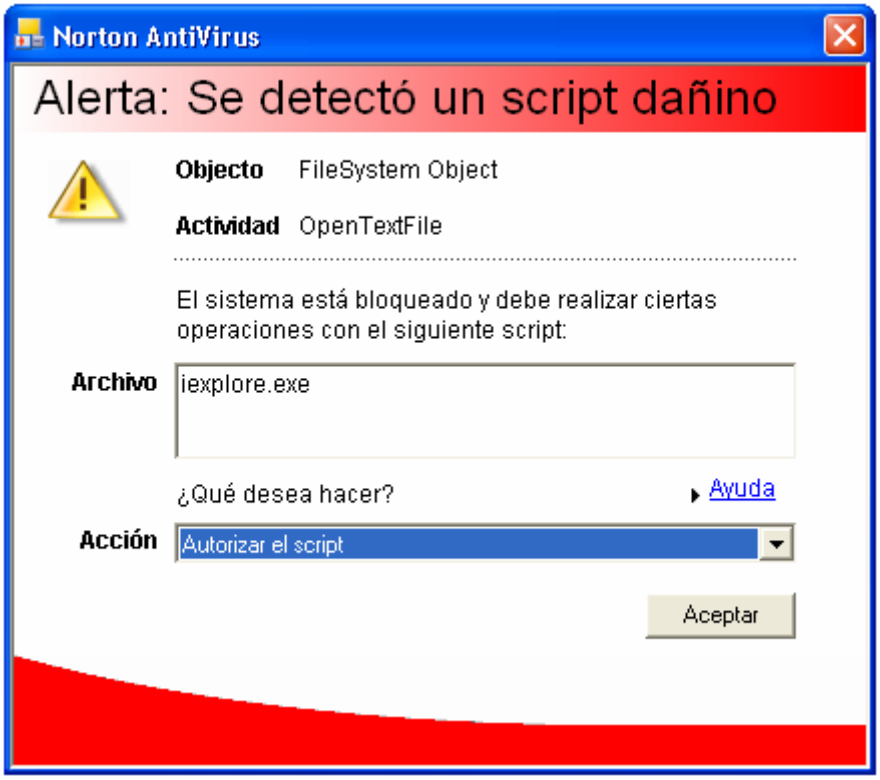


Figura 5.31: Advertencia de seguridad de Norton Antivirus al abrir el archivo

5.14 Generación de Código

Una vez que se ha finalizado el esquema, el siguiente paso es la generación del *script* en SQL para crear la base de datos. El código resultante puede visualizarse mediante el botón **Visualizar Script** o a través de la opción del menú contextual del mismo nombre. Al elegir cualquiera de las dos opciones anteriores se muestra el código en una ventana como la de la Figura 5.32:

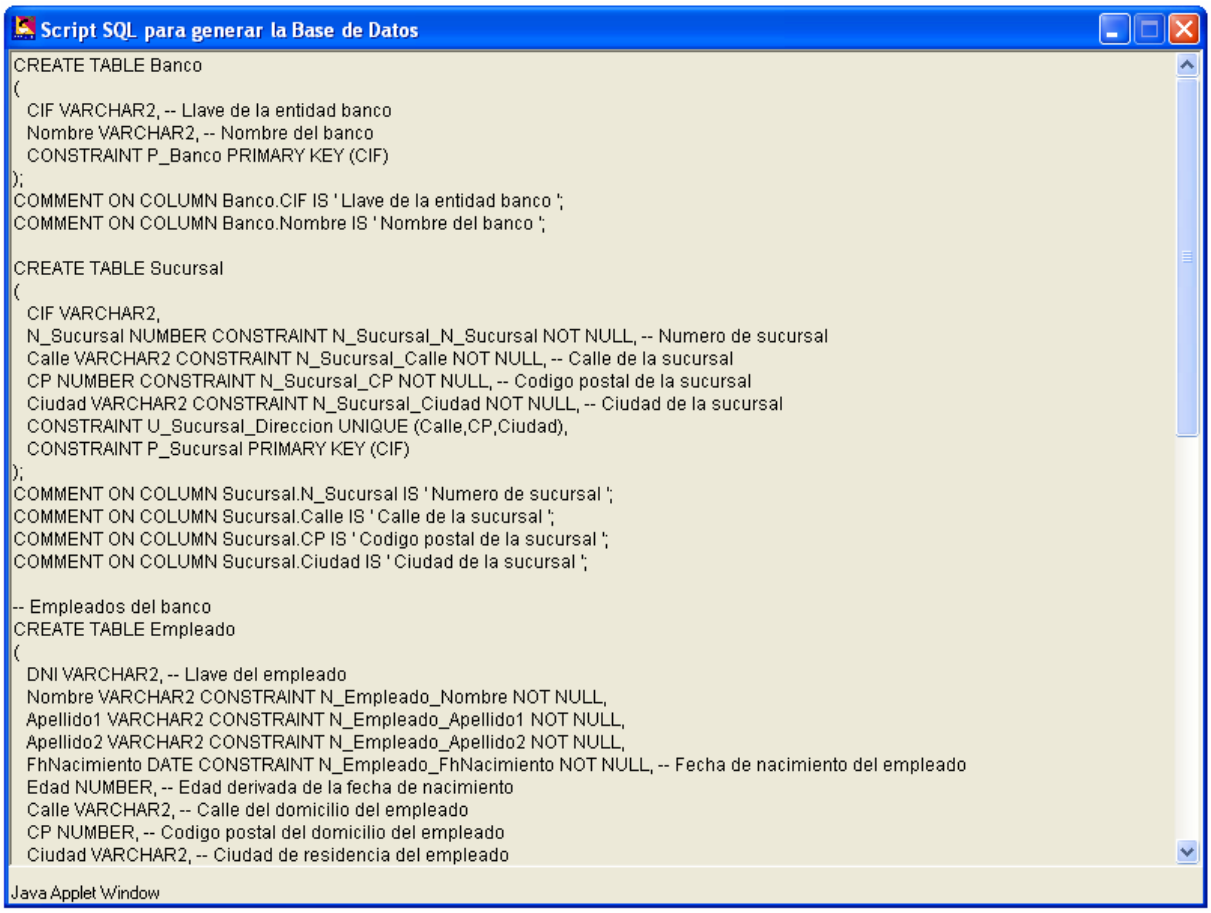


Figura 5.32: Ventana para visualizar el *script*

Si se desea generar un fichero con el *script*, se debe pulsar el botón **Generar Script**. En ese momento, se abre una ventana de diálogo (Figura 5.33) para elegir la localización y el nombre del archivo donde se va a guardar el *script* de generación de la base de datos. Los archivos que contienen los *scripts* deben tener extensión ".sql" y, si ésta no se indica, Derytas la pone automáticamente. Si se elige un archivo existente su contenido es reemplazado con el contenido del nuevo *script*.

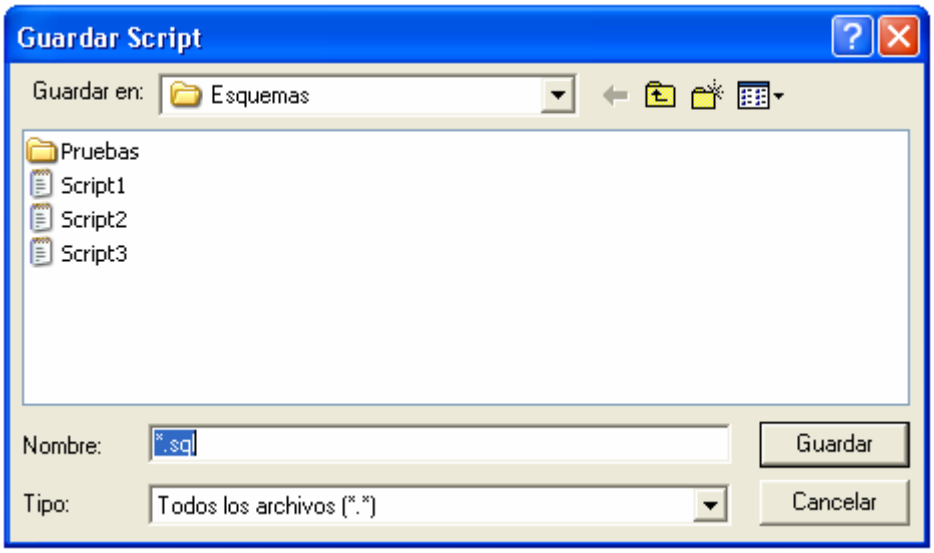


Figura 5.33: Diálogo para guardar un *script*

Debido a las restricciones de seguridad de JavaScript, se nos advierte que vamos a crear un objeto ActiveX (Figura 5.28), damos a "Sí" para que aparezca el diálogo de la Figura 5.33. Si se elige "No", se impide que Derytas pueda leer o guardar ficheros y aparece el mensaje de la Figura 5.28, siendo necesario cargar la página de nuevo para poder utilizar toda su funcionalidad.

Algunos antivirus pueden alertar al usuario que se está ejecutando un *script* para decidir si se permite o no su ejecución (Figura 5.29), en este caso se da permiso para que se ejecute y se guarde el *script*.

5.15 Opciones de Configuración

En este apartado se exponen las opciones de configuración de la aplicación relativas a funcionalidad, opciones gráficas y de generación de código.

Al seleccionar el botón **Opciones** se abre la ventana de la Figura 5.34, que se comenta a continuación:

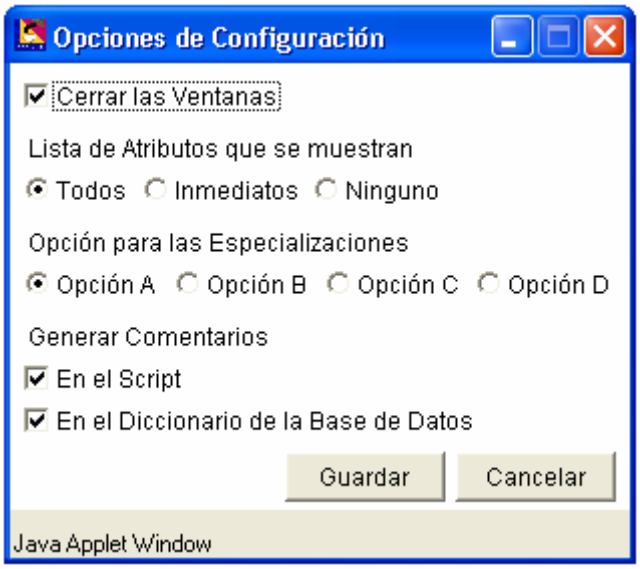


Figura 5.34: Ventana de Opciones de Configuración

- Con la *check* "Cerrar las Ventanas" se controla el comportamiento de las ventanas después de modificar o crear un elemento del esquema, de manera que si está marcada, la ventana se cerrará, y si está sin marcar la ventana se limpiará y permanecerá abierta. Cuando una ventana se abre para modificar un elemento, esta *check* aparece marcada y deshabilitada para que la ventana se cierre tras la modificación. Cuando se marca o desmarca en cualquier ventana, el cambio se extiende al resto de las ventanas de Derytas.

- Los botones de radio "Lista de Atributos que se muestran" indican el nivel de atributos que va a mostrarse:

    - Todos: muestra todos los atributos que se hayan definido (Figura 5.35 a)).

    - Inmediatos: muestra el primer nivel de la jerarquía de atributos (Figura 5.35 b)).

    - Ninguno: no muestra ninguno de los atributos (Figura 5.35 c)).

a) Todos **b) Inmediatos** **c) Ninguno**

Figura 5.35: Opciones para mostrar los atributos

- Los botones de radio "Opción para las Especializaciones" indican la opción para generar el código de las especializaciones que va a utilizarse (apartado 3.4 del Capítulo 3):

    - Opción A: Se crea una tabla para la superclase con sus atributos y una tabla para cada subclase con sus atributos y con la misma la llave que tenga la superclase.

    - Opción B (sólo para especializaciones disjuntas con participación total): Se crea una tabla para cada subclase con sus atributos y con la misma llave que tenga la superclase.

    - Opción C (sólo para especializaciones disjuntas definidas por atributo): Se crea una única tabla con los atributos de la superclase, los atributos de todas las subclases y el atributo de la especialización. Si la especialización no es definida por atributo, éste se crea. La tabla tendrá como llave los mismos atributos llave de la superclase.

- Opción D: Se crea una única tabla con los atributos de la superclase, los atributos de las subclases y un atributo lógico por cada subclase de la especialización. La tabla tendrá como llave los mismos atributos llave de la superclase.

El código para las especializaciones se genera teniendo en cuenta lo siguiente:

1. Cuando se indique Opción B, las especializaciones solapadas y las disjuntas parciales generarán su código según la Opción A. Esto es debido a que la opción B sólo está permitido para especializaciones disjuntas con participación total.

2. Cuando se indique Opción B, si la superclase de la especialización participa en una relación, el código de la especialización se generará según la Opción A, aunque se trate de una especialización disjunta. Esto es debido a la necesidad de que exista una tabla con la llave de la superclase para poder generar el código de la relación.

3. Cuando se indique Opción C, las especializaciones solapadas generarán su código según la Opción D. Esto es debido a que la opción C sólo está permitida para especializaciones disjuntas.

- *Checks* para "Generar Comentarios": marcando y desmarcando estas *checks* se indica si el *script* va a incorporar o no comentarios y qué clase de comentarios:

  - *Check* "En el Script": indica si se van a generar comentarios que aclaren el contenido del script.

  - *Check* "En el Diccionario de la Base de Datos": indica si se van a generar comentarios para el diccionario de la base de datos con la sentencia `COMMENT`.

  La fuente de estos comentarios son los introducidos al generar las entidades y los atributos.

- Botón “Guardar”: permite guardar la configuración elegida para que, desde ese mismo instante, Derytas se comporte como deseamos, y se cierra la ventana.

- Botón “Cancelar”: descarta los cambios de configuración que se hayan realizado, de manera que se mantiene la última configuración y se cierra la ventana.

5.16 Acerca de…

Seleccionando el botón **Acerca de…** nos aparece la ventana de la figura 5.36, que muestra información relacionada con el proyecto, como el autor, los propietarios del proyecto y el año.

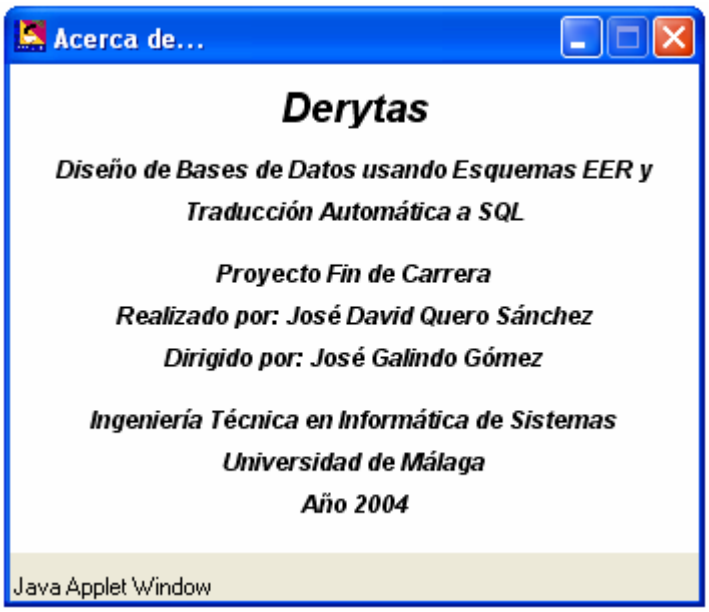


Figura 5.36: Ventana Acerca de…

5.17 Ayuda

Cuando se elige el botón **Ayuda** se abre una nueva ventana del explorador, en la que se carga la ayuda de Derytas (Figura 5.37). La página de ayuda se divide en dos zonas, una zona que contiene las referencias a los temas en los que ésta se encuentra dividida (zona de la izquierda), y otra zona en la que se muestra el contenido de los temas (zona de la derecha). Según se van seleccionando temas, el contenido de éstos se muestra en la zona de la derecha.

Figura5.37: Página de Ayuda

Conclusiones y Líneas Futuras

A continuación se exponen las conclusiones obtenidas tras la realización del proyecto resumiendo muy brevemente el software obtenido. También se compara este software con otros programas similares y se indican posibles ampliaciones o mejoras que pueden hacerse en futuras versiones.

Conclusiones

El objetivo principal de este proyecto ha sido poder dibujar esquemas EER de manera rápida e intuitiva para, una vez terminados, generar el conjunto de sentencias SQL que generan la base de datos que se ha modelado con ese esquema. También se ha perseguido poder utilizar la aplicación en cualquier máquina, independientemente del hardware utilizado o del sistema operativo instalado. Otro de los objetivos conseguidos ha sido poder guardar el esquema en un fichero, lo que permite desarrollar el esquema en períodos de tiempo y en lugares geográficos distintos.

En primer lugar, hemos de destacar que se han cumplido los objetivos planteados para este proyecto, incluso se han visto ampliados.

La aplicación permite al usuario dibujar esquemas EER de manera sencilla mediante un interfaz formado por un conjunto de ventanas que solicitan al usuario los datos que éste desee reflejar en el esquema, incluyendo todos los objetos del modelo EER propuesto por Elmasri y Navathe (2004) [ELM04]: Entidades, Relaciones, Atributos (simples, compuestos, derivados, multivaluados...), especializaciones, tipos unión, subclases compartidas...

La experiencia en el diseño de esquemas EER nos muestra cómo éstos pueden llegar a tener un tamaño considerable, siendo imposible su visualización en una única pantalla. Este hecho obliga a que la zona donde se visualiza el esquema se salga de los límites de la pantalla, lo que supone un problema a la hora de crear los elementos del mismo. Este problema se ha solucionado situando los botones de creación de forma que permanecen visibles durante todo el tiempo, y controlando la posición del ratón para saber dónde situar los elementos en el esquema.

Derytas no sólo permite dibujar el esquema EER, sino que, a medida que se van creando los elementos que forman parte del esquema, se valida que cumplen las restricciones

que les son impuestas por definición o que les son impuestas por el esquema al que pertenecen. Además, a cada elemento del modelo EER puede añadírsele un comentario explicativo que indique su utilidad o su contenido, el cual luego podrá incorporarse automáticamente al diccionario de datos de una base de datos (usando el comando `COMMENT`).

La manera de conseguir que la aplicación pueda utilizarse en cualquier máquina, independientemente del hardware o del software, ha sido desarrollar una aplicación *Web*, de forma que los únicos requisitos son disponer de un ordenador conectado a Internet y, dado que la aplicación es un *applet*, es necesario además tener instalada la máquina virtual de java en nuestro ordenador. En la página de Sun Microsystem (http://www.sun.com), esta máquina virtual se encuentra disponible para cualquier plataforma y de forma totalmente gratuita. Una ventaja adquirida de haber desarrollado una aplicación *Web*, es que las actualizaciones se realizan en el servidor, sin tener que realizar ningún tipo de instalación en los clientes cada vez que se modifique el servidor.

Para poder guardar los elementos del esquema en fichero, se ha utilizado JavaScript (para obtener los datos del *applet* y guardarlos). Esta es la solución que se ha adoptado debido a las restricciones impuestas a los *applets*, que dificultan el acceso de los mismos a ficheros locales por motivos de seguridad.

Líneas Futuras

Toda aplicación es susceptible de ser mejorada, por lo que, a pesar de lo práctico y fácil de utilizar que es Derytas, se han pensado algunas posibles ampliaciones futuras, entre las que podemos destacar las siguientes:

1. Permitir la modificación de atributos: La aplicación permite la modificación de todos los elementos del esquema menos de los atributos. Las entidades, relaciones y atributos (para el caso de atributos compuestos) permiten crear los atributos que se consideren necesarios y también permiten eliminarlos. Los atributos compuestos sólo pueden eliminar sus atributos mientras se está definiendo el atributo compuesto, sin embargo, las entidades y relaciones también pueden eliminar atributos cuando se están modificando. Por lo tanto, una posible mejora de la aplicación consiste en poder modificar los atributos cuando se están definiendo o modificando las entidades, las

relaciones y los atributos compuestos, teniendo en cuenta las restricciones que pueda haber sobre los mismos.

2. Opción Deshacer: Además de poder modificar elementos del esquema, Derytas permite su eliminación del mismo. Una vez que un elemento se ha eliminado no se puede recuperar, por lo que la aplicación se podría ampliar para deshacer los cambios, tanto de los elementos eliminados como de los modificados.

3. Permitir leer/salvar ficheros en el servidor: La aplicación se ejecuta en su totalidad en el cliente, incluso genera los ficheros para guardar el esquema actual y el *script* de creación de la base de datos en el cliente. Aunque estos ficheros no suelen ser de gran tamaño ya que sólo contienen texto plano, requieren ser grabados en algún dispositivo externo cuando queremos utilizarlo en otro ordenador. Una forma de solucionar esto sería permitir que estos ficheros se guardaran (y se recuperaran) del servidor. Esta facilidad también debería tenerse en cuenta en futuras ampliaciones de la aplicación.

4. Ayuda contextual: Mediante el botón de ayuda se accede al índice de la ayuda de Derytas. En futuras ampliaciones, sería interesante considerar la posibilidad de acceder directamente al apartado de la ayuda que está relacionado con el elemento que se está definiendo actualmente. De esta manera, por ejemplo, se podría acceder a la ayuda para entidades desde la ventana para la creación y modificación de entidades.

5. Código de especializaciones a nivel individual: La opción para generar el código para las especializaciones se indica a nivel general de la aplicación en la ventana de opciones. Se podría ampliar la aplicación de manera que esta opción se pueda especificar a nivel de cada especialización de manera individual.

6. Permitir superclases múltiples: La aplicación no permite que una entidad que sea superclase de otra entidad, de una especialización, de un tipo unión o de un tipo intersección pueda serlo también de otra entidad, de otra especialización, de otro tipo unión o de otro tipo intersección. Una futura ampliación de Derytas podría considerar permitirlo, y, además, que las subclases de otras entidades, especializaciones, tipos unión y tipos intersección puedan convertirse en superclases de otras entidades, especializaciones, tipos unión y tipos intersección.

7. Extensión a UML: Dado el auge y el uso extensivo de UML, sería interesante que, en futuras ampliaciones, se pudiera elegir entre EER o UML para dibujar el esquema de la base de datos.

8. Generación automática de disparadores para control de restricciones: Se puede mejorar el código generado incluyendo *triggers* que controlen ciertas restricciones de los datos, por ejemplo, los valores de algunos atributos derivados.

9. Botones móviles: Otra de las mejoras que podría hacerse, sería permitir que la zona de botones fuera móvil, permitiendo que el usuario elija qué zona de la pantalla va a constituir el área de dibujo del esquema.

10. Guardar gráfico e Imprimir: Una vez que se ha conseguido cumplir el objetivo de guardar el contenido del esquema en un fichero, sería muy útil que en futuras ampliaciones se permitiera guardar el esquema en algún formato gráfico que permitiera su impresión.

11. Extensión a SQL:2003: Una vez publicado el estándar SQL:2003, cualquier mejora de la aplicación debería considerar los nuevos tipos de datos introducidos en el mismo.

12. Extensión al modelo FuzzyEER: El modelo EER tiene una extensión llamada FuzzyEER que permite utilizar información imprecisa o difusa. El modelo FuzzyEER [URR03, GAL05], define distintos tipos de atributos difusos, entidades difusas, relaciones difusas, especializaciones difusas, restricciones difusas, etc. Esta extensión utiliza la lógica difusa como base matemática en sus definiciones.

Referencias

A continuación se indican los libros consultados y las direcciones *Web* visitadas para la elaboración de este proyecto.

Referencias Bibliográficas

[BDGA02]	Apuntes de la asignatura "Bases de Datos", Ingeniería Técnica en Informática de Sistemas, Universidad de Málaga. Impartida por J. Galindo Gómez.
[CHA74]	Chamberlin, D. y Boyce, R.: "SEQUEL: A Structured English Query Language". En SIGMOD, 1974
[CHA76]	Chamberlin, D. et al.: "SEQUEL2: A Unified Approach to Data Definition, Manipulation and Control". IBM Journal of Research and Development, 20:6, noviembre de 1976
[CHE76]	Chen, Peter: "The Entity-Relationship Model – Toward a Unified View of Data, TODS, 1:1, marzo de 1976
[CUE00]	Pedro Manuel Cuenca Jiménez: "Programación en Java". Anaya Multimedia.
[ELM04]	Ramez Elmasri, Shamkant B. Navathe: "Fundamentals of Database Systems". Addison-Wesley, 2004.
[FRO00]	Agustín Froufe: "JAVA 2. Manual de usuario y tutorial". Ra-Ma.
[GAL05]	Galindo J., Urrutia A., Piattini M., "Fuzzy Databases: Modeling, Design and Implementation". To publish by Idea Group Publishing Hershey, USA, 2005.
[GRO94]	James R. Groff, Paul N. Weinberg: "Aplique SQL". McGraw-Hill, 1994
[ORA02]	Scott Urman, "Oracle 9i. Programación PL/SQL". McGraw-Hill Interamericana, 2002
[URR03]	Urrutia A. (2003): "Definición de un Modelo Conceptual para Bases de Datos Difusas". Ph. Doctoral Thesis, University of Castilla-La Mancha (Spain).

Recursos de Internet

Internet es una de las mayores fuentes de información actualmente. Todo lo que no se consigue en los libros, generalmente se consigue en la red. Aquí se incluyen algunos enlaces que se han consultado.

[W3ACM02]	Andrew Eisenberg, Jim Melton: "SQL/XML is Making Good Progress". ACM SIGMOD Record, Vol. 31, No. 2, Junio 2002. http://www.acm.org/sigmod/record/issues/0206/standard.pdf
[W3ACM04]	Andrew Eisenberg, Jim Melton: "SQL:2003 Has Been Published". ACM SIGMOD Record, Vol. 33, No. 1, March 2004. http://www.acm.org/sigmod/record/issues/0403/E.JimAndrew-standard.pdf
[W3ALC]	Miguel Ángel Álvarez: "Qué es CGI". http://www.desarrolloweb.com/articulos/758.php?manual=27
[W3ALJ]	Miguel Ángel Álvarez: "Qué es JavaScript". http://www.desarrolloweb.com/articulos/25.php?manual=27
[W3AME]	Ricardo Amenzua: "Curso de JavaScript". http://www.lawebdelprogramador.com/cursos/enlace.php?idp=699&id=45&texto=JavaScript
[W3CAR]	Lola Cárdenas Luque: "Curso de CGI". http://rinconprog.metropoliglobal.com/CursosProg/
[W3JAL]	Javier García de Jalón: "Applets en Java 1.1". Universidad de San Sebastián. http://www1.ceit.es/Asignaturas/Informat2/Clases/Clases9899/Clase14/JavaApplets/
[W3MAR]	Pedro Rufo Martín: "Manual de HTML". http://www.asptutor.com
[W3ROM]	Luis F. Romero: "El Lenguaje HTML". Universidad de Cantabria. http://cdec.unican.es/libro/lenguaje_HTML.htm
[W3SUN]	Tutorial de Java de Sun Microsystems: http://java.sun.com/docs/books/tutorial/index.html
[W3UNAV]	"Tutorial de JavaScript". Universidad de Navarra. http://www.unav.es/cti/manuales/TutorialJavaScript/indices/index.html

[W3UPM]	"Aprenda Java como si estuviera en primero". Escuela Superior de Ingenieros Industriales de San Sebastián. Universidad de Navarra. http://mec21.etsii.upm.es/ayudainf/aprendainf/Java/Java2.pdf