UNIVERSIDAD POLITÉCNICA DE MADRID

FACULTAD DE INFORMÁTICA

TRABAJO FIN DE CARRERA

Sistema de Visualización para un Entorno Avanzado de Desarrollo de Software

Autor: Juan José Amor Iglesias

Tutor: Manuel Collado Machuca

24 de septiembre de 1997

A mi familia

Agradecimientos

Sirvan estas líneas para expresar mi agradecimiento a todos aquellos que, de una u otra forma, me han apoyado en la realización, no ya del presente trabajo, sino de toda la carrera.

En primer lugar me gustaría agradecer la colaboración prestada por la Unidad de Programación del Departamento de Lenguajes y Sistemas, de la Facultad de Informática de Madrid, al proporcionarme todo el material necesario para la realización de este trabajo, desde la información bibliográfica hasta las propias máquinas. En particular, me gustaría agradecer a Manuel la oportunidad que me ha dado de colaborar en el proyecto LOPE y finalmente realizar este trabajo.

En segundo lugar, tengo que agradecer la paciencia de la empresa que me dio mi primer trabajo, *InforEspaña*, que me guardó el puesto cuando decidí concentrarme en la realización de este trabajo, a pesar de los meses que esta situación ha durado.

En tercer lugar, tengo que agradecer el apoyo y confianza recibidos desde mi familia, y especialmente el de mis padres. A ellos también les tengo que agradecer el esfuerzo económico que nos ha permitido costearme la carrera.

Índice General

Resumen **xv**

1 Entornos de desarrollo de software **1**

1.1 Conceptos básicos 1

1.2 Clasificación de los entornos de desarrollo 3

1.2.1 Cobertura del ciclo de vida 3

1.2.2 Tipo de software desarrollado 4

1.2.3 Clasificación de Dart [6] 5

1.3 Integración de herramientas 10

1.3.1 Criterios de integración 11

1.3.2 Niveles de integración 13

2 El Entorno Lope **15**

2.1 Introducción 15

2.2 El subsistema de base de datos 18

2.2.1 Objetos 18

2.2.2 Esquemas de definición de datos 19
2.2.3 Estructura de la base de datos 21
2.2.4 Representación de los objetos 21
2.2.5 Codificación de esquemas 22
2.3 El Subsistema de ejecución de servicios 24
2.3.1 Instrucciones elementales 24
2.3.2 Codificación de las expresiones 27
2.3.3 Ejecución de las expresiones 27
2.3.4 Esquemas de definición de servicios 27
2.4 El subsistema de visualización 28
2.5 El subsistema de diálogo 28
2.6 El terminal virtual 29
2.6.1 Pantalla 29
2.6.2 Ventana 29
2.6.3 Panel de texto 30
2.6.4 Panel gráfico 30
2.6.5 Botón 30
2.6.6 Campo de texto 31
2.6.7 Teclado y ratón 31
2.6.8 Procesos 31

3 Ideas generales sobre el subsistema de visualización 33

3.1 Introducción 33
3.2 Especificación del subsistema 34
3.2.1 Formatos de visualización 34
3.2.2 Vistas 41
3.2.3 Cajas 43

3.3 Líneas de ampliación 44

3.3.1 Alineaciones condicionales 44

3.3.2 Formatos gráficos 46

3.4 Uso del subsistema desde una herramienta LOPE 47

4 Ejemplos de Esquemas de Formatos 51

4.1 Bloques de código 51

4.2 El formato Fecha 57

4.3 El formato Universal 62

4.4 El formato de Documentos 74

5 Características de la implementación 97

5.1 Visión general 97

5.2 Almacenamiento de objetos 99

5.2.1 Creación y destrucción de objetos 101

5.2.2 Lectura y alteración de atributos 102

5.2.3 Atributos de los objetos 102

5.3 Estudio de `VisBox` 104

5.3.1 Descripción de la interfaz 104

5.3.2 Generación de jerarquías de cajas 106

5.3.3 Alineación de las jerarquías de cajas 115

5.3.4 Otras funciones de `VisBox` 120

5.4 Estudio de `VisView` 124

5.4.1 Descripción de la interfaz 125

5.4.2 Creación y borrado de vistas 126

5.4.3 Refresco de vistas 128

5.4.4 Refresco del panel de texto 133

5.4.5 Otras funciones de `VisView` 135

5.5 Modificaciones a los trabajos de Arana [1] y Lorenzana [10] . . . 137

6 Ejemplo de utilización 143

6.1 Código fuente 143

6.1.1 Código del programa 143

6.1.2 Objeto de prueba 148

6.2 Resultados 151

6.2.1 Comentarios previos 151

6.2.2 Resultados en pantalla 152

7 Código fuente 163

7.1 Módulos de definición 163

7.1.1 Módulo `VisBoxL.def` 163

7.1.2 Módulo `VisViewL.def` 167

7.1.3 Módulo `VisBox.def` 171

7.1.4 Módulo `VisView.def` 179

7.2 Módulos de implementación 185

7.2.1 Módulo `VisBoxL.mod` 185

7.2.2 Módulo `VisViewL.mod` 189

7.2.3 Módulo `VisView.mod` 193

7.2.4 Módulo `VisBox.mod` 214

Bibliografía 257

Índice de Materias 259

Índice de Figuras

2.1 Arquitectura del Sistema Lope 16
2.2 Codificación interna del esquema documento 25
2.3 Codificación interna de un documento 26

3.1 Diferentes tipos de alineación horizontal 39
3.2 Diferentes tipos de alineación vertical 39
3.3 Ejemplo de alineación 40
3.4 Alineaciones de las cajas 40
3.5 Esquemas de Formatos 42
3.6 Ejemplo de alineación condicional Z 45
3.7 Ejemplo de alineación condicional T 45

4.1 Un bloque de lenguaje *Pascal* 52
4.2 Cajas para bloques en lenguaje *Pascal* 52
4.3 Esquema para bloques de código 53
4.4 Formato para representar bloques de código *Pascal* 56

4.5 Esquema para objetos fecha 57
4.6 El formato Fecha 57
4.7 El formato Mes 61
4.8 El Formato Universal 63
4.9 El Formato *utype* 64
4.10 El Formato *uvalue* 64
4.11 Los Formatos *uref* y *utext* 65
4.12 El formato *utuple* 66
4.13 Los Formatos *ulistcomp* y *ulistdircomp* 75
4.14 Esquemas para objetos Documento 77
4.15 Una presentación de documento 78
4.16 El Formato Documento 79
4.17 El Formato Título 79
4.18 El formato Secciones 80
4.19 El formato Sección 81
4.20 El formato para el cuerpo de una sección 81
4.21 El formato línea 82
4.22 El formato para el título de una sección 82
4.23 El formato para numerar secciones 83

5.1 Diagrama modular 98
5.2 Un problema con la alineación 116
5.3 Solución al problema de la alineación 117
5.4 Inserción de componente intermedia 131
5.5 Inserción de la última componente 131
5.6 Tres casos para eliminar *basura* 134
5.7 Listas de cajas asociadas a un objeto 136

6.1 Estado al crear la vista 153
6.2 Inserción de una palabra 154
6.3 Reemplazo de una palabra 155
6.4 Cambio del mes 156
6.5 Reemplazo de todo el objeto fecha 158
6.6 Extraccion de la sección *Ideas Generales* 159
6.7 Estracción de la Introducción 160
6.8 Reinserción de la Introducción al final 161
6.9 Reinserción de la sección *Ideas Generales* 162

Resumen

El Proyecto Lope se concibe como un Proyecto de Investigación en el campo de los entornos avanzados de desarrollo de *Software*. Este tipo de entornos trata de cubrir buena parte de las actividades que comprenden el ciclo de vida del software, yendo por lo tanto mucho más allá de las tareas de programación.

Una de las ideas clave del proyecto es que define realmente un *metaentorno*, o entorno para definir entornos de desarrollo, ya que resulta lo suficientemente abierto como para poder implementar dentro lo que un entorno necesite: compilador de cualquier lenguaje, editores, herramientas gráficas. . . Es decir, no fuerza el uso de ninguna metodología de desarrollo ni lenguaje de programación: son aspectos totalmente configurables por el usuario.

Este trabajo desarrolla uno de los subsistemas básicos del prototipo del metaentorno Lope: el **Subsistema de Visualización**. Este subsistema se diseña con la idea de poder representar cualquier información que maneje el Sistema, de cualquier manera posible, ya sea textual o gráfica; si bien en este trabajo quedan pendientes de estudio las primitivas gráficas.

El documento de este trabajo se ha organizado en los siguientes capítulos:

1. **Entornos de desarrollo de software**, donde se realiza una introducción a los entornos de desarrollo en sus diversas variantes.

2. **El Entorno** LOPE, donde se describe en líneas generales todo el Sistema en el que se enmarca este trabajo.

3. Las **Ideas generales sobre el subsistema de visualización** se exponen en este capítulo, sirviendo de introducción a la comprensión del elemento principal: el *formato*.

4. Los **Ejemplos de formatos** que se exponen aquí complementan al capítulo anterior con el objetivo de afianzar los conceptos descritos.

5. Las **Características de la implementación** se describen aquí en profundidad, explicando estructuras de datos, algoritmos, etc., utilizados.

6. El **Ejemplo de utilización** describe un programa de prueba con el objetivo de dar ideas al lector sobre cómo podrían construirse herramientas en la capa de edición de LOPE.

7. Finalmente, se incluye todo el **código fuente** implementado.

Capítulo 1

Entornos de desarrollo de software

1.1 Conceptos básicos

En este capítulo nos introduciremos en los entornos de desarrollo, resumiendo los diferentes tipos que existen así como la situación actual y algunos proyectos desarrollados en este contexto.

El concepto fundamental aquí es el de “entorno”. Para nosotros, un **entorno** es un conjunto de herramientas hardware y software, más o menos integradas, utilizadas para construir sistemas software, es decir, desde aplicaciones sencillas hasta desarrollos completos de proyectos de ingeniería.

Inicialmente, los entornos se reducían prácticamente a la presencia de un compilador junto a un cargador-montador. Posteriormente se han ido desarrollando herramientas que facilitaban la tarea del programador, tales como editores o depuradores. Más adelante se han ido desarrollando otras herramientas de más alto nivel que permitía implementar técnicas de diseño.

Actualmente, pese a la evolución descrita, se utilizan mucho los que conocemos como "**entornos de programación**". Estos entornos solo soportan la fase de codificación del ciclo de vida del software; es decir, incluyen herramientas para compilar, editar y depurar; y se utilizan siempre que los trabajos de programación no sean excesivamente extensos (esto se conoce como "*programming in the small*").

Sin embargo, muchas veces las herramientas utilizadas por los programadores se encuentran débilmente acopladas, es decir, son independientes. El objetivo ideal sería conseguir la *integración* en los entornos, que proporcionen al usuario un interfaz consistente, en una herramienta única que no deje ver divisiones entre las funcionalidades ofrecidas (edición, compilación. . .). Smalltalk es un ejemplo de intento de integración en este sentido.

Con el nombre de "entornos para lenguajes compilados" se conocen las uniones de lenguajes de alto nivel con ciertas herramientas sencillas, con el propósito de facilitar la realización de entornos más complejos. Para facilitar la tarea se apoyan desde el propio lenguaje utilizado conceptos como modularización y abstracción de datos, como sucede con el lenguaje **Ada**. Sin embargo, este tipo de entornos suele contar con un conjunto de herramientas independientes, por lo tanto con baja integración.

Los "entornos para lenguajes interpretados" apoyan con una desarrollada funcionalidad e integración entre sus herramientas, la flexibilidad necesaria para el desarrollo de programas experimentales.

Los "**entornos de desarrollo de sistemas**" tratan de ayudar a reducir las complejidades en tiempo y escala. Cuando el proyecto es grande, y muchos programadores tienen que trabajar en él, surgen problemas muy distintos a los habituales problemas de programación. Con frecuencia, el éxito del proyecto dependerá de la buena voluntad de los miembros del equipo de programadores para solucionar estos problemas nuevos que estamos comentando. Las herramientas para los entornos de desarrollo de sistemas tratan de apoyar la resolución de

estos problemas, y lo hacen proporcionando mecanismos de control de versiones o historias de módulos, entre otros.

Finalmente, llegamos a los "**entornos de desarrollo de software**". Estos entornos intentan automatizar o apoyar todas las actividades del ciclo de desarrollo del software, es decir, intentan incluir tareas de programación "a lo grande" como la gestión de proyectos, de configuración, etc. También supone el soporte del mantenimiento del software a largo plazo y a gran escala. Las dos áreas de investigación actuales que cubren estos aspectos son los entornos de programación y de desarrollo de sistemas. Ya se acepta que la mejora de la productividad en el desarrollo de software depende de la disponibilidad de entornos integrados de apoyo a los proyectos, en los que a ser posible puedan los desarrolladores elegir qué herramientas usar (metodología, lenguaje. . .).

En resumen, se considera que un *entorno integrado de desarrollo de software* ideal, no solo automatizará ciertos aspectos repetitivos del desarrollo, sino también mantendrá la consistencia entre las partes del sistema desarrollado, desde los documentos de análisis y diseño hasta el código desarrollado y los propios manuales del usuario. Además, debería proporcionar al desarrollador una visión amplia del sistema a través de una herramienta de usuario flexible y de fácil manejo.

1.2 Clasificación de los entornos de desarrollo

Para situar el proyecto del que forma parte este trabajo, veremos a continuación diversos criterios de clasificación de entornos de desarrollo de software.

1.2.1 Cobertura del ciclo de vida

El desarrollo de un sistema software pasa o debe pasar, por diferentes etapas que van mucho más allá de la fase de codificación: diseño, documentación...

El ciclo de vida del software es el conjunto de estas etapas, ordenadas por el momento de realizarse. Existen diversos modelos propuestos, como el ciclo de vida en cascada o en espiral.

La Ingeniería del Software señala la necesidad de adoptar un modelo de ciclo de vida, que garantice el seguimiento adecuado de las diferentes fases del proyecto. Por ello, se ve necesario el que los entornos de desarrollo del software apoyen todas las fases del ciclo de vida que sea posible, automatizando algunas de ellas.

El entorno de desarrollo debe considerar por lo tanto la división del proceso en diversas fases o actividades, considerando éste como un proceso continuo y evolutivo.

Las herramientas **CASE** actuales, en la mayoría de los casos, han apoyado a la Ingeniería del Software de la manera más sencilla: cubriendo en profundidad alguna actividad concreta del ciclo de vida. Así pues, existen herramientas que apoyan las fases de análisis, diseño, planificación, etc; implementando alguna técnica concreta. Es decir, la mayoría de estas herramientas cubren una o unas pocas actividades del ciclo de vida, pero pocas son las que contemplan el proceso de desarrollo desde el punto de vista de ser continuo y evolutivo, puesto que eso hace necesario la colaboración estrecha o **integración** entre diversas herramientas.

1.2.2 Tipo de software desarrollado

En función del software desarrollado podemos encontrar diferentes entornos especializados. Es decir, de la misma forma que existen entornos pensados para programar en un lenguaje u otro, también tenemos entornos pensados especialmente para desarrollar sistemas de tiempo real, de gestión, distribuidos, etc.

El coste de generación de cada entorno es en sí mismo elevado, y son muchos los tipos de software existentes. Por ello, resultará útil la reutilización de un

mismo entorno para diferentes tipos de sistemas a desarrollar. La idea aquí será generar con poco coste los entornos especializados para los distintos tipos de software desarrollado. La idea del "*meta-entorno*" o generador de entornos es precisamente ésa.

1.2.3 Clasificación de Dart [6]

Esta clasificación, poco formal, fue propuesta por Dart [6]. En dicha clasificación se observa la ausencia de un lugar para los llamados *entornos de cuarta generación*, los que vienen a presentarse como más orientados al usuario final, citando como ejemplo los entornos visuales para producir software de consulta de bases de datos.

Quitando esta clase de entornos, Dart [6] los clasifica en cuatro categorías: entornos orientados al lenguaje usado, a la estructura, basados en herramientas y basados en la metodología. Veámoslos brevemente.

Entornos centrados en el lenguaje

Este tipo de entornos se construyen sobre un lenguaje, proporcionando herramientas de desarrollo apropiadas para ese lenguaje. Son los entornos que ofrecen limitaciones para tareas de programación extensa, y en general altamente interactivos.

Estos entornos permiten que el código escrito en el lenguaje para el que se ha pensado el entorno, pueda desarrollarse, ejecutarse, depurarse y cambiarse rápidamente.

Al estar orientados a un lenguaje concreto, pueden incluir diversas técnicas de implementación orientadas a ese lenguaje. Además, permiten construir las aplicaciones interactivamente por incrementos, pudiendo así partir de sencillos prototipos que pueden probarse para luego ir construyendo todo el sistema.

Dado que los propios lenguajes de alto nivel no suelen soportar adecuada-

mente actividades propias del desarrollo de grandes sistemas (como gestión de versiones del código) estos entornos pueden incluir facilidades para soportar este tipo de actividades.

En general, son útiles para la fase de codificación del desarrollo, proporcionando técnicas de compilación incremental o de interpretación que ayudan a reducir el impacto de los pequeños cambios en el código que tienen lugar durante el mantenimiento. Sin embargo, al ser tan especializados, generalmente solo soportan un lenguaje e incluso son característicos de una determinada plataforma, dificultando por tanto la portabilidad del código.

Como ejemplo de esta clase de entornos citamos dos. El primero de ellos es el de un lenguaje interpretado como puede ser el **LISP** o el **BASIC**. Son entornos interactivos sencillos donde el usuario directamente escribe líneas de código que son interpretadas de inmediato. Por ejemplo, con BASIC el entorno se comportaba de la manera siguiente: si la línea introducida por el usuario estaba precedida de un número, se consideraba parte de un programa a almacenar. Si no, las órdenes de la línea BASIC se ejecutaban de inmediato (entre ellas, se incluía la orden de ejecución del programa almacenado).

El segundo entorno que mencionamos, por citar uno reciente, es el **Smalltalk**. Este lenguaje orientado a objetos, pero mucho más educativo que el **C++**, se acompañaba de un entorno gráfico con ventanas y ratón, de tal forma que la aplicación que construía el usuario resultaba ser una modificación del propio entorno. Es decir, toda aplicación reutiliza los componentes que ya existen como son los menús y las ventanas, todo ello expresado mediante una jerarquía predefinida de clases. El editor de clases que incluye es un elemento principal que permite *navegar* fácilmente por la jerarquía de clases que se está construyendo en cada aplicación.

Entornos orientados a la estructura

Estos entornos incorporan técnicas que permiten manipular estructuras directamente, evitando la dependencia de un determinado lenguaje. En este tipo de entornos, el editor es una parte fundamental, pues a través de éste el usuario puede crear y manipular estructuras del código. Hay que decir que el programador, antes de escribir código, concibe de éste las estructuras correspondientes. La presencia de un editor de estructuras puede ayudar a evitar ciertos errores de codificación, como por ejemplo olvidarse de cerrar un bloque `WHILE` de *Pascal*. Es decir, un editor de estructuras recibe a la entrada comandos para construir código fuente analizando éste, proporcionando palabras clave del lenguaje de manera automática, etc. E internamente, el editor no almacena el código como texto, sino como jerarquía de estructuras (árboles de sintaxis abstracta o AST[1]).

Los editores de estructura y sus descendientes, los dirigidos por la sintaxis, se han perfilado como alternativa teórica a los editores de texto y como herramienta para crear y manipular programas. Pero además resultan interesantes a la comunidad investigadora como poderosas interfaces de entornos de desarrollo de programas basados en el conocimiento, la incrementalidad y la integridad. Tienen además un elevado interés didáctico al ayudar a liberar al usuario principiante de errores sintácticos, al tiempo que proporcionan sencillos interfaces de usuario basados en menús. Combinado con el uso de plantillas de programas se ve como una interesante herramienta para enseñar a programar a los principiantes.

Además, un objetivo importante es la integración: que el usuario no sea consciente de que se encuentre en determinado momento en el editor, ahora pase a compilar y luego a depurar. Por ejemplo, el visualizador no solo debe mostrar el código fuente sino tal vez información dinámica resultante de la ejecución; como resaltar errores sintácticos o semánticos. Interlisp ha sido un

[1]Del inglés *Abstract Syntax Tree*

intento de acercarse a este objetivo.

En realidad, lo que puede estar sucediendo es que, mientras el usuario edita el programa, tiene lugar un proceso de compilación en segundo plano e incluso de ejecución, de modo que el programador va conociendo errores a medida que va realizando el programa.

Estos ideales solo son posibles, hoy por hoy, mediante el uso de entornos orientados a la estructura.

Citando un ejemplo, el "*Cornell Program Synthesizer*" de Teitelbaum, es un entorno para desarrollo de programas en **PL/1**. En este entorno, se opera directamente sobre las estructuras codificadas del programa. Por ejemplo, se pueden cambiar interactivamente los contenidos variables de un bloque de sentencia *IF... THEN... ELSE...*, produciéndose efectos inmediatos de ejecución.

Entornos de herramientas

Estos tipos de entornos, también llamados "*toolkit*", son una colección de pequeñas herramientas que incluyen apoyo independiente del lenguaje adecuado para proyectos grandes, tales como la gestión de configuraciones o control de versiones, aunque la principal misión no deja de ser el soporte de la fase de codificación.

El entorno parte del propio sistema operativo, y añade herramientas tales como un compilador, un editor, ensamblador, depurador, control de versiones, etc. Existen diversos ejemplos de entornos de herramienta como el Unix/PWB (Unix Programmer Workbench) o el que proporciona DEC-VMS para los Vax. También hay desarrollos prototípicos como el Portable Common Tool Environment (PCTE).

Uno de los más conocidos es el que proporciona Unix/PWB. Fue desarrollado para correr en diversas plataformas, de tal forma que el usuario tiene siempre las mismas herramientas de trabajo, independientemente de que lo haga en una

máquina IBM o Digital. Esto ahorra al programador conocer los detalles de cada máquina, ahorrando también el trabajo que supone el hecho de cambiar de hardware[2].

Unix/PWB está orientado al archivo, de modo que el usuario puede interconectar herramientas usando éstos, o bien "*pipes*" de Unix. Esto produce la necesidad de tener en cuenta las convenciones específicas de entrada y salida de las herramientas: si no concuerdan, no funcionará (por ejemplo, la salida del preprocesador de C debe ir en cierto formato, un fichero objeto producido por el compilador debe ir en otro, etc.) Sin embargo, no es difícil mantener un estándar en los formatos de salida y entrada de esas pequeñas herramientas, y el resultado es disponer de un entorno más abierto (incorporar nuevas herramientas es perfectamente posible), y más flexible (las tareas complejas a realizar con esas herramientas pueden automatizarse utilizando, por ejemplo en Unix, *scripts* de *shell*).

Entre sus inconvenientes citamos la falta de una interfaz de usuario uniforme; y las limitaciones del mecanismo de intercambio de información basado únicamente en archivos de texto.

Entornos asociados a la metodología

Este tipo de entornos implementan apoyos para un rango amplio de actividades en el proceso de desarrollo de software, incluyendo tareas tales como gestión de proyectos, es decir, tareas para programación con muchos ("*programming in the many*").

Cada uno de estos entornos apoyan una metodología particular para el desarrollo de software. Éstas pueden ser de dos clases: metodologías de desarrollo y metodologías para gestión del proceso. Las primeras solucionan varios pasos del ciclo de desarrollo del software, proporcionando métodos de especificación, di-

[2]Por ejemplo, en cualquier máquina Unix usaremos la orden `cc` para compilar un programa en C, con opciones muy similares

seño, validación, verificación y reutilización. Estas metodologías pueden basarse en técnicas más formales (como las especificaciones con máquinas de estado o redes de Petri) o menos (como los diagramas de flujo de datos); siendo estas últimas las que más se usan. Las herramientas que hoy llamamos CASE son, en muchos casos, ejemplos de entornos basados en metodologías de desarrollo. Por ejemplo, la herramienta *Westmount* implementa entre otras la técnica de DFDs[3], y chequea automáticamente la consistencia entre esos diagramas, y también entre éstos y las descripciones de cada proceso (llamadas miniespecificaciones) y el diccionario de datos.

Las metodologías para la gestión del proceso, incluyen facilidades de gestión de configuraciones y control de versiones, gestión de proyectos, planificación de tareas, etc. Son herramientas que tratan de garantizar consistencia a estas actividades, automatizando parte de ellas.

1.3 Integración de herramientas

Recordemos que un objetivo ideal de los entornos de desarrollo de software es la *integración* entre las distintas herramientas que lo componen.

Intentemos definir entonces la integración entre dos herramientas:

> Se dice que dos herramientas están integradas cuando pueden colaborar para alcanzar un objetivo común que engloba los objetivos parciales para los que fueron concebidas inicialmente.

Para determinar si dos herramientas están integradas o no, existen diferentes criterios que veremos. También existen diferentes niveles de integración. Dado que la idea del entorno es la integración entre sus herramientas, podemos dar una nueva definición de entorno:

[3] Diagrama de Flujo de Datos

> Un entorno de desarrollo es una plataforma de integración que proporciona una infraestructura sobre la que diferentes herramientas pueden integrarse con el fin de automatizar el proceso de desarrollo del software.

A esto hay que añadir, con el concepto de *metaentorno* en mente, la idea de que el entorno de desarrollo debe ser una plataforma de ejecución y también de generación de herramientas.

1.3.1 Criterios de integración

En Thomas [13] se citan cuatro criterios para decidir el nivel de integración entre herramientas.

1. Integración por compartición de datos.

   Este tipo de integración se produce cuando las herramientas comparten un conjunto de informaciones. La integración deberá conseguir una interoperabilidad entre herramientas, evitando redundancia de datos, o inconsistencias entre herramientas.

   Según se cita en Chen [2], esta compartición puede tener lugar por *comunicación directa*, es decir, la salida de una herramienta es la entrada de otra; por *ficheros*, por *mensajes* (propio de sistemas distribuidos) o por *repositorio* (base de datos común al que acceden todas las herramientas).

   El primer caso, la transferencia directa de datos entre herramientas, es lógicamente el más eficiente, pero también el más complicado ante una perspectiva de integración de muchas herramientas.

   El segundo método, la transferencia por ficheros, plantea el mecanismo más sencillo de integración, sobre todo si se adopta un estándar para el intercambio (el formato CDIF, *CASE Data Interchange Format* es un intento en este sentido).

El método más flexible es el último de los mencionados, la base de datos común. Debido a la creciente complejidad de la información que las herramientas necesitan compartir, se determina que el modelo relacional resulta insuficiente para el repositorio, orientándose las líneas de investigación hacia las bases de datos orientadas a objetos, dada su capacidad para almacenar reglas y procedimientos además de los propios datos.

Sería deseable la aparición de un estándar en este campo, pues actualmente solo existen intentos por parte de diversos fabricantes, incompatibles entre sí, lo cual tiene el inconveniente de tener que ceñirse únicamente a las soluciones que ofrezca un determinado fabricante.

2. Integración por control. Con este tipo de integración, en una herramienta pueden producirse eventos que disparen funciones de otras herramientas. Para ello existe un nivel de ejecución con interfaces mediante las que las herramientas pueden invocarse entre sí. Es necesaria la compartición de datos entre herramientas para poder suministrar servicios entre sí, por lo que esta clase de integración exige previamente la integración de datos.

   La integración por control puede conseguirse mediante un sistema de mensajes entre herramientas que permita realizar invocaciones a nivel herramienta-herramienta, herramienta-servicio y servicio-servicio, entendiendo por *servicio* una de las diversas acciones que pueda realizar la herramienta, caracterizadas por no ser divisible (por ejemplo, la orden de compilar un código fuente).

3. Integración en el nivel de presentación.

   Es un campo en el que está interesada toda la comunidad informática (no solo el campo de las herramientas de desarrollo); dado que la integración a este nivel permitiría minimizar el esfuerzo y el tiempo de aprendizaje, al integrarse las interfaces de usuario de las herramientas.

Las herramientas integradas a este nivel presentan una interfaz hombre-máquina consistente, de tal modo que el usuario puede aprender a manejar todas las herramientas prácticamente al aprender una de ellas.

La integración a este nivel se logra al tomar como objetivos los habituales de los entornos *amigables*, es decir, aquellos que proporcionan un sistema de diálogo con las herramientas común, que usan formas de presentación adecuadas al modelo mental que el usuario tiene del entorno o que mantienen en la medida de lo posible toda la información útil a la vista del usuario. La integración se debe basar en un sistema de interfaz normalizado como **Motif** aunque esto no es suficiente: para conseguir el objetivo de adecuación al modelo mental del usuario será necesario diseñar los diálogos de una manera apropiada.

4. Integración del proceso.

   Cuando exista este tipo de integración, las herramientas apoyarán u obligarán a seguir los pasos de una determinada metodología de desarrollo. Se trata de que en el entorno, se representen y traten de modo natural los diferentes *pasos* de la metodología elegida, así como los *eventos* (por ejemplo, al compilar un módulo sin errores se dispara automáticamente la ejecución de la fase de pruebas sobre éste) y las *restricciones* (limitaciones que imponga la metodología, tales como que el encargado de las pruebas de un módulo no deba coincidir con el autor del mismo).

   En general, para conseguir este tipo de integración se requerirá una integración buena tanto a nivel de datos como de control.

1.3.2 Niveles de integración

Si utilizamos como criterio de integración el primero que veíamos, el de compartición de datos, podríamos distinguir varios *niveles de integración*. Son los siguientes:

- Nivel de comunicación: Las herramientas simplemente comparten un medio por el que transmiten datos, sin estructura; de tal modo que estos datos podrían ser interpretados por cada herramienta de manera distinta.
- Nivel léxico: En este nivel, existe un acuerdo entre las herramientas acerca del formato de los datos. El problema es que éste es muy dependiente de las propias herramientas, lo que dificulta la integración de otras nuevas.
- Nivel sintáctico: En este nivel, hay un acuerdo acerca de las reglas que rigen la construcción de las estructuras de datos, sin que dependa del tipo de herramienta.
- Nivel semántico: añade al anterior una definición de las operaciones que pueden realizarse sobre las estructuras de datos y su significado. Esta información puede ser parte del código de las herramientas o, mejor, estar en la propia base de datos, ya que permite extender el entorno con nuevas herramientas fácilmente, si bien resulta menos eficiente.

Capítulo 2

El Entorno Lope

En este capítulo revisamos las ideas generales sobre el Entorno de Desarrollo Lope.

2.1 Introducción

El metaentorno Lope pretende ser la base de los nuevos entornos avanzados de desarrollo de *software*. Las características del mismo son:

- Se apoya en una base de datos o repositorio unificado, que almacena la información de modo no redundante. Toda la información acerca del software en desarrollo se encontrará, pues, en dicha base de datos. Todos esos datos pueden incluir múltiples referencias entre ellos.

- El entorno es autocontenido: todos los esquemas de datos se almacenan a su vez en la misma base de datos o repositorio común. Un entorno concreto desarrollado en el metaentorno Lope incluirá en la base de datos tanto la definición del entorno concreto como el software que se está desarrollando.

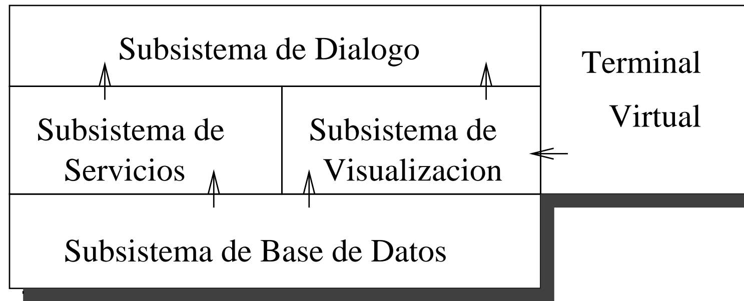

Figura 2.1: Arquitectura del Sistema LOPE

- Un entorno concreto, por lo tanto, se definirá con un conjunto de esquemas, reglas, etc. que se codificarán y almacenarán en la base de datos. Una máquina implementará dicha base de datos así como los subsistemas necesarios para ejecutar las reglas, programas, etc. incluidos en ella, para materializar un determinado entorno. Como caso particular se tendrá el entorno para desarrollo de entornos, que se utilizará durante la generación de un entorno concreto.

- Dentro de las clasificaciones propuestas en el capítulo anterior, LOPE será un entorno *integrado*, orientado a la estructura (es decir, con editor dirigido por la sintaxis), basado en el método, pero flexible (puede implementar cualquier metodología).

En definitiva, pues, una idea fundamental de LOPE es tener todas las herramientas necesarias para implementar cualquier entorno de desarrollo de *software*, pero sin limitarse a ninguno de ellos.

El prototipo de implementación del Sistema se ha organizado en la arquitectura cuyo esquema se muestra en la figura 2.1. Como vemos, el Sistema consta de cinco subsistemas:

1. El subsistema de base de datos, que almacena toda la información utilizada en el sistema, incluyendo por lo tanto, datos del usuario, esquemas de datos, programas que codifican los servicios, formatos de visualización,

esquemas de diálogo, etc. Como se ve, la base de datos almacena no solo datos, sino también los esquemas o **tipos** que especifican dichos datos.

2. El subsistema de servicios, que lleva asociados unos esquemas de definición de operaciones de manipulación, e implementa un intérprete que ejecuta dichas operaciones; dándole así una componente *dinámica* al entorno.

3. El subsistema de visualización, desarrollado en este trabajo, que lleva asociados unos esquemas de formatos de visualización e implementa un intérprete que visualiza la información deseada de la base de datos según esos formatos.

4. El terminal virtual, que proporciona servicios de presentación e interacción con el usuario.

5. El subsistema de diálogo, con el que se definirán las herramientas invocables por el usuario; para lo que usarán unos esquemas de definición apropiados e implementará un intérprete de ejecución. Estas herramientas permiten al usuario examinar la información almacenada y modificarla o introducir nuevas informaciones.

Vemos pues, que la unión de los subsistemas de base de datos y el de servicios, resulta conjuntamente algo similar a una base de datos orientada a objetos (ya que los servicios proporcionan la parte dinámica que complementa a los datos, la parte estática).

Los sistemas de visualización y diálogo proporcionan mecanismos suficientes para definir herramientas de desarrollo, sean interactivas o no.

Seguidamente hablaremos en profundidad de los subsistemas necesarios en el desarrollo de este trabajo, esbozando tan solo ligeramente los restantes subsistemas.

2.2 El subsistema de base de datos

La base de datos se diseña jerárquicamente (es decir, hay datos compuestos que contienen otros datos, formando así *árboles de datos*). Dado que la base de datos admite referencias entre componentes, debe garantizar integridad. De hecho, garantiza las siguientes:

- Integridad estructural: Todo objeto tiene una estructura consistente con su esquema o tipo.
- Integridad referencial: Toda referencia no nula apuntará siempre a un objeto existente.
- Persistencia: La base de datos tiene soporte en disco, lo que garantiza que los datos se conserven entre sucesivas sesiones de trabajo con el sistema.

2.2.1 Objetos

La unidad de información manejada en la base de datos es el **objeto**. Un objeto es una unidad de información con identidad (es decir, al crearse un objeto se le asigna un identificador que se conserva durante toda su vida). Este identificador se utiliza como referencia al objeto. Todo objeto es conforme a un esquema o tipo, y tiene un valor. El esquema se representa con un símbolo y éste es el nombre con que dicho esquema se ha declarado en la base de datos. El valor debe ser consistente con la definición del tipo, y puede ser, como los tipos, simples o compuestos. Los primeros, son valores simples de cualquiera de los siguientes tipos:

- Números (enteros)
- Cadenas de texto
- Símbolos

- Referencias (identificadores que apuntan a objetos)

2.2.2 Esquemas de definición de datos

Los esquemas son objetos de la base de datos que describen los tipos sobre los que se pueden construir dichos objetos. Los esquemas pueden ser todo lo complejos que se desee, partiendo de los siguientes tipos básicos:

- Número entero (`.integer.`)
- Texto o string (`.text.`)
- Símbolo (`$`)
- Referencia o identificador (`.integer.`)
- Directorio (`%`)

Para generar esquemas de objetos compuestos, se prevén los siguientes tipos:

- Tupla: `+ tipo1 + tipo2 + tipo3 ...`
- Alternativa: `| tipo1 | tipo2 | tipo3 ...`
- Iteración: `* tipo`
- Referencia: `^ tipo`
- Sinónimo: `tipo`

Todos los tipos se nombran mediante símbolos, ya sean básicos o definidos. Para referirnos a un mismo esquema con diferentes nombres puede usarse el tipo sinónimo. Los objetos simples son objetos que contienen valores de un tipo básico, representables externamente de manera predefinida (salvo los identificadores o referencias, que se conciben para un uso interno) y también

suministrables por el usuario. Para cada uno de ellos existe un valor *nulo*: el cero para los números, el símbolo nulo para los símbolos, etc.

Los objetos de tipo referencia son similares a los enteros, pero ese valor es en realidad un identificador de objeto (puntero a éste, pues). Por lo tanto, no serán representables externamente ni suministrables por el usuario (no tiene demasiado sentido representar un identificador que es manejado internamente por la base de datos y que puede cambiar dinámicamente).

Los objetos de tipo tupla son objetos compuestos con N componentes, donde cada componente es un nuevo objeto del tipo indicado en la tupla. Los tipos de las componentes deben ser diferentes, por lo que en caso de querer designar objetos del mismo tipo deberá usarse el sinónimo. Cuando un objeto tupla se crea, sus componentes se inician a los valores nulos de cada tipo.

Los objetos de estructura alternativa son objetos compuestos que en cada momento tienen una única componente de alguno de los tipos especificados. Al ser creados se ha decidido que tengan por valor, el valor nulo de la primera componente indicada en la declaración de la alternativa.

Los objetos iteración son una lista de N objetos del mismo tipo, el indicado en la lista. Se puede referenciar a cada componente mediante su número de orden. Al ser creados contienen 0 componentes.

Los objetos de directorio son objetos similares a las listas, con la salvedad de que cada componente puede ser de cualquier tipo, y que además cada una de ellas se identifica mediante una etiqueta (símbolo). Es decir, se establece una asociación entre cada objeto componente con un símbolo. Como en las listas, podemos acceder a las componentes por su número de orden en lugar de su nombre.

2.2.3 Estructura de la base de datos

Habíamos comentado al principio que la base de datos era de tipo jerárquico. Esto quedará más claro ahora, una vez definidos los esquemas de datos, al indicar que una base de datos es en realidad un directorio (que llamaremos *raíz*) y que contendrá dentro todos los demás objetos simples o compuestos que se deseen, incluyendo nuevos directorios.

En el directorio raíz se definen de manera predefinida directorios para los distintos subsistemas. En concreto, son los siguientes:

- Directorio `schema`. Este directorio contiene dentro la definición de todos los esquemas de datos. Ningún objeto de la base de datos carecerá de un esquema en este directorio que lo defina, a menos que sea de un tipo predefinido.
- Directorio `format`. Este directorio se define para contener todos los formatos del subsistema de Visualización, objeto del presente trabajo de fin de carrera.
- Directorio `service`. Este directorio contendrá los códigos de las operaciones de manipulación de datos.

Pero además, existe un objeto que no cuelga del directorio raíz , Se trata de la **tabla de símbolos**, que establece la relación entre el código interno de cada símbolo y el nombre externo con el que se le conoce.

2.2.4 Representación de los objetos

Los objetos se codifican internamente como una estructura de campos para el esquema del objeto y para su valor, que a su vez tendrá un tipo. Los objetos simples contienen el valor dentro de esta estructura mientras que los compuestos contienen como representación del valor la secuencia de identificadores de los

objetos componente. En el caso de los directorios, la secuencia es de pares (nombre, identificador).

Para representar externamente la base de datos, utilizaremos una representación simbólica, la misma que ha sido utilizada en la utilidad de exportación e importación de objetos[1].

La representación simbólica utilizada para los objetos simples es dada de la forma: `(tipo) valor`, donde el valor será directamente un número para el caso del tipo entero o algún sinónimo; una flecha seguida del número para el caso de una referencia, un nombre para los símbolos; y un texto encerrado entre comillas para los tipos texto o sus sinónimos.

Los símbolos y punteros nulos se representan con $\emptyset$, los textos nulos con una cadena vacía, y los enteros nulos con el 0.

En los objetos compuestos, el valor se representará con una lista con las componentes encerrada entre corchetes. Junto al corchete de apertura se situará un símbolo para indicar la clase de objeto compuesto (es decir, la apertura se hará con `[+` para tuplas, `[|` para alternativas, `[*` para listas y `[%` para directorios).

Cada componente irá en la notación descrita, es decir, el par `(tipo) valor`, pero precedida del nombre o etiqueta en caso de tratarse de un directorio.

Para referirnos a una componente se usará el número de orden (válido para seleccionar componentes en tuplas, alternativas (=1), directorios e iteraciones) o un símbolo (válido para seleccionar componentes en tuplas o alternativas, por el tipo; o en directorios por la etiqueta).

2.2.5 Codificación de esquemas

Todo esquema correspondiente a cada tipo de objeto se encuentra definido en el directorio de esquemas, con el nombre de ese tipo. Los esquemas se codifican

[1]Esta utilidad, llamada `ObjXPort`, fue realizada durante 1995 por el grupo de colaboración del Proyecto Lope

con estructuras que almacenan los parámetros (símbolos) apropiados. El tipo de un esquema indicará la estructura o esquema de definición correspondiente. Estos esquemas de definición se llaman *metaesquemas*, son fijos y no existen en la base de datos, siendo designados mediante símbolos predefinidos. Estos metaesquemas son:

```
Entero (.integer.)     ($) .integer
Texto (.text.)         ($) .text.
Símbolo ($)            ($) $
Referencia (^)         ($) ^
Directorio (%)         ($) %

Tupla (+)              (*) $
Alternativa (|)        (*) $
Iteración (*)          ($) $
```

Los esquemas definidos se codificarán de la siguiente manera:

```
+ tipo1 + tipo2 ...         (+) [* ($) tipo1 ($) tipo2 ... ]
| tipo1 | tipo2 ...         (|) [* ($) tipo1 ($) tipo2 ... ]
* tipo                      (*) tipo
^ tipo                      (^) tipo
tipo                        ($) tipo
%                           (%) %
```

Para aclarar esto, supongamos que queremos declarar un esquema para definición de documentos de la siguiente manera:

```
documento ::= + titulo + autor + secciones ;
titulo ::= .text. ;
```

```
autor ::= .text. ;
secciones ::= * seccion ;
seccion ::= .text ;
```

La codificación interna será entonces la de la figura 2.2.

Esto supone la inclusión en el directorio de esquemas, de los objetos anteriores con esos esquemas.

Finalmente, veamos en la notación explicada un objeto documento, en la figura 2.3.

2.3 El Subsistema de ejecución de servicios

Este subsistema implementa un intérprete de ejecución de operaciones de manipulación de datos. Exporta lo necesario para definir e invocar *servicios*, para las herramientas de la capa de edición. Los servicios se codificarán en la base de datos, como cualquier otra información que maneje el Entorno. El subsistema se diseña como una máquina virtual que ejecuta operaciones escritas en un lenguaje con notación polaca inversa (RPN), de manera que el intérprete se implementará basándose en una pila de ejecución.

2.3.1 Instrucciones elementales

Son operaciones que pueden realizarse en la base de datos, así como instrucciones de control para permitir programas con estructura de selección, bucles e invocación a otros servicios.

```
schema documento (+) [*
      ($) título
      ($) autor
      ($) secciones
]

schema secciones (*) sección

schema sección (+) [*
      ($) título
      ($) párrafos
]

schema párrafos (*) párrafo

schema párrafo ($) .texto.

schema título ($) .texto.

schema autor ($) .texto.
```

Figura 2.2: Codificación interna del esquema documento

```
(documento) [+
      (título) "Don Quijote de la Mancha"
      (autor) "Miguel de Cervantes"
      (secciones) [*
         (sección) [+
           (título) "Prólogo"
           (párrafos) [*
              (párrafo) "Desocupado lector, ..."
              (párrafo) "Sólo quisiera dártela ..."
                       . . .
                     ]
         ]
         (sección) [+
            (título) "Capítulo primero"
            (párrafos) [*
               (párrafo) "En un lugar de la Mancha, ..."
               (párrafo) "Es, pues, de saber que ..."
                        . . .
            ]
      ]
]
```

Figura 2.3: Codificación interna de un documento

2.3.2 Codificación de las expresiones

Una expresión es una secuencia de instrucciones, y en la base de datos se almacenará como objeto con valor simple utilizando el tipo texto.

2.3.3 Ejecución de las expresiones

Como hemos indicado, las expresiones son ejecutadas por una máquina virtual con estructura de pila. Es decir, se trata de un autómata que extrae instrucciones del código de la expresión y las ejecuta sobre la pila de evaluación. Habrá además una pila de control que contendrá la información necesaria para invocar una expresión desde otra.

2.3.4 Esquemas de definición de servicios

Un servicio es una operación de manipulación que puede ser invocada por el usuario. El servicio es como una transacción, es decir, se realizará en su totalidad o no se realizará; de forma que nunca será interactivo: todo servicio que comienza continuará hasta su final.

Un servicio es una acción con un conjunto opcional de parámetros, y devuelve un código de salida para indicar si la operación tuvo éxito o no.

Se define el servicio como un conjunto de pares `guarda -> acción`, de tal modo que si se cumple la guarda se deberá ejecutar la acción sin errores. La evaluación de las guardas tendrá lugar por orden de aparición con lo que se ejecutará la primera acción por satisfacción de su guarda. Si ninguna guarda se cumple, el servicio fallará y no cambiará ninguna información.

Se prevé la definición de macros (expresión representada con un nombre simbólico) con el fin de simplificar la escritura de guardas y acciones. Ejecutar una macro es equivalente a ejecutar la expresión representada.

2.4 El subsistema de visualización

Este subsistema es el objeto de este trabajo. Aquí resumiremos brevemente su funcionalidad.

El subsistema maneja las vistas de los objetos, siendo una vista una asociación entre tres clases de objetos: las cajas, los formatos de visualización, y el objeto representado.

El formato de visualización, que se almacenará como objeto en el directorio de formatos de la base de datos, define de qué manera se va a representar un determinado objeto. El resultado de aplicar el formato al objeto será una caja, normalmente compuesta, que contendrá dentro una o más cajas, cada una de ellas simples o compuestas. Una caja simple contendrá texto terminal.

Las cajas tienen una posición en su caja madre, de tal forma que son una representación lógica directa de la representación física a la que dará lugar la vista.

En definitiva, diremos que el formato es una función que aplicada a un objeto devolverá como resultado una caja. Y el subsistema desarrollado en este punto será básicamente un intérprete de formatos que implementa esa función.

2.5 El subsistema de diálogo

Servirá para definir herramientas interactivas o no, invocables por el usuario. Forma lo que hemos venido llamando "*capa de edición*".

Una herramienta se apoyará en servicios definidos y consistirá en un esquema de diálogo con el que el usuario seleccionará el servicio a invocar junto a sus parámetros. Los editores son herramientas que trabajarán con el subsistema de visualización para implementar sus funciones.

2.6 El terminal virtual

El Terminal Virtual pretende proporcionar a los subsistemas de LOPE una abstracción de la terminal física, simplificando ésta, eliminando de este modo cuestiones como el refresco de pantalla, etc. También ofrece una abstracción de las posibilidades de una estación de trabajo, con ratón, teclado y múltiples ventanas en pantalla.

La implementación viene muy condicionada por el entorno en el que se está desarrollando el prototipo LOPE: las estaciones de trabajo *Unix*, por lo que se realiza utilizando el sistema de ventanas de texto *curses* o gráfico *X-Window*; si bien se pretende portar en un futuro al entorno *MS-Windows*.

2.6.1 Pantalla

La interacción con el usuario se realiza mediante un terminal gráfico único. Esta pantalla, ofrecerá una zona rectangular de presentación. El tamaño y sistema de coordenadas se determinará al iniciar la pantalla, fijando por lo tanto, tamaño de ésta (pixels), relación de aspecto, ancho del marco o la altura del título de las ventanas.

Habrá operaciones para abrir y cerrar la pantalla, y ésta podrá contener una o más ventanas.

2.6.2 Ventana

La unidad de presentación será la *ventana*, que tendrá un título y un marco, cuyo aspecto será el mismo para todas las ventanas de la pantalla.

Dentro podremos definir diversos objetos que serán:

- Paneles de texto

- Paneles gráficos

- Botones
- Campos de texto
- Ventanas hijas

A partir de estos objetos se construyen objetos más elaborados tales como menús, listas de selección, etc.

Habrá operaciones para crear, destruir y destacar (poner frente a las demás ventanas).

2.6.3 Panel de texto

Es una zona rectangular con un determinado tamaño y posición en la que podemos escribir textos en cualquier posición deseada, con un determinado tipo de letra y modo de presentación (normal, destacada, etc).

Hay funciones para editar el texto de un panel, y para determinar el tamaño del texto antes de presentarlo. Las dimensiones del panel serán fijas o variables, contando con elementos como *barras de scroll* para los casos en que sean necesarios.

2.6.4 Panel gráfico

Es una zona rectangular donde podemos dibujar gráficos, incluyendo también textos. Los gráficos son vectoriales y tendremos primitivas tales como líneas, elipses, rectángulos, polígonos o polilíneas, etc. Cada una de ellas podrá incluir también atributos tales como el tipo de trazo o el color de relleno.

2.6.5 Botón

Es un elemento visible que se podrá pulsar con el puntero del ratón, o una tecla asociada, e incluirá un rótulo que permitirá identificarlo. El botón tendrá

también varios modos de presentación, y su pulsación generará un evento correspondiente a la pulsación de la tecla equivalente.

2.6.6 Campo de texto

Permite representar en una línea un texto y la posibilidad de editarlo.

2.6.7 Teclado y ratón

Todas las teclas se codifican de manera uniforme, incluyendo teclas de función y botones del ratón. Las pulsaciones se obtienen con órdenes de lectura.

2.6.8 Procesos

El terminal virtual permitirá correr simultáneamente varias herramientas de Lope mediante procesos o corrutinas. El gestor del terminal virtual debe permitir que un proceso avance hasta que se solicite una lectura, en cuyo caso lo dejará suspendido hasta que se pulse la tecla de la lectura.

Un proceso podrá lanzar otro, bien sea terminando y cediendo el control, lanzando otros y continuando; o bien lanzando otros y esperando a que terminen.

Capítulo 3

Ideas generales sobre el subsistema de visualización

En este capítulo revisamos las ideas generales sobre el Subsistema de Visualización para el Entorno Lope, así como una especificación completa de su funcionalidad.

3.1 Introducción

Hemos visto en el capítulo anterior una descripción de los distintos módulos que integra el Entorno Lope. El Sistema de Visualización es uno de ellos, y se concibe como un mecanismo suficientemente abierto como para representar cualquier objeto interno de su Base de Datos, de la forma que el usuario desee, sin tener que obedecer a demasiadas restricciones.

Para ello, el Subsistema de Visualización se ha definido en base a los denominados *formatos*, objetos de la Base de Datos que especifican cómo debe visualizarse un objeto cualquiera. Actualmente, los formatos especifican cualquier representación basada en texto, quedando pendiente la especificación para

representaciones gráficas.

A partir de un objeto y un formato, el Subsistema generará una jerarquía de cajas o rectángulos que será una representación interna directa de lo que se verá en pantalla.

3.2 Especificación del subsistema

El subsistema maneja internamente dos clases de objetos: las cajas, que se construyen en jerarquías; y las vistas, que conecta la jerarquía de cajas que representa a un objeto, con su representación física en una ventana de la pantalla.

Para especificar cómo se construyen las jerarquías de cajas se utilizan los formatos, por lo que describiremos éstos en primer lugar.

3.2.1 Formatos de visualización

En esta implementación se manejan únicamente formatos de texto, quedando como líneas futuras de trabajo, la especificación e implementación de formatos para representaciones gráficas.

Definiremos un formato como un esquema de representación de un objeto, es decir, un formato aplicado a un objeto generará una representación visible de éste.

Antes de describir los formatos, insistimos que las representaciones visibles tienen la forma de jerarquías de cajas, es decir, jerarquías de rectángulos, de manera que un rectángulo tendrá unas determinadas dimensiones y coordenadas, correspondiendo éste a una parte de la representación visual del objeto. Como jerarquía, una caja podrá ser *terminal*, es decir, dentro contendrá texto para representar en las coordenadas indicadas; o bien puede ser *compuesta*, lo que indica que contendrá un determinado número de cajas *hijas*.

En el prototipo Lope, los formatos se almacenan como objetos la Base

de Datos, dentro de un directorio predefinido llamado `format` y creado en el directorio raíz. Al final, se muestran los esquemas que definen esta clase de objetos (ver figura 3.5).

Los formatos son los siguientes:

Formato conversión

Un formato **conversión** aplicado a un objeto, generará una caja terminal con un texto fijo. Dicho texto vendrá determinado por los parámetros contenidos en el formato, y el propio objeto. En general, el texto se generará como la combinación de textos fijos con el valor de uno o más atributos del objeto seleccionado (como puede ser su valor o el número de componentes que tiene, si es compuesto).

Por ejemplo, supongamos que deseamos que la representación asociada a un determinado objeto simple, es la siguiente:

```
Nombre: Juan
```

Donde "*Juan*" es el valor del objeto, y el resto es un texto fijo independiente de los atributos del objeto representado. El formato será de tipo *conversión* y se codificará internamente con:

```
Nombre: %v
```

Notar que `%v` es el símbolo que representa al atributo *valor* del objeto.

Para poder representar cualquier información acerca de cualquier objeto, se van a prever en el formato conversión los siguientes atributos:

- %v para el valor del objeto (será un texto si el tipo es símbolo o texto; y un número si es de tipo entero o referencia)
- %n para el nombre (etiqueta) del objeto
- %s para el nombre del tipo (esquema) del objeto
- %t para el nombre del tipo del valor del objeto
- %* para la posición (índice) del objeto en el objeto padre
- %# para el número de componentes del objeto
- y %% para indicar el propio símbolo "%"

Hay que tener en cuenta que no todos los objetos tienen todos los atributos. Por ejemplo, para el atributo "etiqueta" de un objeto, hace falta que éste exista como componente de un objeto tipo directorio.

Por ello se definen también unos *valores por defecto* para todos los atributos: los atributos que proporcionen un valor alfanumérico generarán una cadena vacía, mientras que los que proporcionen un valor numérico generarán un número 0. En el caso del valor del objeto, éste puede ser de tipo texto o numérico. En cada caso, el valor por defecto será, bien la cadena vacía, bien el número 0.

Formato alternativa

El formato alternativa es realmente una colección de formatos, de los que se eligirá uno en función de alguna propiedad del objeto.

Internamente se define como un directorio de formatos, con lo que cada formato tendrá una etiqueta que lo identifique.

Cualquiera de los atributos vistos en el formato conversión sirve como discriminante para seleccionar el formato aplicable. Si el atributo es un texto, se

seleccionará la alternativa cuya etiqueta coincida con ese texto. Si es numérico (por ejemplo, número de componentes o índice) se convertirá el número a texto y se buscará también la alternativa con ese texto por etiqueta.

Nótese que en el caso de usarse como discriminante el *valor* del objeto, se tomará siempre como alfanumérico, es decir, si el valor es de tipo numérico, se convertirá a cadena de caracteres y se buscará entre las etiquetas del directorio.

El formato alternativa puede tener un formato por defecto (es decir, no es obligatorio) que se eligirá en caso de no encontrarse ninguna alternativa que coincida con el atributo seleccionado. Si no existiera este formato, se generará una caja terminal vacía de dimensiones cero, lo que lógicamente equivale en términos de representación física, a no generar nada.

Formato composición

Mientras que con el formato anterior, se elegía un nuevo formato desde el actual; con el formato composición podremos elegir un nuevo objeto a representar. Para ello, el formato incluye una cadena de selectores, que se aplicarán al objeto de partida para obtener un nuevo objeto.

Cada selector podrá ser un valor numérico (para seleccionar una componente del objeto), un nombre (para seleccionar un tipo en objeto compuesto tipo tupla, o por etiqueta si es de tipo directorio), o también un símbolo especial que referencie al objeto padre (representado aquí por “<<”) o al objeto actual (representado por “^^”).

Además, el formato composición incluye un nuevo formato, a aplicar al objeto finalmente seleccionado.

Formato nombrado

Este formato incluye únicamente la referencia a otro mediante su nombre, que será aplicado al mismo objeto actual.

Formato alineación tupla

Con este formato empezamos los que generan cajas compuestas.

Este formato genera un conjunto de cajas alineadas entre sí mediante una serie de códigos de alineación que a continuación enumeramos. Cada caja hija se generará aplicando nuevos formatos al objeto actual.

Por lo tanto, el formato incluye una lista de N formatos nuevos, así como una lista de N-1 códigos de alineación.

Los códigos de alineación indican cómo se colocan relativamente dos cajas. Dichos códigos son:

- Alineación horizontal centrada: las dos cajas se alinean horizontalmente (la segunda a la derecha de la primera, y ambas centradas horizontalmente).

- Alineación horizontal superior: la segunda caja se coloca a la derecha de la primera, partiendo de la misma fila.

- Alineación horizontal inferior: la segunda caja se coloca a la derecha de la primera, alcanzando hacia abajo la misma fila.

- Alineación vertical centrada: la segunda caja se coloca debajo de la primera, centradas verticalmente.

- Alineación vertical izquierda: la segunda debajo de la primera, y justificadas ambas a la izquierda.

- Alineación vertical derecha: la segunda debajo de la primera, y justificadas ambas a la derecha.

En la figura 3.1 se muestran todas las alineaciones horizontales posibles, mientras que las verticales se muestran en la la figura 3.2.

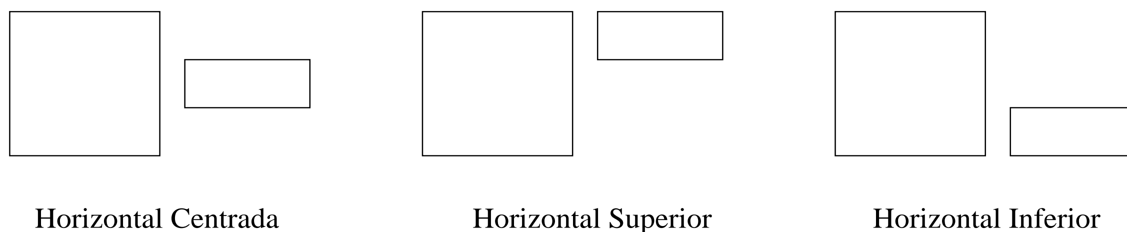

Figura 3.1: Diferentes tipos de alineación horizontal

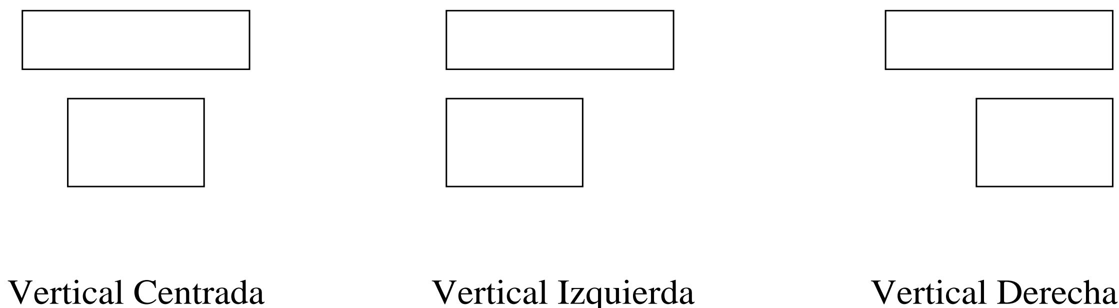

Figura 3.2: Diferentes tipos de alineación vertical

Hay que insistir en el punto de que la alineación especifica cómo se colocan entre sí dos cajas, sin importar el resto. Para decidir posiciones absolutas en la caja madre, se determina que la primera caja de la lista tendrá siempre su origen en la esquina superior izquierda de la caja madre.

Para verlo más claro, pongamos un ejemplo práctico: estamos representando código escrito en lenguaje *modula-2* y se pretende indentar un bloque de código B dos espacios a la derecha del indicador de comienzo del bloque (`BEGIN`). Es decir, debe quedar como se muestra en la figura 3.3.

Las alineaciones descritas no permiten directamente colocar en la posición indicada relativa a la caja "BEGIN", la caja con las sentencias. Hay que insertar una caja con dos espacios en blanco, alineada con la caja "BEGIN" en alineación vertical a la izquierda, y luego la caja de sentencias ponerla alineada horizontal superior con la caja con blancos, quedando las cajas como se indica en la figura 3.4.

```
BEGIN
  sentencia;
  sentencia;
  ...
```

Figura 3.3: Ejemplo de alineación

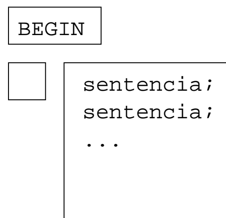

Figura 3.4: Alineaciones de las cajas

Formato alineación lista

Este formato genera un conjunto de cajas alineadas entre sí mediante códigos de alineacion similares a los anteriores. A diferencia del anterior, el formato a aplicar es igual para todas las cajas, y se genera una caja por componente del objeto a representar (por lo que éste debe ser compuesto, si no lo es generará una caja terminal vacía). Además se incluye opcionalmente un separador, con lo que se incluirá también un segundo código de alineación: el primero entre una caja y el separador, y el segundo entre el separador y la siguiente caja.

Si el separador es vacío, no se usará tampoco el segundo código de alineación.

Si el objeto compuesto está vacío, o es simple, se generará una caja simple vacía (de dimensiones 0).

Insistamos en esta diferencia: el formato alineación tupla trata objetos de cualquier clase (ya que aplica cada subformato al mismo objeto) mientras que el formato de alineación lista aplica N veces un formato a cada componente de

un objeto con N componentes.

Formato por defecto y formato universal

Además de los anteriores, es posible solicitar al sistema una visualización sin especificar formato (especificando como formato un objeto nulo). En este caso, buscará en el directorio de formatos un formato general para los objetos de ese tipo (codificado con el mismo nombre del tipo) y si no existe, aplicará un formato universal que deberá definirse obligatoriamente y en principio se utilizará para representar los objetos de la manera que se hace con la utilidad de importación y exportación en ficheros de texto, ya mencionada en el capítulo 2. Este formato lo codificaremos en el directorio de formatos con el nombre “`universal`”.

Esquemas de formatos

Especificados los formatos que hay, podemos pasar a definir los esquemas con los que se construyen en la Base de Datos. Dichos esquemas se encuentran en la figura 3.5, representados mediante la notación utilizada en Collado [3] para representación simbólica de esquemas de la Base de Datos.

3.2.2 Vistas

La **Vista** es el objeto que va a manejar la Herramienta LOPE que necesite generar una representación visual de algo. Una *vista* se define como una presentación visible del contenido de un objeto en un soporte físico, normalmente la pantalla, o una ventana de la misma.

Una vista es también una asociación entre un objeto a representar, y el formato utilizado. Como consecuencia de ambos se genera una *jerarquía de cajas*, siendo la caja más externa o *caja raíz* la que da lugar a la representación de la vista.

```
formato ::= | conversion | formato_alternativa | alineacion_tupla |
  alineacion_lista | composicion | formato_nombrado ;

conversion ::= .text. ;
formato_nombrado ::= $ ;
formato_alternativa ::= + atributo + % + formato ;
alineacion_tupla  ::=  + alineaciones + formatos ;
alineacion_lista ::=  + formato + alineacion + separador +
  alineacion2 ;
composicion ::= + selectores + formato ;
formatos ::= * formato ;
alineaciones ::= * alineacion ;
selectores ::= * selector ;
atributo ::= $ ;
alineacion ::= $ ;
alineacion2 ::= $ ;
separador ::= .text. ;
selector ::= $ ;
```

Figura 3.5: Esquemas de Formatos

La vista tiene un tamaño, y éste se corresponde con el tamaño final de la caja raíz que representa, quien a su vez viene determinado exclusivamente, por el formato utilizado y el objeto representado.

Para su representación, la vista utiliza un *panel de texto* del Subsistema de Terminal Virtual. Ese panel de texto podrá tener ciertas propiedades (como tamaño físico) de modo que la caja representada no quepa en él: para eso el Terminal Virtual proporciona las correspondientes *barras de desplazamiento* o *scroll*.

No obstante, en el momento de crear la vista se generará un panel de texto con un tamaño virtual total especificado por el usuario y un tamaño físico (parte visible en pantalla) opcional. Si no se indica, se considera que se quiere minimizar dimensiones, con lo que el panel generado tendrá un tamaño físico igual a las dimensiones de la caja raíz generada.

3.2.3 Cajas

Hemos nombrado continuamente las cajas, como elementos que representan físicamente los objetos utilizando los formatos.

Este es el momento de recordar que una caja es un elemento visible rectangular. Por lo tanto, incluye entre sus propiedades, su tamaño (ancho y alto) y su posición relativa en la caja madre contenedora.

Una caja puede ser, como hemos indicado, de dos tipos:

- **Terminal**: Este tipo de cajas contienen directamente texto en una línea, por lo que su alto será **1** y su ancho vendrá determinado por los caracteres del texto que contiene. También pueden definirse vacías, con lo que su alto y su ancho serán **0**.

- **Compuesta**: Estas cajas no contienen directamente texto, sino un determinado número de cajas hijas, alineadas entre sí según determine el

formato aplicado. Su tamaño vendrá determinado por los tamaños y posiciones de las cajas hijas que contenga.

Para comodidad del usuario, la caja incluye también entre sus atributos, referencias al objeto que representa, y a la vista a la que pertenece. Posteriormente, en el capítulo 5, dedicado a detalles de implementación, se expondrán todos los atributos y su significado exacto.

3.3 Líneas de ampliación

Ya mencionábamos al principio el hecho de que dejáramos para futuros desarrollos ciertos aspectos no cubiertos en el presente trabajo. Aquí daremos algunas ideas al respecto, en particular en lo referente a las alineaciones condicionales y los formatos gráficos.

3.3.1 Alineaciones condicionales

Hemos visto hasta ahora distintos modos de alineación entre dos cajas consecutivas. Estos mecanismos no son todo lo flexibles que quisiéramos. En ocasiones una representación de cajas alineadas podría salirse de los extremos de la ventana física, por ejemplo.

Con el fin de flexibilizar algo más este punto surge la idea de la alineación condicional. Por el momento, se han propuesto dos tipos de alineación condicional, a las que se les ha dado un nombre "gráfico": alineación **T** y alineación **Z**.

Una alineación **Z** es una alineación horizontal mientras quepan las cajas hijas en la fila actual, pasando a la fila siguiente y partiendo del extremo izquierdo en caso de no caber todas las cajas (es decir, se van colocando los elementos hasta llenar una fila y usando filas inferiores en su caso, tal como se suele hacer al imprimir texto cuando todas las palabras no quepan en una única línea).

Alineacion Condicional Tipo-Z
Tipo-Z

Figura 3.6: Ejemplo de alineación condicional Z

Alineacion Condicional Tipo-T
Condicional
Tipo-T

Figura 3.7: Ejemplo de alineación condicional T

Para más claridad, puede verse el ejemplo de la figura 3.6, donde vemos que se quieren alinear horizontalmente las palabras de la frase “alineación condicional tipo-Z”. Dado que la palabra “tipo-Z” ya no cabe, se pasa ya a la línea siguiente.

La alineación **T** sugiere aplicar todas las alineaciones en el mismo sentido: horizontalmente si se puede, y si no todas en vertical. En el ejemplo de la figura 3.7 puede verse que, como las tres palabras de la frase no caben en una misma línea, se decide colocar todas alineadas verticalmente (en este caso, a la izquierda).

Ambos códigos de alineación condicional pueden combinarse con cualquiera de los tipos de alineación vistos, es decir, no se restringe al ejemplo basado en alineaciones horizontales.

En principio, las alineaciones condicionales harán efecto cuando existan problemas con la colocación de las cajas sin salirse de los límites de la caja principal, cuyas dimensiones pueden estar muy limitadas. En caso de que todo el formato aplicado solo tenga una alineación condicional, solo existirán dos formas de colocar las cajas, es decir, dejando las cajas alineadas horizontalmente o en vertical

(suponiendo la alineación **T**, por ejemplo). Si existe más de una, notaremos que existirán diversas soluciones de colocación de las cajas para lograr encajarlas en la ventana principal.

Esto nos sugiere que tal vez debamos asociar a cada alineación algo similar a una "*prioridad*", que nos permita decidir en casos como éste.

3.3.2 Formatos gráficos

Como dijimos al principio, los entornos de desarrollo de software deben contar con herramientas que implementen técnicas gráficas usadas durante el desarrollo (tal es el caso de los diagramas modulares o los DFDs[1]). Por ello, indicábamos que en este subsistema faltaban las primitivas para generar representaciones gráficas.

Actualmente, este punto está aun poco estudiado. La idea sería dotar al Subsistema de Visualización de mecanismos para representar diagramas. Estos diagramas deben representar cualquier objeto y de cualquier forma. Sin embargo, en la práctica habrá que restringirse a un conjunto limitado de tipos de diagrama, los cuales pueden decidirse según las necesidades que se vayan observando.

Estos diagramas básicos se representarían haciendo uso de las primitivas gráficas que nos proporcione el terminal virtual en sus *paneles gráficos*. Dada la independencia de éstos con los paneles de texto, habría que modificar el módulo `VisView` con el fin de generar un panel de texto o gráfico por cada caja terminal, en lugar de agruparlo todo en un solo panel de texto como se hace actualmente.

[1]Diagrama de Flujo de Datos

3.4 Uso del subsistema desde una herramienta LOPE

En estas líneas describiremos cuál es el uso normal que tendrá el Subsistema desde otras capas superiores.

Toda herramienta que desee visualizar algún objeto lo hará mediante la llamada:

```
VisView.Create(objeto, formato, anchovirtual, altovirtual
      anchofisico, altofisico, panel, vista);
```

En esta llamada, se entregan como parámetros los siguientes: `objeto`, el que se desea representar; `formato`, el que se desea utilizar para representar el objeto; `ancho` y `alto virtual`, las dimensiones (virtuales) totales del panel de texto creado; `ancho` y `alto físico`, el que se desea visible físicamente en pantalla. Y devuelve sobre las variables `vista` y `panel`, respectivamente, el identificador de vista utilizado, y el panel de texto creado al efecto.

A continuación, cualquier modificación que se realice sobre el objeto representado será reflejada automáticamente en la visualización creada.

Al crear la vista, toda la representación se realiza en el modo `Normal` de representación[2]. Es posible alterar posteriormente este modo para, por ejemplo, destacar una parte de lo visualizado, usando la función:

```
VisBox.Display(caja, modo)
```

[2]Los modos de representación son los previstos en Durbán [7] y en Collado [4], así como en el trabajo que actualmente se realiza para implementar el nuevo Terminal Virtual

donde el argumento "`caja`" es la caja que se desea destacar; y el argumento "`modo`" es el nuevo modo deseado. Se puede comprobar que no es inmediato deducir la caja que representa a un determinado objeto, y que la propia vista solo nos da información acerca de la caja raíz. Sin embargo, será posible destacar la caja que representa a un objeto de la siguiente manera:

1. Usar la función `Object.GetBoxList` para obtener del objeto deseado, la lista de cajas que lo representan

2. Recorrer la lista para buscar aquella caja que pertenece a la vista deseada

3. Llamar a la función `Display` para destacar esa caja

Es fácil necesitar en la Herramienta LOPE un editor de lo que estamos viendo en pantalla. Para ilustrar el funcionamiento de éste, supongamos que tenemos representado en una ventana un objeto, al cual accede el usuario con el ratón y modifica algo de las coordenadas `(X,Y)`. Al hacer una edición, el terminal virtual generará un aviso al término de ésta. La herramienta solo tiene que leer las coordenadas del evento, que serán las pertenecientes al lugar editado, y localizar el objeto afectado mediante la función `GetBoxAtXY` y accediendo a las propiedades de la caja devuelta para esa posición.

Al modificar el objeto, se redibujará la pantalla automáticamente (cosa que el usuario no apreciará salvo que se recoloquen las cajas cercanas con motivo de los cambios), y se logrará el objetivo: mantener coherencia entre lo que se ve, la representación interna de lo que se ve, y el contenido de la base de datos.

Finalmente, al terminar la sesión deben destruirse las vistas utilizadas, para lo que se hará uso de la función:

```
VisView.Delete(vista)
```

Esta función destruye cualquier memoria reservada (con lo que destruye la jerarquía de cajas generadas así como el propio objeto vista). Además, destruye el panel de texto utilizado.

En el código de `VisView.def` (véase el capítulo 7) se pueden observar más funciones, si bien desde una Herramienta LOPE no serán en general necesarias. Las funciones de refresco, por ejemplo, están pensadas para un uso exclusivo del Subsistema de Base de Datos.

En lo que respecta a las funciones exportadas por `VisBox.def`, están pensadas todas ellas para ser utilizadas desde las funciones de tratamiento de vistas, y tan solo será útil con frecuencia en las capas superiores, la mencionada función `Display`.

Capítulo 4

Ejemplos de Esquemas de Formatos

En esta sección desarrollaremos sencillos ejemplos de formatos. Para ilustrarlos, se utiliza la representación de datos creada para la utilidad `ObjXport` (ya mencionada en la sección 2.2.4 y también descrita en Collado [3]). Además, se usa otra notación más intuitiva, llamada *lenguaje* LOPE. Dicho lenguaje aparece descrito en Collado [5].

4.1 Bloques de código

El primer ejemplo mostrará un caso sencillo para representar en pantalla bloques de código escritos en *Pascal*. Veamos en primer lugar el planteamiento:

Recordemos que en el lenguaje *Pascal* un bloque es una secuencia de sentencias, encabezada por la palabra `begin`, y finalizado por `end`.

Entonces, un bloque de tres sentencias deberá representarse como se indica en la figura 4.1.

```
begin
  sentencia1 ;
  sentencia2 ;
  sentencia3 ;
end;
```

Figura 4.1: Un bloque de lenguaje *Pascal*

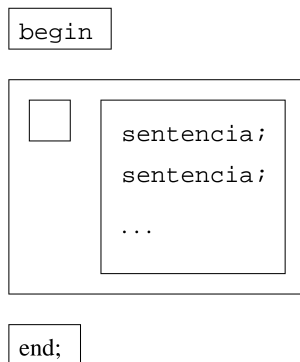

Figura 4.2: Cajas para bloques en lenguaje *Pascal*

Para poder lograr la colocación deseada de cada objeto, teniendo en cuenta las limitaciones que los formatos previstos nos imponen, tendremos que organizar una jerarquía de cajas como la de la figura 4.2.

Ya que el número de sentencias no es conocido de antemano, lo mejor es organizar la caja que contiene las sentencias con un formato de **alineación lista**: cada sentencia tendrá su propia caja terminal, y se generarán tantas cajas como sentencias tenga el bloque.

Si suponemos que el bloque se define como un objeto de iteración de elementos con texto (es decir, su esquema es similar al de la figura 4.3), el formato será el que se muestra en la figura 4.4.

La figura anterior trata de sugerir simbólicamente cómo será el formato, y

```
bloque    ::= * sentencia ;

sentencia ::= .text. ;
```

Figura 4.3: Esquema para bloques de código

como se ve alinea verticalmente a la derecha tres cajas: dos de ellas contienen respectivamente, las palabras *begin* y *end*; y la central es compuesta, y en alineación superior contiene la caja con los espacios para indentación y la caja compuesta con las sentencias (para lo que usamos formato alineación lista).

Nótese que para presentar cada sentencia no es necesario hacer ninguna selección, sino referirnos al valor del objeto directamente; ya que el formato alineación lista recibe directamente el objeto bloque, que ya es una lista de sentencias, y es el formato lista quien selecciona cada componente de manera automática. También puede verse que el formato lista tiene uno, y no dos, códigos de alineación. Se debe a que no se declara ningún separador.

Realmente, para especificar cuál sería el contenido de la Base de Datos al introducir el formato, lo haríamos mediante la notación de la utilidad `ObjXport`. Según esa notación, el formato será el siguiente.

```
format (%) [%

  formato_bloque (formato) [|
    (alineacion_tupla) [+
      (alineaciones) [*
         (alineacion) |<
         (alineacion) |<
      ]
```

```
      (formatos) [*
         (formato) [|
            (formato_nombrado) formato_begin
         ]
         (formato) [|
            (formato_nombrado) formato_sentencias
         ]
         (formato) [|
            (formato_nombrado) formato_end
         ]
      ]
    ]
  ]

  formato_begin (formato) [|
    (conversion) "begin"
  ]

  formato_end (formato) [|
    (conversion) "end"
  ]

  formato_sentencias (formato) [|
    (alineacion_tupla) [+
      (alineaciones) [*
        (alineacion) -^
      ]
      (formatos) [*
        (formato) [|
```

```
            (formato_nombrado) indentador
          ]
          (formato) [|
            (formato_nombrado) formato_sent2
          ]
        ]
      ]
    ]

    indentador (formato) [|
      (conversion) "  "
    ]

    formato_sent2 [|
      (alineacion_lista) [+
        (formato) [|
          (conversion) "%v"
        ]
        (alineacion) |<
        (separador) ""
        (alineacion2) null
      ]
    ]
]
```

Nótese que ahora hemos tenido que incluir el separador y la segunda alineación en el formato lista, aunque lo hemos hecho con objetos de contenido nulo.

```
FORMAT formato_bloque =
 ALIGN
   formato_begin |<
   formato_sentencias |<
   formato_end
END;

FORMAT formato_begin =
 "begin"
END;

FORMAT formato_end =
 "end"
END;

FORMAT formato_sentencias =
 ALIGN
  indentador -^
  formato_sent2
END;

FORMAT formato_sent2 =
 LIST
  ALIGN
    "%v"
    |<
  END;
END;
```

Figura 4.4: Formato para representar bloques de código *Pascal*

```
Fecha ::= + dia + mes + an˜o ;
```

Figura 4.5: Esquema para objetos fecha

```
FORMAT formato_fecha =
 ALIGN
  dia: "Madrid, a %v de " --
  mes: formato_mes --
  an˜o: " de %v"
END;
```

Figura 4.6: El formato Fecha

4.2 El formato Fecha

Para ilustrar este trabajo un poco más, y también para mostrar la utilidad del formato alternativa, vamos a desarrollar otro ejemplo de formato: se trata del formato Fecha.

La idea es, dada una fecha cuya estructura en la base de datos será de una terna de tres números (día, mes y año), representar la fecha con el formato conocido: "Madrid, a X de YY de ZZZZ".

Nótese que pretendemos que YY no se muestre en cifras, sino con el nombre del mes, para lo que nos será de utilidad el formato alternativa. En primer lugar, el esquema de un objeto fecha será el de la figura 4.5.

Suponiendo que tenemos definido ya el formato "`formato_mes`", la definición del formato fecha será, utilizando alineación horizontal centrada, la que se muestra en la figura 4.6.

Nótese que, como el formato alineación tupla no hace selección de componentes del objeto, por lo que para seleccionar éstas (por ejemplo, el día) será necesario utilizar el formato composición. De ahí que hayamos puesto delante de cada conversión, la componente seleccionada seguida de dos puntos. Por tanto, el formato fecha quedará codificado en la base de datos de la siguiente manera:

```
formato_fecha (formato) [|
 (alineacion_tupla) [+
   (alineaciones) [*
     (alineacion) --
     (alineacion) --
   ]
   (formatos) [*
     (formato) [|
       (composicion) [+
          (selectores) [*
            (selector) dia
          ]
          (formato) [|
            (conversion) "Madrid, a %v de "
          ]
       ]
     ]

     (formato) [|
       (composicion) [+
          (selectores) [*
            (selector) mes
```

```
          ]
          (formato) [|
             (formato_nombrado) formato_mes
          ]
       ]
     ]

     (formato) [|
       (composicion) [+
          (selectores) [*
            (selector) an~o
          ]
          (formato) [|
            (conversion) " de %v"
          ]
       ]
     ]
   ]
  ]
]
```

En cuanto al formato mes, se hará mediante una alternativa, seleccionando el valor del objeto tratado como discriminante. Es decir, simbólicamente puede quedar como indica la figura 4.7.

En la base de datos quedará codificado de la siguiente manera:

```
formato_mes (formato) [|
 (formato_alternativa) [+
```

```
    (atributo) value
    (%) [%
      \1 (formato) [|
           (conversion) "Enero"
         ]
      \2 (formato) [|
           (conversion) "Febrero"
         ]
      \3 (formato) [|
           (conversion) "Marzo"
         ]

      ...

      \11 (formato) [|
            (conversion) "Noviembre"
          ]
      \12 (formato) [|
            (conversion) "Diciembre"
          ]
     ]
     (formato) [|
       (conversion) "Mes invalido"
     ]
  ]
]
```

```
FORMAT formato_mes =

   CASE .value. OF
     1: "Enero"
     2: "Febrero"
     3: "Marzo"
       ...
     11: "Noviembre"
     12: "Diciembre"
   ELSE
        "Mes invalido"

END;
```

Figura 4.7: El formato Mes

4.3 El formato Universal

En el la sección 3.2.1 decíamos que el formato universal (el que se aplica cuando no se especifica un formato al construir una vista; y además no existe un formato por defecto para el esquema del objeto a representar) debía ser similar a la salida del ya mencionado programa `ObjXport`. Aquí mostraremos ese formato universal con esas características.

Repasemos en principio cómo representa los objetos el programa ObjXPort. En primer lugar, consideremos los objetos divididos en dos grupos: simples y compuestos. Según esta división, caben las representaciones:

1. **Simples**: El objeto se representa con:

   ```
   (tipo) valor
   ```

   Por ejemplo, un entero se representará con:

   ```
   (integer) 5
   ```

2. **Compuestos**: El objeto se representa así:

   ```
   (tipo) [X
             (tipo1) valor1
             (tipo2) valor2
             ...
          ]
   ```

```
FORMAT universal =
  ALIGN
    utype -^ uvalue
END;
```

Figura 4.8: El Formato Universal

donde la X es el signo que indica el tipo de objeto compuesto (en una tupla, por ejemplo, X = "+").

Nos fijamos entonces que todo objeto, sea simple o compuesto, se puede representar de la forma:

(*tipo*) *valor*

Donde ya representaremos el *valor* de la forma que nos convenga.

Definamos entonces formatos para representar el tipo (llamémosle `utype`) y el valor (sea éste `uvalue`). El formato quedará como indica la figura 4.8.

En cuanto al formato para representar el tipo, será ya una conversión que encerrará el tipo (esquema) entre paréntesis y añadirá a continuación un espacio en blanco para separarlo del valor. Ver la figura 4.9.

El formato para representar el valor tiene que ser capaz de discernir entre los distintos tipos de valor. Lo que haremos será, en caso de ser tipo entero o un símbolo, representar su valor directamente. Si es de tipo texto o referencia, llamaremos al formato correspondiente (que genere, respectivamente, texto encerrado entre comillas o número precedido por el símbolo "^"). Y para cada tipo compuesto (lista, tupla, etc) tendremos el correspondiente formato.

```
FORMAT utype =
  "(%s) "
END;
```

Figura 4.9: El Formato *utype*

```
FORMAT uvalue =
  CASE ValueType OF
    ^          : uref
    .text.     : utext
    .integer.  : "%v"
    $          : "%v"
    +          : utuple
    *          : ulist
    |          : ualt
    %          : udir
  END;
```

Figura 4.10: El Formato *uvalue*

```
FORMAT uref =
  "^%v"
END;

FORMAT utext =
  "\"%v\""
END;
```

Figura 4.11: Los Formatos *uref* y *utext*

Por lo tanto, el formato `uvalue` queda representado como en la figura 4.10.

En cuanto a los formatos `uref` y `utext` (notar para este último que se tienen que preceder las comillas de un carácter de escape, “\”), quedarán como dice la figura 4.11.

Nos quedan los tipos compuestos. Todos ellos son similares: hay que recurrir a la indentación, para obtener un resultado similar a éste:

```
[X
  ...
  ...
]
```

Para ello la indentación a la derecha de los símbolos `[` y `]` utilizamos un formato similar al del bloque de lenguaje *Pascal* (recordar la figura 4.2). En consecuencia, nuestro formato será el que se ve en 4.12.

Véase que alineamos verticalmente a la derecha tres cajas: dos con los mencionados símbolos, y una tercera, a su vez alineación, con los caracteres de

```
FORMAT utuple =
  ALIGN
    '[+' |<

    ALIGN
     '  ' -^ ulistcomp
    END; |<

    ']'
  END;
```

Figura 4.12: El formato *utuple*

indentación (dos espacios en blanco) alineados superiormente con la lista de componentes.

Habrá que definir formatos iguales al anterior, pero cambiando el signo `+` por el que corresponda, para representar listas, alternativas y directorios. El formato `ulistcomp` es el que lista las componentes. En el caso del directorio en lugar de llamar a `ulistcomp` llamaremos a otro, `ulistdircomp` porque hay que incluir las etiquetas en alineación superior con la componente.

Ambos formatos son alineaciones verticales a la izquierda que llaman por cada componente al formato universal. Veamos entonces estos formatos, en la figura 4.13.

Ya tenemos totalmente definido el formato universal. A continuación se incluye el mismo, representado con la salida de `ObjXPort` que será muy similar a la salida que ofrezca este mismo formato cuando esté funcionando:

```
universal (formato) [|
   (alineacion_tupla) [+
      (alineaciones) [*
         (alineacion) -^
      ]
      (formatos) [*
         (formato) [|
            (formato_nombrado) utype
         ]
         (formato) [|
            (formato_nombrado) uvalue
         ]
      ]
   ]
]

utype (formato) [|
   (conversion) "(%s)\ "
]

uvalue (formato) [|
   (formato_alternativa) [+
      (atributo) type
      (%) [%
         \+ (formato) [|
            (formato_nombrado) utuple
         ]
         \* (formato) [|
            (formato_nombrado) ulist
```

```
        ]
        \| (formato) [|
           (formato_nombrado) ualt
        ]
        \^ (formato) [|
           (formato_nombrado) uref
        ]
        .text. (formato) [|
           (formato_nombrado) utext
        ]
        .integer. (formato) [|
           (conversion) "%v"
        ]
            \$ (formato) [|
           (conversion) "%v"
        ]
        \% (formato) [|
           (formato_nombrado) udir
        ]
     ]
     (formato) [|
        (conversion) "%v"
     ]
  ]
]

utuple (formato) [|
   (alineacion_tupla) [+
      (alineaciones) [*
```

```
            (alineacion) |<
            (alineacion) |<
         ]
         (formatos) [*
            (formato) [|
               (conversion) "\[+"
            ]
            (formato) [|
               (alineacion_tupla) [+
                  (alineaciones) [*
                     (alineacion) -^
                  ]
                  (formatos) [*
                     (formato) [|
                        (conversion) "\ \ "
                     ]
                     (formato) [|
                        (formato_nombrado) ulistcomp
                     ]
                  ]
               ]
            ]
            (formato) [|
               (conversion) "\]"
            ]
         ]
      ]
   ]
```

```
ualt (formato) [|
   (alineacion_tupla) [+
      (alineaciones) [*
         (alineacion) |<
         (alineacion) |<
      ]
      (formatos) [*
         (formato) [|
            (conversion) "\[|"
         ]
         (formato) [|
            (alineacion_tupla) [+
               (alineaciones) [*
                  (alineacion) -^
               ]
               (formatos) [*
                  (formato) [|
                     (conversion) "\ \ "
                  ]
                  (formato) [|
                     (formato_nombrado) ulistcomp
                  ]
               ]
            ]
         ]
         (formato) [|
            (conversion) "\]"
         ]
      ]
```

```
    ]
  ]

  ulist (formato) [|
    (alineacion_tupla) [+
      (alineaciones) [*
        (alineacion) |<
        (alineacion) |<
      ]
      (formatos) [*
        (formato) [|
          (conversion) "\[*"
        ]
        (formato) [|
          (alineacion_tupla) [+
            (alineaciones) [*
              (alineacion) -^
            ]
            (formatos) [*
              (formato) [|
                (conversion) "\ \ "
              ]
              (formato) [|
                (formato_nombrado) ulistcomp
              ]
            ]
          ]
        ]
        (formato) [|
```

```
            (conversion) "\]"
          ]
        ]
      ]
    ]

    ulistcomp (formato) [|
      (alineacion_lista) [+
        (formato) [|
          (formato_nombrado) universal
        ]
        (alineacion) |<
        (separador) "\ "
        (alineacion2) |<
      ]
    ]

    udir (formato) [|
      (alineacion_tupla) [+
        (alineaciones) [*
          (alineacion) |<
          (alineacion) |<
        ]
        (formatos) [*
          (formato) [|
            (conversion) "\[%%"
          ]
          (formato) [|
            (alineacion_tupla) [+
```

```
            (alineaciones) [*
               (alineacion) -^
            ]
            (formatos) [*
               (formato) [|
                  (conversion) "\ \ "
               ]
               (formato) [|
                  (formato_nombrado) ulistdircomp
               ]
            ]
         ]
      ]
      (formato) [|
         (conversion) "\]"
      ]
   ]
]
]

ulistdircomp (formato) [|
   (alineacion_lista) [+
      (formato) [|
         (alineacion_tupla) [+
            (alineaciones) [*
               (alineacion) -^
            ]
            (formatos) [*
               (formato) [|
```

```
                (conversion) "%n\ "
              ]
              (formato) [|
                (formato_nombrado) universal
              ]
            ]
          ]
        ]
        (alineacion) |<
        (separador) "\ "
        (alineacion2) |<
      ]
    ]

    uref (formato) [|
      (conversion) "^%v"
    ]

    utext (formato) [|
      (conversion) "'%v'"
    ]
```

4.4 El formato de Documentos

El formato documento es un ejemplo de utilización simultánea de todos los tipos de formatos, además de mostrar la utilidad de la recursión en el propio formato. El formato está pensado para el tipo de objeto "*DocRequisitos*", ejemplo que puede encontrarse en Arana [1]. Lo hemos modificado ligeramente, en el apar-

```
FORMAT ulistcomp =
  LIST
   ALIGN
     universal |<
     '' |<
   END;
  END;

FORMAT ulistdircomp =
  LIST
   ALIGN
     ALIGN
        '%n' -^ universal
     END; |<
     '' |<
   END;
  END;
```

Figura 4.13: Los Formatos *ulistcomp* y *ulistdircomp*

tado de las secciones, por lo que incluimos en la figura 4.14 los esquemas que definen estos objetos. Hacemos hincapié en este ejemplo ya que es el utilizado en los programas de prueba que más adelante veremos.

Recordemos que un objeto del tipo `DocRequisitos` es una tupla con componentes: Título, Autor, Fecha y Secciones. Queremos que la representación de un documento tenga el aspecto de la figura 4.15.

Nótese que además de la indentación queremos la numeración de las secciones (ésta no existe en el objeto documento). Veremos que en efecto podemos definir un formato para hacer esto.

Comenzamos, pues. De la figura anterior puede deducirse que el documento para nosotros va a ser una alineación vertical alineada a la izquierda de los componentes: Título, Autor, Fecha y Secciones. Por lo tanto, el formato quedará como en la figura 4.16. Véase que tenemos que seleccionar las cuatro componentes, ya que la alineación tupla no lo hace. Además hemos incluido separadores para lograr la presentación final deseada.

En cuanto al formato para representar el título y el autor, ambos son iguales (de hecho podría ser el mismo formato). Dado que ambos objetos son una tupla, con el valor (título y autor respectivamente) como segunda componente, y una lista de referencias como primera componente, y dado que las referencias no las queremos presentar, los formatos Título y Autor seleccionarán la segunda componente y representarán su valor directamente. Véase, por ejemplo, el formato Título en la figura 4.17.

En cuanto al formato fecha, ya lo vimos en la sección 4.2. Aquí la presentación deseada es algo más sencilla (no incluye las palabras "*Madrid, a...*") pero el concepto es el mismo. No obstante, al final mostraremos el formato entero en la notación de *ObjXport*.

Vamos entonces con la representación de las secciones. En principio, habrá de *0* a *N* secciones, lo que supone que nuestro formato es una lista alineada verticalmente a la derecha (ver la figura 4.18). Es en el formato para cada

```
DocRequisitos ::= + Titulo + Autor + Fecha + secciones ;
Titulo ::= Nombre ;
Autor ::= Nombre ;
Fecha ::= + dia + mes + an˜o ;
secciones ::= * Seccion ;
Nombre ::= + Referencias + .text. ;
dia ::= .integer. ;
mes ::= .integer. ;
an˜o ::= .integer. ;
Seccion ::= + Titulo + Cuerpo ;
Cuerpo ::= | Parrafos | secciones ;
Referencias ::= * RefRequisito ;
Parrafos ::= * Parrafo ;
RefRequisito ::= | RefPalabra | RefLinea | RefSeccion | RefParrafo ;
Parrafo ::= * Linea ;
RefPalabra ::= \ˆ Palabra ;
RefLinea ::= \ˆ Linea ;
RefSeccion ::= \ˆ Seccion ;
RefParrafo ::= \ˆ Parrafo ;
Linea ::= * Palabra ;
Palabra ::= .text. ;
```

Figura 4.14: Esquemas para objetos Documento

```
Lope Browser-Editor
Carlos Luque

25 de Junio de 1996

1. Entornos de desarrollo

  1.1 Introduccion

        En este capítulo ...
        'Entorno' se refiere a...

  1.2 Tipos de entornos de desarrollo

    1.2.1 Criterios de clasificación de los entornos de desarrollo

        Los criterios que se tendrán en cuenta ...

    1.2.2 Un Entorno como un marco de integración de herramientas

        Teniendo en cuenta los criterios enunciados
        en el punto anterior ...

        Dos herramientas se consideran integradas cuando ...

 2. El Entorno Lope

 ...
```

Figura 4.15: Una presentación de documento

```
FORMAT FormDoc =
  ALIGN
    Titulo: formTitulo |<
    Autor: formAutor |<
    '' |<
    Fecha: formFecha |<
    '' |<
    secciones: formSecciones
  END;
```

Figura 4.16: El Formato Documento

```
FORMAT formTitulo =
  2: "%v"
END;
```

Figura 4.17: El Formato Título

```
FORMAT formSecciones =
  LIST
    ALIGN
      formSeccion |<
      " " |<
    END;
  END;
```

Figura 4.18: El formato Secciones

sección donde hay que pensar un poco más.

En el ejemplo visto en la figura 4.15, vemos que una sección puede ser principalmente de dos tipos: 1º, una sección simple, cuyo contenido son párrafos. 2º, una sección compuesta, que contiene otras secciones dentro. En todo caso, la sección siempre tiene un número de sección y un título.

Por lo pronto, el formato sección deberá generar una combinación de cajas para lograr la indentación del cuerpo frente al título. Esta combinación de cajas es la que veíamos en la figura 3.4. El título será representado mediante un nuevo formato (que incluye el número de sección) y el cuerpo con otro formato (que seleccionará entre aplicar un formato para los párrafos, o de nuevo el formato para secciones). Vemos, pues, en la figura 4.19 el formato para una sección.

El formato para el cuerpo de una sección será una alternativa que, en función del esquema de la componente que tiene el cuerpo, elegirá de nuevo el formato para lista de secciones, o bien entrará ya en los párrafos. Ver la figura 4.20.

El formato para párrafos será una lista aplicando a cada párrafo el formato para un párrafo, con alineación vertical izquierda. A su vez, el formato para un párrafo es una lista con el mismo tipo de alineación que aplicará a cada componente el formato línea. Y finalmente el formato para una línea aplicará

```
FORMAT formSeccion =
  ALIGN
    Titulo: formTituloSeccion |<
    '  ' -^ Cuerpo: formCuerpoSeccion
  END;
```

Figura 4.19: El formato Sección

```
FORMAT formCuerpoSeccion =
  CASE schema OF
    Parrafos: formParrafos
    secciones: formSecciones
  END;
```

Figura 4.20: El formato para el cuerpo de una sección

```
FORMAT formLinea =
  LIST
    ALIGN
      '%v' --
      ' ' --
    END;
  END;
```

Figura 4.21: El formato línea

```
FORMAT formTituloSeccion =
  ALIGN
    <<: formNumeroSeccion -- ' ' -- formTitulo
  END;
```

Figura 4.22: El formato para el título de una sección

una lista con alineación horizontal centrada, a cada componente el formato palabra, que es directamente una conversión del valor. Dado que todos son muy similares, mostraremos en la figura 4.21 solo el último caso: el formato para líneas.

El formato para titular una sección tiene que mostrar, en alineación horizontal centrada, el número de una sección y el propio título de ésta. Para el título utilizaremos el mismo formato del título del documento. En cuanto a la numeración de la sección, tenemos que seleccionar el objeto padre (la lista de secciones) para conocer el número. Así pues, el formato será el de la figura 4.22.

Vamos finalmente con el número de la sección. Una sección numerada

```
FORMAT formNumeroSeccion =
  ALIGN
    <<.<<: CASE schema OF
                 Cuerpo: <<: formNumeroSeccion
           ELSE
                ' '
           END; --
    '.%*'
  END;
```

Figura 4.23: El formato para numerar secciones

"X.Y.Z" es en realidad la componente Z del cuerpo de la sección que la contiene (para lo que usamos el atributo `index`, codificado con "%*"). Para obtener "Y" tenemos que referirnos al atributo `index` del objeto padre. Para obtener "X" debemos hacer lo mismo de nuevo. Habremos llegado a la raíz cuando el esquema del padre de la sección deje de ser el cuerpo de otra sección.

Además, con las alineaciones elegidas para el número de la propia sección y el resto (posibles números de secciones antecesoras) haremos que el orden de aparición sea, primero el resto y luego el número índice de esta sección. Veamos pues el formato, en la figura 4.23.

Aclaremos un poco más el formato: se podrá observar que aplicamos la comprobación de si el formato es "Cuerpo", al padre del padre y no directamente al padre. Esto es así pues el objeto en el que estamos en este momento no es en el cuerpo de la sección a tratar, sino en el título, que es hija suya.

Luego, en caso de que el esquema sea otro cuerpo de sección, debemos aplicar este mismo formato al padre del cuerpo. En caso de no ser un cuerpo, simplemente generamos un espacio vacío. En este formato se puede ver de nuevo la

utilidad de la recursión, aunque ya la hayamos utilizado al considerar nuevas secciones dentro de una sección.

Para finalizar, a continuación se incluye, en la notación del programa `ObjXport`, el formato documento completo.

```
formDoc (formato) [|
  (alineacion_tupla) [+
    (alineaciones) [*
      (alineacion) |<
      (alineacion) |<
      (alineacion) |<
      (alineacion) |<
      (alineacion) |<
    ]
    (formatos) [*
      (formato) [|
        (composicion) [+
          (selectores) [*
            (selector) Titulo
          ]
          (formato) [|
            (formato_nombrado) formTitulo
          ]
        ]
      ]
      (formato) [|
        (conversion) "\ "
      ]
      (formato) [|
```

```
    (composicion) [+
      (selectores) [*
        (selector) Autor
      ]
      (formato) [|
        (formato_nombrado) formAutor
      ]
    ]
  ]
  (formato) [|
    (composicion) [+
      (selectores) [*
        (selector) Fecha
      ]
      (formato) [|
        (formato_nombrado) formFecha
      ]
    ]
  ]
  (formato) [|
    (conversion) "\ "
  ]
  (formato) [|
    (composicion) [+
      (selectores) [*
        (selector) secciones
      ]
      (formato) [|
        (formato_nombrado) formSecciones
```

```
          ]
        ]
      ]
    ]
  ]
]

formTitulo (formato) [|
  (composicion) [+
    (selectores) [*
      (selector) \2
    ]
    (formato) [|
      (conversion) "%v"
    ]
  ]
]

formAutor (formato) [|
  (composicion) [+
    (selectores) [*
      (selector) \2
    ]
    (formato) [|
      (conversion) "%v"
    ]
  ]
]
```

```
formFecha (formato) [|
  (alineacion_tupla) [+
    (alineaciones) [*
      (alineacion) --
      (alineacion) --
    ]
    (formatos) [*
      (formato) [|
        (composicion) [+
          (selectores) [*
            (selector) dia
          ]
          (formato) [|
            (conversion) "%v\ de\ "
          ]
        ]
      ]
      (formato) [|
        (composicion) [+
          (selectores) [*
            (selector) mes
          ]
          (formato) [|
            (formato_nombrado) formMes
          ]
        ]
      ]
      (formato) [|
        (composicion) [+
```

```
          (selectores) [*
            (selector) an˜o
          ]
          (formato) [|
            (conversion) "\ de\ %v"
          ]
        ]
      ]
    ]
  ]
]

formSecciones (formato) [|
  (alineacion_lista) [+
    (formato) [|
      (formato_nombrado) formSeccion
    ]
    (alineacion) |<
    (separador) "\ "
    (alineacion2) |<
  ]
]

formSeccion (formato) [|
  (alineacion_tupla) [+
    (alineaciones) [*
      (alineacion) |<
      (alineacion) -^
    ]
```

```
      (formatos) [*
        (formato) [|
          (composicion) [+
            (selectores) [*
              (selector) Titulo
            ]
            (formato) [|
              (formato_nombrado) formTituloSeccion
            ]
          ]
        ]
        (formato) [|
          (conversion) "\ \ "
        ]
        (formato) [|
          (composicion) [+
            (selectores) [*
              (selector) Cuerpo
                            (selector) \1
            ]
            (formato) [|
              (formato_nombrado) formCuerpoSeccion
            ]
          ]
        ]
      ]
    ]
  ]
```

```
formCuerpoSeccion (formato) [|
  (formato_alternativa) [+
    (atributo) schema
    (%) [%
      Parrafos (formato) [|
        (formato_nombrado) formParrafos
      ]
      secciones (formato) [|
        (formato_nombrado) formSecciones
      ]
    ]
  ]
]



formParrafos (formato) [|
  (alineacion_lista) [+
    (formato) [|
      (formato_nombrado) formParrafo
    ]
    (alineacion) |<
    (separador) "\ "
    (alineacion2) |<
  ]
]

formParrafo (formato) [|
  (alineacion_lista) [+
    (formato) [|
```

```
      (formato_nombrado) formLinea
    ]
    (alineacion) |<
    (alineacion2) |<
  ]
]

formLinea (formato) [|
  (alineacion_lista) [+
    (formato) [|
      (conversion) "%v"
    ]
    (alineacion) --
    (separador) "\ "
    (alineacion2) --
  ]
]

formTituloSeccion (formato) [|
 (alineacion_tupla) [+
   (alineaciones) [*
     (alineacion) --
     (alineacion) --
   ]
   (formatos) [*
     (formato) [|
       (composicion) [+
         (selectores) [*
           (selector) <<
```

```
          ]
          (formato) [|
            (formato_nombrado) formNumeroSeccion
          ]
        ]
      ]
      (formato) [|
        (conversion) "\ "
      ]
      (formato) [|
        (formato_nombrado) formTitulo
      ]
    ]
  ]
]

 formNumeroSeccion (formato) [|
   (alineacion_tupla) [+
     (alineaciones) [*
       (alineacion) --
     ]
     (formatos) [*
       (formato) [|
         (composicion) [+
           (selectores) [*
             (selector) <<
             (selector) <<
           ]
           (formato) [|
```

```
            (formato_alternativa) [+
              (atributo) schema
              (%) [%
                Cuerpo (formato) [|
                  (composicion) [+
                    (selectores) [*
                      (selector) <<
                    ]
                    (formato) [|
                      (formato_nombrado) formNumeroSeccion
                    ]
                  ]
                ]
              ]
              (formato) [|
                (conversion) "\ "
              ]
            ]
          ]
        ]
      ]
      (formato) [|
        (conversion) "%*\."
      ]
    ]
  ]
]

formMes (formato) [|
```

```
(formato_alternativa) [+
  (atributo) value
  (%) [%
      \1 (formato) [|
          (conversion) "Enero"
        ]
      \2 (formato) [|
          (conversion) "Febrero"
        ]
      \3 (formato) [|
          (conversion) "Marzo"
        ]
      \4 (formato) [|
          (conversion) "Abril"
        ]
      \5 (formato) [|
          (conversion) "Mayo"
        ]
      \6 (formato) [|
          (conversion) "Junio"
        ]
      \7 (formato) [|
          (conversion) "Julio"
        ]
      \8 (formato) [|
          (conversion) "Agosto"
        ]
      \9 (formato) [|
          (conversion) "Septiembre"
```

```
          ]
        \10 (formato) [|
            (conversion) "Octubre"
          ]
        \11 (formato) [|
            (conversion) "Noviembre"
          ]
        \12 (formato) [|
            (conversion) "Diciembre"
          ]
      ]
    (formato) [|
      (conversion) "Mes\ erroneo"
    ]
  ]
]
```

Capítulo 5

Características de la implementación

5.1 Visión general

La implementación del Subsistema de Visualización se ha realizado en cuatro módulos, cuyo diagrama se aprecia en la figura 5.1, aunque además se han aplicado ciertas modificaciones a los módulos realizados en Arana [1] y Lorenzana [10].

Las dependencias que se muestran son a nivel de módulos de definición, ya que existen más dependencias en la implementación. Sin embargo, el diagrama pretende aclarar que la Herramienta Lope importará los tipos de datos y servicios principalmente de `VisView`.

Respecto a los módulos implementados, éstos proporcionan los siguientes servicios:

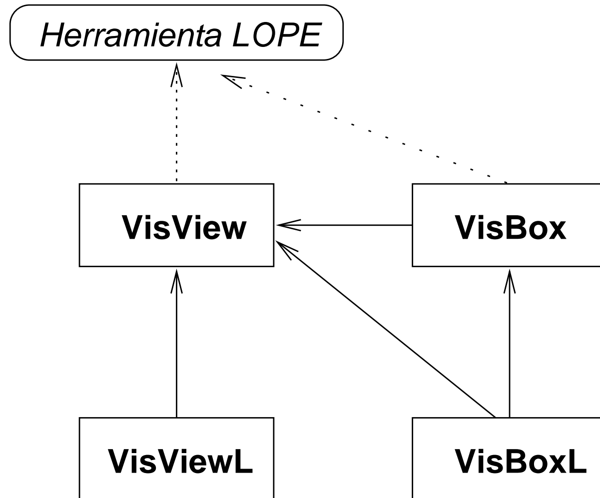

Figura 5.1: Diagrama modular

1. **VisBoxL**: Define el objeto *caja* a bajo nivel, es decir, el tipo `TypeBox` con los atributos de una caja, así como operaciones para crearlas y destruirlas dinámicamente.

2. **VisViewL**: Define el objeto *vista* a bajo nivel, es decir, de modo similar al módulo anterior, define el tipo `TypeView` y las operaciones necesarias para crear y destruir vistas.

3. **VisBox**: Este módulo realiza todo el trabajo relacionado con la generación de jerarquías de cajas. Usa los servicios de `VisBoxL` para formar árboles de cajas a partir del formato y objeto entregados.

4. **VisView**: Este módulo es el que define las operaciones de más alto nivel del Subsistema, y es el que principalmente se va a usar desde una Herramienta Lope, ya que la misma invocará operaciones de `VisView` para generar *vistas* (es decir, combinaciones de formatos, objetos y cajas) y enviarlas a un elemento del terminal virtual.

En realidad, lo que se debe ver desde el exterior es la existencia de las abstracciones de cajas y vistas. La decisión de crear módulos de bajo nivel vino

de la recomendación de encapsular las operaciones del gestor de almacenamiento de objetos del subsistema.

Como apuntábamos al principio, pese a que se pretende que la Herramienta LOPE utilice exclusivamente los servicios del subsistema a través de `VisView`, en el diagrama de módulos anterior puede observarse que también importará servicios de `VisBox`. El motivo es que hay operaciones, como `Display`, que se usan desde la Herramienta y es exportada desde este módulo directamente. Además, la Herramienta LOPE necesitará importar, por ejemplo, el módulo `VisViewL` para poder declarar variables de tipo *vista*.

Las siguientes secciones tratan en profundidad la implementación realizada. En la primera nos centramos en estudiar el mecanismo de almacenamiento elegido para los objetos que maneja internamente el subsistema. Seguidamente se describirán los módulos que gestionan cajas y después los que gestionan las vistas. Para finalizar, concluiremos viendo cómo hemos modificado el código de los trabajos de Arana [1] y Lorenzana [10], para añadirle el soporte necesario para el funcionamiento del Subsistema de Visualización.

5.2 Almacenamiento de objetos

Como sabemos, el subsistema objeto de este trabajo, maneja internamente dos clases de objetos: las cajas y las vistas. En en capítulo 3 comentábamos cómo las cajas se tenían que organizar en jerarquías, y teníamos las vistas como enlace entre el objeto representado, el formato usado, el panel de texto físico, etc. Por lo tanto, cajas y vistas son objetos con una serie de atributos bien definidos, como los siguientes:

- **Dimensiones** de la caja
- **Lista** de cajas hijas
- **Nombre** del objeto representado

- etcétera

La necesidad de los diferentes atributos que finalmente aparecen en la implementación va surgiendo a lo largo de la misma. En cualquier caso, concluimos que necesitamos un mecanismo que nos permita, para cajas y vistas:

- Crearlas y destruirlas
- Referirnos a ella desde cualquier lugar
- Poder acceder a su información
- Poder alterarla

Las distintas operaciones para gestión de objetos caja y vistas han sido encapsuladas en sendos módulos, respectivamente, `VisViewL` y `VisBoxL`. Cada uno de estos módulos exporta funciones para crear y destruir el objeto correspondiente, leer propiedades y asignarlas. Además, en ellos se definen todas las propiedades de cada objeto, y el mecanismo para identificar cada uno: este mecanismo se basa en el uso de un identificador, número entero, cuya declaración no se ha dejado en los módulos que estamos comentando, sino que a su vez se acaban importando del módulo `StoTbl`, tratando así de mantener el estilo en precedentes trabajos del Proyecto Lope.

Internamente, se ha procurado que el acceso a cada caja o vista sea lo más rápido y económico posible. Por ello, el almacenamiento de cada uno de los dos grupos de objetos se realiza mediante un **array de punteros**. El índice del array es el identificador de objeto usado, de tal forma que acceder a una caja con identificador `i` es tan simple como acceder al valor `boxSetPtr^[i]^`, siendo `boxSetPtr` el nombre que le damos al array. El propio array es dinámico, es decir, se crea con llamadas a `LibSys.MemoryGet` en lugar de declarar la variable directamente. De este modo se amplía la flexibilidad de modificación de parámetros como el tamaño del mismo.

5.2.1 Creación y destrucción de objetos

Para asignar y liberar dinámicamente identificadores de cajas (o vistas) se utiliza el mecanismo clásico de mantener una variable interna con el valor del máximo identificador actualmente asignado. De esta manera, cuando haya que reservar un nuevo identificador, en lugar de buscarlo nos iremos al siguiente de ese máximo. Si el máximo ha llegado ya al final del array, buscaremos un identificador libre empezando desde el principio (un identificador se considera libre si la entrada correspondiente del array contiene el puntero nulo).

La creación falla, pues, cuando no haya identificadores libres.

Éste es, pues, el comportamiento de la función `Create` en los módulos de bajo nivel:

1. Incrementar máximo asignado.
2. Si el máximo asignado no ha superado al número de entradas del array, devolver ese máximo como identificador `i`. Además, reservar memoria para la i-ésima posición del array e inicializar todos los atributos.
3. Si el máximo asignado sí superó el número de entradas, buscar un identificador libre y, si se encuentra, operar de forma parecida creando y asignando atributos.
4. Si no se encuentra un identificador libre, fallar.

Hay que decir que la inicialización de atributos tiene lugar en función de la semántica de éstos. Por lo tanto, sus valores iniciales se comprenderán cuando hablemos de los módulos de nivel superior. Por lo que respecta al fallo de la operación, si ésta tiene lugar, además de generar un error interno, devolverá al usuario el identificador nulo.

En cuanto a la destrucción de objetos, operación llamada `Delete` en los módulos de bajo nivel, hace lo siguiente:

1. Fallar si el identificador pasado es no válido: identificador nulo, superior al máximo permitido, o perteneciente a una entrada del array no utilizada.

2. Liberar memoria ocupada por la entrada del array

3. Si el identificador era el máximo actual, decrementar éste.

4. Retornar identificador nulo

5.2.2 Lectura y alteración de atributos

Los dos módulos de bajo nivel, `VisViewL` y `VisBoxL`, exportan las funciones `GetProperties` para leer atributos, y `SetProperties` para alterar los atributos. Ambas funciones comienzan comprobando que el identificador es válido (véase el criterio en la sección anterior). A continuación, las funciones harán su papel de devolver al usuario la información pedida, o modificar el objeto con la proporcionada.

5.2.3 Atributos de los objetos

Vamos a ver rápidamente los atributos considerados para las dos clases de objetos. Se verá su necesidad cuando se vaya describiendo el resto de la implementación.

Cajas

Las cajas tienen los siguientes atributos:

- Dimensiones: posición relativa (x,y) en caja madre, ancho y alto.
- modo: Modo de presentación en pantalla (normal, negrilla…)
- isTerminal: TRUE si la caja es terminal

- text: solo tiene sentido si la caja es terminal, pues es el texto que ésta contiene
- boxList: lista de cajas hijas, solo tiene sentido si la caja es compuesta
- numberOfSons: número de cajas hijas, 0 si es terminal
- index: Posición en lista de la caja madre
- fatherRef: Referencia a la caja madre
- objectRef: Referencia al objeto representado
- viewRef: Referencia a la vista
- formatRef: Referencia al formato
- prevBox, nextBox: punteros para lista de cajas asociadas a un objeto
- sameObjectLink: referencia a sí mismo

Vistas

Las vistas tienen los siguientes atributos:

- formatRef: Referencia al formato
- objectRef: Referencia al objeto representado
- boxId: Referencia a la caja raíz de la jerarquía de cajas
- oldHeight, oldWidth: Antiguos alto y ancho de la caja raíz
- virtualScreen: Referencia a panel de texto virtual
- textPanel: Panel de texto físico

Insistimos en que el significado de muchos atributos quedará claro después, al estudiar el resto de la implementación.

5.3 Estudio de `VisBox`

En esta sección vamos a describir completamente el módulo `VisBox`. Este módulo, como hemos dicho, es el que gestiona las cajas a "alto nivel", es decir, es quien genera y alinea las jerarquías de cajas a partir del formato y el objeto a representar; para lo que utiliza los servicios de bajo nivel proporcionados por la base de datos (para acceder a ésta) y por `VisBoxL` (para manejar el objeto *caja*).

Además, exporta otras operaciones interesantes, que en general afectan a un conjunto de cajas relacionadas entre sí de alguna manera, por lo que no pueden ser exportadas de `VisBoxL`. Dichas operaciones son descritas a continuación.

Posteriormente pasamos a ver un análisis interno, estudiando en primer lugar el mecanismo de generación de jerarquías (viendo cómo van generándose a partir de los distintos tipos de formato).

Luego veremos cómo se hace la alineación, a partir de los códigos especificados en el formato.

Después veremos las operaciones preparadas para soportar la modificación de objetos representados en alguna vista (algoritmo de *realineación*).

5.3.1 Descripción de la interfaz

El módulo `VisBox` exporta las siguientes funciones, tal como puede verse al final, en la sección 7.1.3.

- `Compose`. Esta función es la más extensa de este módulo. Es la que construye una jerarquía de cajas a partir del formato y del objeto representado, en el modo de presentación pasado como argumento. Además necesita como parámetro la vista para establecer algunos atributos de todas las cajas generadas.

- `Align`. Esta función es la que, a partir de los códigos de alineación de una jerarquía de cajas, y los contenidos de las cajas terminales (todo esto habrá sido determinado durante la operación `Compose`), alinea las cajas y establece los atributos no conocidos hasta ahora (tamaño y posición relativa en caja madre).

- `Build`. Esta es la función principal de la interfaz, ya que es la que entrega la jerarquía de cajas alineada. Internamente realiza una llamada a `Compose` y luego a `Align`.

- `Destroy`. Esta función sirve para destruir jerarquías de cajas. Además, elimina cualquier referencia a las cajas eliminadas desde otros lugares, por ejemplo desde los punteros existentes de los objetos en la base de datos.

- `Display`. Esta función se utiliza para cambiar el modo de presentación de una caja y todas las hijas.

- `ChangeText`. Con esta función cambiamos el texto que contiene una caja terminal.

- `InsertNthBox`. Permite insertar una nueva hija de una caja compuesta, en la posición n-sima.

- `DeleteNthBox`. Permite borrar la caja hija n-sima.

- `ReplaceNthBox`. Lógicamente, es una operación `DeleteNthBox` y luego una `InsertNthBox`. Pero internamente ahorra operaciones.

- `ReAlign`. Implementa el algoritmo de *realineación* de cajas. Consiste básicamente en recalcular el tamaño de la caja pasada como argumento (las cajas hijas se suponen ya alineadas) y propaga los cambios hacia arriba hasta llegar a la caja raíz de la jerarquía.

5.3.2 Generación de jerarquías de cajas

Como hemos dicho, esta parte corresponde a la función `Build`. Dicha función, como puede verse en el código fuente, únicamente hace dos cosas: llamar a la función `Compose` (que crea la jerarquía de cajas sin alinearlas) y luego a la función `Align` (que alinea toda la jerarquía).

Composición de la jerarquía

Vamos a describir en detalle cómo se compone la jerarquía de cajas. La función `Compose` es la responsable de ello, aunque ésta lo único que hace es ajustar algunos atributos de la caja raíz de la que partimos (la cual se crea aquí mismo). Estos atributos hay que establecerlos para luego propagarlos a las hijas, como es el caso de `viewRef`. Además, la referencia a la caja madre la ponemos nula, y la referencia al formato la hacemos apuntar al formato deseado.

A continuación se llama a la función `BuildHierarchy` que es quien construye realmente la jerarquía de cajas, siendo por lo tanto una función recursiva. Esta función tiene como argumentos los siguientes:

- object: Objeto a representar
- format: Formato a utilizar
- mode: Modo de presentación
- box: Caja devuelta (aquí no se crea, solo se modifica)
- xPos, yPos: Posición relativa en caja madre

La caja raíz tiene su posición, relativa a la ventana, y se considera por lo tanto en coordenadas *(1,1)*, como puede verse en el código de `Compose`.

`BuildHierarchy` tiene en cuenta que el formato pasado como argumento puede ser nulo, en cuyo caso llama a `ObtainDefaultFormat`, quien a su vez intenta:

1. Obtener un formato etiquetado en el directorio de formatos con el nombre del esquema del objeto a representar (es el formato por defecto propio de cada tipo).

2. Si no se encuentra, obtiene el formato etiquetado con el nombre "*universal*". Este nombre se define en el símbolo `GenericFormatSymbol` y identifica a un formato capaz de representar cualquier objeto. Dicho formato fue descrito en el ejemplo de la sección 4.3.

Para obtener un formato con un determinado nombre se llama a la función `ObtainFormat`, quien busca el formato en el directorio con el nombre "*format*".

A continuación, la función `BuildHierarchy` coge el nombre del formato elegido (ahora será, bien el formato pasado como argumento, bien el que se haya elegido durante el proceso del párrafo anterior). Este nombre, dado que el esquema de formatos se define como alternativa, es la componente única (con índice 1).

Obtenido el tipo del formato, se llama a cada una de las funciones encargadas de tratar cada tipo de formato. Dichas funciones son descritas seguidamente, y todas ellas acabarán, bien estableciendo la caja actual como terminal, bien llamando de nuevo a `BuildHierarchy` después de seleccionar un formato, o bien construyendo una lista de cajas hijas llamando a `BuildHierarchy` para cada una de ellas.

Tratamiento del formato conversión

Para tratar el formato de conversión aparece la función `TreatConvFormat`. Como puede verse en el código fuente, al tenerse como terminal la caja, podemos establecer definitivamente algunos atributos. Los que más nos interesan son el *alto* de la caja (que será 1 ya que la conversión establece siempre una línea de texto) y el *ancho* que será la longitud del texto que contenga la caja. De

estos valores y las alineaciones partirá el procedimiento `Align` para establecer tamaños y posiciones de todas las cajas.

Otra acción que se realiza aquí es relacionar definitivamente el objeto representado con la caja (puesto que recordemos que hay para cada objeto una lista de cajas que lo representan, pensada para localizarlas fácilmente a la hora de propagar una modificación del objeto a las vistas).

Pero lo más importante que aquí se hace es *expandir* o efectuar la conversión. Consiste en recorrer el texto con la conversión y cada vez que aparezca un signo "%" considerarlo como código de control, puesto que ya veíamos en la sección 3.2 que las referencias a atributos del objeto se codificaban con una letra precedida del signo "%".

De ello se encarga la función `Expand`, cuyo algoritmo es el siguiente:

```
j <- 0.

PARA i DESDE 0 AL HIGH(texto) HAZ:

  SEA c <- texto[i].

  SI c <> '%' ENTONCES
     textoexpandido[j] <- texto[i], Incrementar(j).
  SINO
      SEA c' <- texto[i+1].

         SEGUN valor de c' HACER:
           'v': Expandir valor y colocar a partir de
                  textoexpandido[j].
           'n': Expandir etiqueta y colocar a partir de
                  textoexpandido[j].
```

```
            ...

            '%': textoexpandido[j] <- '%'; Incrementar(j);
          FIN

    FIN-SI

  FIN-PARA
```

La variable `j` es el puntero a la posición actual del texto expandido, por lo que crecerá en general, en varias unidades, al expandir un atributo del objeto. La función `CopyStrAtPos` es la encargada de hacer esto: copia la cadena de caracteres solicitada a partir de la posición `j` de la cadena destino, y devuelve `j` modificado de modo que siempre apunte al final de la cadena. Dado que puede terminarse la conversión y la cadena utilizada ser más larga, hay que colocar al final un carácter de fin de cadena con el fin de hacerla compatible con las funciones estándares de tratamiento de cadenas (por ejemplo, las de `LibStr`).

La expansión del valor del objeto es un poco más elaborada: si éste es numérico (tipos entero o referencia) se convierte a cadena usando `Int2Str`. Si es símbolo, se obtiene el nombre del mismo y se trata como valor textual, y si es texto, se copia directamente.

En cuanto a otros atributos, se pasan como texto directamente buscando el nombre que tengan (por ejemplo, esquemas o tipos). Si es numérico (por ejemplo, el índice o número de hijos) se convierten a texto haciendo uso de `Int2Str`.

Si el atributo elegido no existe, se generará un valor nulo adecuado (si el atributo es numérico, un cero; si es alfanumérico, no se generará nada).

Si el código de conversión solicitado es desconocido, no se expande ningún

valor y se genera el error, aunque la conversión puede continuar.

Tratamiento del formato nombrado

Para esta operación tenemos la función `TreatNamedFormat`. Como puede verse en el código fuente, lo único que hay que hacer es obtener el formato nuevo con el nombre que es el valor del formato actual, y aplicarlo al objeto (llamando otra vez a `BuildHierarchy`.

Tratamiento del formato composición

La función `TreatCompFormat` es la encargada de este trabajo. Las operaciones que realiza son, en primer lugar, obtener los selectores, que aparecen como componente del formato con el nombre codificado como `SelectorsSymbol`. El objeto obtenido así es de tipo lista, y contiene la lista de selectores que parten del objeto actual para obtener el nuevo objeto. A continuación, se obtiene el nuevo formato a aplicar, que es la componente del formato composición con el nombre `FormatSymbol`. Y ya con el nuevo objeto y formato seleccionados, se llama de nuevo a `BuildHierarchy`.

La operación `ApplySelectors` se ocupa de obtener el nuevo objeto a partir del actual y la lista de selectores. Dicha función opera de la siguiente manera:

- Para cada selector de la lista de selectores hacer:

  1. Si es el símbolo `FathObjSymbol` (“`<<`”), el próximo objeto será el objeto padre
  2. Si es el símbolo `ThisObjSymbol` (“`^^`”), el próximo objeto será el actual
  3. Si es un valor numérico *N*, el próximo objeto será la componente *N*-sima del actual

4. En otro caso, es un valor texto. Si el objeto actual es directorio, el valor texto se considera como etiqueta seleccionada: por lo tanto, se selecciona la componente con ese nombre como nuevo objeto. Si no es directorio, el selector se considera como nombre de una componente de tupla, y se seleccionará la componente con ese nombre.

Si durante la selección del nuevo objeto se ha producido algún error, será debido a intentar seleccionar un nuevo objeto que no existe. En ese caso se generará un error y pasará al siguiente selector directamente.

Tratamiento del formato alternativa

Para procesar el formato alternativa se ha implementado la función `TreatAltFormat`. Lo primero que hace este código es obtener la primera componente del formato, que según los esquemas vistos en la figura 3.5, contiene el nombre (símbolo) del atributo elegido como discriminante.

A continuación, se compara el valor de ese símbolo con los distintos valores posibles. Si es el valor (`ObjSymb.ValueSymbol`), se actuará de diferente manera según el tipo del valor. Si éste es un número entero, se convertirá a texto, y se hará con él lo mismo que si el tipo del valor fuera texto: se obtiene el símbolo con ese nombre. Nótese que el discriminante se usa como etiqueta en el directorio de alternativas, con lo que cada etiqueta será un nombre contenido en la tabla de símbolos.

Si el discriminante elegido es su etiqueta en un directorio, su esquema o el tipo del valor, se obtendrá el correspondiente símbolo que representa a ese nombre y se tomará como discriminante.

Finalmente, si se elige el número de componentes o la posición, ese valor numérico se convertirá a texto y se operará de nuevo buscando un símbolo con ese nombre.

El símbolo elegido, almacenado en la variable `code` del procedimiento, se

utiliza para elegir la componente del directorio de alternativas, de donde obtenemos el nuevo formato a aplicar. Si no se encuentra la componente con ese nombre, recurriremos a la tercera componente del formato que contendrá el formato por defecto.

Elegido el nuevo formato, llamaremos con éste, y el mismo objeto inicial, a `BuildHierarchy`. Dado que el formato por defecto es opcional, en caso de que no se haya declarado y tampoco se encuentre la alternativa adecuada en el directorio de formatos, se terminará la función devolviendo una caja nula (de dimensiones 0, de tipo terminal, con texto vacío).

Tratamiento del formato alineación tupla

Empezamos aquí con los formatos que generan listas de cajas. Hay que empezar diciendo que, tras el tratamiento de este formato, y del que veremos a continuación, la caja actual ya no se tocará más (será una caja compuesta de N cajas hijas, las cuales se generarán en las siguientes llamadas a `BuildHierarchy`). Por tanto, en esta función, y en la dedicada a la alineación lista, se relacionará la caja actual con el objeto a tratar mediante `InsertBoxInList`, al igual que hicimos hasta ahora solo en el tratamiento del formato de conversión.

De esta manera, cada caja quedará relacionada con el objeto que realmente represente, excluyendo casos como el que la caja actual represente a otro objeto que después ha quedado convertido en otro al aplicarle por ejemplo un formato de composición.

Aclarado este punto, veamos los pasos que realiza en concreto el tratamiento de la alineación tupla.

El formato alineación tupla consta de dos listas: una de códigos de alineación y otra de formatos a aplicar. Así pues lo primero que hacemos es obtener ambas listas.

A continuación creamos la lista de cajas hijas.

Brevemente describiremos la lista de cajas hijas: Esta lista se ha implementado como una sencilla lista doblemente enlazada cuyas operaciones se han encapsulado en un conjunto de funciones denominadas `BoxListInsert`, `BoxListInsertBottom` (para insertar por el principio o el final, respectivamente); `BoxListTop`, `BoxListBottom` (para obtener los elementos extremos de la lista); `BoxListDelete` y `BoxListDeleteBottom` (para borrar el elemento superior e inferior de la lista, respectivamente). Se eligió doble enlace dado que se hacía muy recomendable en las funciones públicas `InsertNthBox` y `DeleteNthBox` para evitar algunos recorridos completos de la lista.

Siguiendo con la función que estamos tratando, al crear la lista de cajas hijas también ajustamos definitivamente algún atributo de la caja madre, como el valor de `isTerminal` (que ahora será `FALSE`) o el número de cajas hijas (que será coincidente con el número de formatos de la lista).

El bucle principal, que hace una iteración para cada formato de la lista, hace lo siguiente: en primer lugar, crea una caja nueva y le prepara algunos atributos que ya son definitivos: su posición en la caja madre, así como algunas referencias: a la caja madre, a la vista y al formato. Esta última referencia apuntará a la componente I-ésima de la lista de formatos.

Cada caja hija se acompañará de un código de alineación con la siguiente caja, en la lista de cajas hijas. Por lo tanto, todas las cajas, excepto la última, tendrán código de alineación definido en la lista de cajas hijas. Por eso, leeremos el código de alineación I-ésimo siempre que no estemos ya con la última caja de la lista.

Establecida toda esta información, se llama a `BuildHierarchy` con la caja hija, el nuevo formato pero el mismo objeto que antes, como argumentos, continuando así la construcción de la jerarquía.

Al final del bucle se ajustarán definitivamente (`SetProperties`) las propiedades de la caja madre. Se deja para el final para tener correcto el atributo con la lista de cajas hijas.

Tratamiento del formato alineación lista

Lo primero que realiza la función encargada de esta tarea, `TreatIteFormat`, es generar una caja vacía (es decir, terminal con dimensiones 0 y sin texto), en caso de que el número de componentes del objeto a representar sea cero, o éste sea simple.

En otro caso, obtendremos la componente con el formato a aplicar en cada caja hija, así como el separador que es opcional y el primer código de alineación.

Seguidamente, y como hacíamos en la función anterior, establecemos para la caja madre el atributo `isTerminal` a un valor `FALSE`. Otro atributo a decidir es el número de cajas hijas. Éste será, para *N* componentes del objeto, también *N* si no hay separador o bien *2*N - 1* si sí lo hay; ya que de existir separador, habrá *N-1* cajas con separador intercaladas entre las *N* cajas del propio formato iteración lista.

Puede verse en el código que en el momento de decidir lo anterior, también se aprovecha para obtener el segundo código de alineación en el caso de que exista separador.

Antes de entrar en el bucle de las cajas hijas, se procede a relacionar la caja madre con el objeto compuesto, como hacíamos en la función anterior o en el tratamiento del formato de conversión.

El bucle de cajas hijas tiene *N* iteraciones. Esto quiere decir que, de existir separador, su tratamiento tendrá lugar dentro del propio bucle. En ese bucle, lo primero que hacemos es crear la caja hija y ajustarle algunos atributos (referencias a la caja madre, a la vista y al formato hijo, así como el índice dentro de la lista de cajas). Luego obtenemos la componente *i*-ésima del objeto aplicado y se llama a `BuildHierarchy` con dicha componente como nuevo objeto, el nuevo formato y la caja hija. Después insertaremos esa caja al final de la lista de cajas hijas, usando el primer código de alineación.

A continuación, se debe colocar la caja hija con el separador, en caso de que

éste exista y que no estemos en el tratamiento de la última componente. Se crea, pues, una nueva caja hija terminal con el texto del separador, altura 1 y anchura igual a la del texto. Luego la insertamos al final de la lista de cajas hijas, usando como código de alineación el segundo.

5.3.3 Alineación de las jerarquías de cajas

Vamos a ver el algoritmo que implementa el procedimiento `AlignHierarchy`. El algoritmo utiliza como información, los tamaños conocidos de las cajas, y los códigos de alineación, que recordemos se encuentran junto a cada caja hija, en las listas de cajas hijas. Es decir, dado un elemento de la lista, el código de alineación que acompaña a la caja establece la colocación relativa entre esa caja y la siguiente de la lista.

El algoritmo basa su funcionamiento en: 1º, calcular las posiciones de las cajas hijas. 2º, conocidas esas posiciones, obtener el tamaño de la caja actual. Por supuesto, la necesidad de recursión consiste en que, para obtener las posiciones de las cajas hijas son necesarios, además de los códigos de alineación, los tamaños de cada una de ellas, que se obtienen llamando al mismo algoritmo por cada caja hija. Es decir, los pasos son los siguientes:

1. Si la caja es terminal, FIN. (Pues toda caja terminal tiene ya calculado su tamaño).
2. Para cada caja hija H, hacer
   - Aplicar algoritmo a H
3. La primera caja hija estará siempre en posiciones *(0,0)*
4. Para cada caja hija H, de la *2* a la N, hacer
   - Calcular posición a partir de la posición de la anterior, el tamaño de ambas y el código de alineación.

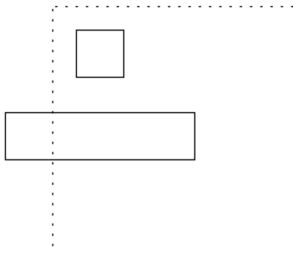

Figura 5.2: Un problema con la alineación

5. Colocadas las hijas, obtener el tamaño de la caja madre

El tamaño de la caja madre se obtiene fácilmente, mirando cuál es la caja hija que más a la derecha termina (será el extremo derecho), es decir, miraremos las sumas *xPos + width* de cada caja; y cuál es la que más abajo llega (el extremo inferior), lo que se obtiene comparando los valores *yPos + height* de todas las cajas.

Sin embargo, este algoritmo no da siempre los resultados correctos. Imaginemos que se desean alinear verticalmente tres cajas, la primera, hemos dicho que caerá en las posiciones *(0,0)*. Si tenemos alineación vertical centrada con la siguiente caja, y ésta es más ancha, la caja quedará colocada con coordenadas negativas (es decir, *se saldrá* de la caja madre), ver la figura 5.2.

La solución a esto es añadir un nuevo paso al algoritmo: al final de calcular las posiciones, se *repasan* para reajustarlas en caso de que alguna caja tenga alguna coordenada negativa. Para implementar esto, lo que se hace es, al calcular las posiciones de las cajas, llevar la cuenta de la menor de las coordenadas obtenidas, y al final reajustarlas todas en caso de que alguna sea negativa (sumando el valor necesario para que la más negativa quede a cero).

Aplicando esto a la figura anterior, las cajas nos quedarán ahora colocadas correctamente, como se muestra en la figura 5.3.

Veamos entonces con más detalle cómo se ha implementado todo esto. Podemos ver en el código fuente que efectivamente solo hacemos algo cuando la caja sea compuesta. En ese caso, lo primero que hacemos es recorrer todas las cajas

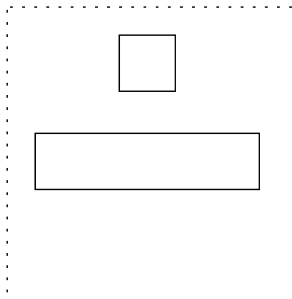

Figura 5.3: Solución al problema de la alineación

hijas, y aplicarles a ellas este algoritmo, es decir, llamamos a `AlignHierarchy` para cada caja hija.

Después, tomamos la primera caja de la lista y la colocamos en la posición relativa a la caja madre *(0,0)*. Para calcular las posiciones de las demás llevamos unas variables, llamadas `prevX`, `prevY`, `prevWidth` y `prevHeight` que contendrán, en el momento de tratar una determinada caja hija, los valores característicos (posición y tamaño) de la caja tratada justo antes.

De esta forma, vemos que por ejemplo, la posición `xPos` de la caja actual con un código de alineación respecto de la anterior de tipo vertical izquierda, será:

```
xPos := prevX
```

ya que tienen que partir de la misma coordenada X. Sin embargo, al colocarse la segunda caja justo *debajo* de la primera, el valor de `yPos` debe ser:

```
yPos := prevY + prevHeight
```

De este modo, al recorrer todas las cajas, tendrán entre sí una colocación relativa correcta. Sin embargo, ya hemos visto que esto no impide que alguna caja quede con coordenadas negativas. Por ello, llevamos el mínimo valor alcanzado en cada `xPos` e `yPos` en las variables respectivas `minX` y `minY`. En el caso de salir alguna negativa, corresponderá a la caja más *salida* de la caja madre,

con lo que el reajuste consistirá en sumar a esa coordenada el valor absoluto del mínimo alcanzado.

Finalmente, para obtener el tamaño de la caja madre, lo que hemos hecho es anotar también los valores máximos alcanzados `maxX` y `maxY` por alguna caja (que también se ajustarán si alguno de los mínimos anteriores fue negativo). Estos valores, al final resultan ser precisamente, los valores del ancho y alto de la caja madre.

Realineación mínima de las jerarquías

Antes de plantear el mecanismo de realineación de cajas, nos introduciremos brevemente en las distintas posibilidades que surgen cuando se modifica un objeto.

En primer lugar, tenemos que decir que, al modificar un objeto, puede tener como resultado la creación, modificación o borrado de una o más cajas. Sin embargo, a la hora de realinear solo nos interesa considerar modificaciones. Una inserción de una caja nueva en una jerarquía existente implica realmente la modificación de una caja compuesta. Un borrado es similar (a menos que se borre la caja raíz, pero para este caso particular, por supuesto, no recolocaremos cajas).

Como hemos visto, cada objeto lleva asociado una lista de cajas que lo representa. Si el objeto es simple y el formato final es conversión, serán cajas terminales. Podrán ser compuestas en otros muchos casos. Veremos después, al ver el módulo `VisView` (sección 5.4) que se actuará, según el tipo de modificación, solicitando la realineación siempre a nivel de la caja *modificada*: si lo sucedido es la modificación de una caja terminal, se realineará al nivel de esa caja. Si es la inserción, borrado o reemplazo, se realineará al nivel de la caja madre.

En ese caso, y si suponemos que la caja hija insertada o reemplazada (en el

segundo caso) se da ya alineada (es decir, se da su tamaño definitivo), realinear será una operación que actuará de esta forma:

1. Colocar las cajas hijas teniendo en cuenta códigos de alineación y tamaños (se suponen ya calculados)

2. Calcular tamaño de esta caja a partir de las posiciones de las hijas

3. Propagar la realineación a la caja madre de la actual

Es decir, lo que hacemos es parecido a lo que hacíamos en la alineación, pero sin recursión a las cajas hijas (es decir, colocamos las hijas pero no llamamos previamente a ninguna función que calcule los tamaños de éstas, pues se suponen ya calculados) Con esto se establece el tamaño de la caja madre y, si ésta no es la raíz, habrá que aplicar el mismo algoritmo a la caja madre de ésta, pues al cambiar el tamaño probablemente habrá que recolocarla junto a sus hermanas y esto supondrá cambios en el tamaño de la caja que lo contiene.

Por supuesto, veremos en `VisView` que si la modificación es, por ejemplo, insertar una caja, llamaremos a `Align` para la caja nueva antes de entregar la caja madre a este algoritmo, pues, insistimos, `ReAlign` solo recoloca las hijas y se llama recursivamente para la caja madre de la pasada como argumento, a menos que ésta sea ya la caja raíz.

Establecidas estas ideas podemos pasar a ver cómo se ha implementado. La función que hace esto la hemos llamado `ReAlign`. Puede verse que, en primer lugar, se decide actuar solo si la caja pasada como argumento no es la raíz. En ese caso, se colocan las hijas de manera similar a como se hacía en `AlignHierarchy`, pero sin previamente pedir a ninguna función que calcule sus tamaños. Después, también de manera idéntica a `AlignHierarchy`, se comprueban posibles coordenadas de las hijas negativas, se corrigen en su caso, y se determina el tamaño de la caja madre.

Finalmente, la llamada recursiva `ReAlign(father)` propaga estas modificaciones "*hacia arriba*".

Hay que hacer un comentario: a la vista de esto, aparentemente no podremos llamar a `ReAlign` a nivel de la caja raíz: en realidad esto nunca tiene lugar ya que, en caso de tener que realinear a ese nivel, lo que se hace es alinear la jerarquía completa (llamar a `Align`) tal como puede verse en los procedimientos de refresco de vistas (ver sección 5.4.3).

Actualización mínima de la pantalla

A la hora de minimizar las operaciones ante la modificación de un objeto, el método descrito en las líneas anteriores es suficiente para minimizar la reconstrucción de jerarquías de cajas. Sin embargo, esto se corresponde con una minimización de una parte del trabajo: la que correspondería a la renovación de la representación lógica interna de la vista.

Además de esto, hay que pensar en mecanismos que minimicen también la otra parte del trabajo: el rellenado del panel de texto. Evidentemente, no nos vamos a limitar a reconstruir el panel de texto con la jerarquía de cajas nuevas, pero minimizar esas operaciones no es un problema trivial. Veremos en detalle la solución adoptada cuando entremos en el módulo `VisView`, en la sección 5.4.4.

5.3.4 Otras funciones de `VisBox`

Como ya indicamos, `VisBox` exporta también otras funciones interesantes desde las capas superiores. Seguidamente las veremos todas.

Procedimiento `InsertNthBox`

Esta función se utilizará durante el refresco de vistas por inserción de componentes (de lo que hablaremos en detalle en la sección 5.4).

El objetivo de esta función es colocar una nueva caja en la posición *N*-sima de una lista de cajas hijas de otra, manteniendo la coherencia de los atributos de todas las cajas (por ejemplo, el atributo `index`).

Lo primero que hace el procedimiento es comprobar la caja madre solicitada es terminal, en cuyo caso no hará nada. En caso de ser caja compuesta, se comprobará que la posición solicitada esté entre *1* y *N+1* siendo *N* el número de componentes actual de la caja madre. Esto es así pues la caja se insertará en la posición *I*-ésima (valor comprendido entre *1* y *N*) pero si se solicita la posición *N+1* se interpretará que se desea insertar la caja al final de la lista.

Hecha esta comprobación, podemos ajustar el atributo `index` de la caja a insertar, pues será exactamente el valor de la posición solicitada.

Si el índice solicitado es *N+1*, se llamará directamente a `BoxListInsertBottom` y no habrá que hacer más (aquí se ve la utilidad de la lista doblemente enlazada: de haberla hecho más simple ahora tendríamos que recorrer la lista entera hasta encontrar el final).

Si el índice solicitado es *1* se insertará con `BoxListInsert` y luego se realizará un bucle para las demás cajas de la lista, con el fin de incrementar el valor del atributo `index` en todas ellas (la antes primera caja ahora es la segunda, la antigua segunda es ahora la tercera, etcétera).

El caso que nos queda es que la posición deseada esté comprendida entre *2* y *N*, ambas inclusive. En ese caso, hay que construir manualmente un nuevo elemento de lista de cajas, recorrer la lista para localizar la caja *i-1*, enganchar los punteros de las cajas de posiciones *i-1* e *i+1* al nuevo elemento de la lista, y finalmente incrementar el valor del atributo `index` de las cajas *i+1* hasta *N*.

Procedimiento `DeleteNthBox`

Este procedimiento empieza también comprobando que la caja madre sea compuesta. Otra comprobación a realizar es que el índice de la caja que se desea

borrar exista (es decir, que esté comprendido entre *1* y *N* con *N* el número de cajas hijas actuales).

A continuación comienza la búsqueda de la caja de la posición solicitada. El tratamiento es diferente si es la primera caja de la lista o no (puesto que los punteros a ajustar de caja siguiente y caja previa, no serán siempre los mismos: en el primer caso hay que ajustar los punteros a caja previa de la que ahora será primera caja, a `NIL`. En el segundo, hay que eliminar un elemento intermedio de la lista, lo que supone *enganchar* las cajas *i-1* e *i+1* entre sí).

Finalmente, hay que ir desde la caja *i+1* hasta la *N* ajustando los valores del atributo `index` (se decrementarán en una unidad todos ellos) y llamar a `Destroy` para la caja borrada, que además de borrar recursivamente cualquier caja hija, eliminará también referencias a estas cajas desde los objetos representados.

Procedimiento `ReplaceNthBox`

Evidentemente podríamos haber sustituido las llamadas a este procedimiento por una llamada a `DeleteNthBox` y otra a `InsertNthBox`. Sin embargo, el reemplazo directo conlleva las ventajas de evitar recorridos de la lista de cajas, cambios del atributo `index` así como el trabajo de liberar y luego reservar memoria de nuevo.

De hecho, el código de esta función resulta muy simple en comparación con las dos anteriores. Simplemente tiene que localizar el elemento *i*-ésimo de la lista, hacer que la caja que representa sea la nueva, y aplicar el procedimiento `Destroy` a la antigua.

Procedimiento `Display`

Este procedimiento permite cambiar el modo de presentación de una caja, y es algo que implica dos cosas:

- Propagar ese cambio a todas las cajas descendientes de la actual

- Enviar el nuevo modo a la zona del panel de texto que ocupa esa caja

Para realizar la primera función se ha construido el procedimiento `BuildDisplay`, quien, dada una caja y un modo de presentación, hace:

1. Cambia el modo de la caja actual
2. Para cada caja hija H, llama a `BuildDisplay(H,modo)`

Para enviar el nuevo modo a la zona del panel de texto que ocupa la caja, hay que obtener, en primer lugar, la posición absoluta de la caja en dicho panel. En principio solo contamos con la posición relativa en su caja madre, pero podemos obtener las coordenadas absolutas ejecutando el algoritmo siguiente, que recibe a su entrada los parámetros por referencia `box`, `x` e `y`:

1. Si la caja madre de `box` es `NullBoxId` (es decir, `box` es la caja raíz), entonces hacer:
   - `x <- box.xPos`
   - `y <- box.yPos`
2. Si la caja `box` no es la raíz hacer:
   - `x <- x + box.xPos`
   - `y <- y + box.yPos`
   - Llamar de nuevo a este algoritmo, con parámetros `box`, `x` e `y`

La idea es obtener las coordenadas absolutas mediante una recursión desde la caja actual hasta la caja raíz. La implementación del algoritmo se realiza en el procedimiento `ObtainAbsXY` y debe recibir como primeros valores de x e y el *0*.

Una vez que conocemos la posición absoluta (`absX`, `absY`) de la caja, cambiaremos el modo de presentación del rectángulo de vértices (`absX`, `absY`) y (`absX + ancho`, `absY + alto`)

Procedimiento `ChangeText`

Este procedimiento cambia el texto de una caja terminal, para lo que trabaja directamente con el atributo `text` solo si la caja es terminal.

Procedimiento `ReplaceAlignCode`

Recordemos que en la lista de cajas hijas de otra, los elementos incluidos son, la propia caja hija y el código de alineación de ésta con la siguiente (aparte de los punteros a los elementos contiguos de la lista).

En este procedimiento lo que hacemos es cambiar el código de alineación del elemento i-ésimo de la lista. Para ello, recorremos ésta hasta alcanzar la componente deseada, y directamente cambiamos el código de alineación.

Procedimiento `Destroy`

Este procedimiento se ejecuta antes de borrar una caja, y trata de eliminar cualquier información dependiente de ésta, como pueden ser cajas hijas, o referencias a éstas o a la propia caja madre desde otros objetos.

Así pues lo primero que hace es recorrer la lista de cajas hijas, si la hay (es decir, si es compuesta) y para cada caja hija llamar recursivamente a este procedimiento; para después eliminar cada elemento de la lista.

Al final, se llama a `DeleteBoxFromList`, que borra cualquier referencia a esta caja desde los objetos y finalmente elimina la propia caja llamando al procedimiento `Delete` de `VisBoxL`.

5.4 Estudio de `VisView`

En las siguientes líneas vamos a describir con cierta profundidad el módulo `VisView`. Este módulo, como decíamos al principio, gestiona las vistas, definiendo las operaciones de más alto nivel del subsistema. Las aplicaciones de

la capa de edición de LOPE utilizarán principalmente la interfaz de `VisView` aunque ya hemos indicado que se ayudarán de otros elementos exportados desde otros módulos. `VisView` también proporciona los servicios principales para mantener la coherencia entre el contenido de la base de datos y la visualización, servicios que son utilizados desde el subsistema de base de datos, de la forma que describiremos después.

Empezaremos repasando la interfaz exportada, para seguidamente estudiar el funcionamiento interno, que se apoya fuertemente en el trabajo que realiza el módulo `VisBox`, visto en la sección anterior.

5.4.1 Descripción de la interfaz

El módulo `VisView` exporta las siguientes funciones, tal como puede verse al final, en la sección 7.1.4.

- `Create`. Esta función construye una vista completamente, es decir, crea el objeto vista y sobre él, y a partir del objeto y el formato pasados como argumento, construye alinea y visualiza la correspondiente jerarquía de cajas, devolviendo también el panel de texto donde se está viendo.

- `Delete`. Este procedimiento realiza la operación complementaria a la anterior: borra una vista y todos los objetos que se crearon para ella: jerarquía de cajas, referencias a las cajas desde los objetos y panel de texto de representación.

- `RefreshViews`. Este procedimiento realiza un redibujado de todas las vistas que representen el objeto pasado como argumento. Como tal, vale para refrescar vistas que representen cualquier clase de objeto. Sin embargo, en caso de que la modificación producida sea la inserción, extracción o reemplazo de una componente del objeto, resultará menos costoso usar uno de los procedimientos especializados siguientes.

- `RefreshViewsByInsertedComponent`. Este procedimiento es en realidad una llamada a `RefreshViews`, salvo que el formato una alineación lista, en cuyo caso minimiza el trabajo realizado.

- `RefreshViewsByDeletedComponent`. Este procedimiento minimiza el trabajo, como el anterior, para el caso de extracción de componentes en objetos compuestos.

- `RefreshViewsByReplacedComponent`. Este procedimiento, similar al anterior, se orienta a modificaciones consistentes en reemplazar componentes en objetos compuestos.

- `GetBoxAtXY`. Este procedimiento permite a las capas superiores obtener la caja de mayor profundidad que ocupa una determinada posición del panel de texto.

- `InsertBoxInList`. Con este procedimiento podemos insertar una caja en la lista de cajas que representan el objeto. Está pensado para uso exclusivo del constructor de jerarquías (`VisBox.Compose`).

- `DeleteBoxFromList`. Este procedimiento elimina una caja de la lista que representa al objeto. También está pensado para uso interno.

5.4.2 Creación y borrado de vistas

Creación

De esto se ocupa el procedimiento `Create`, como ya hemos dicho. Los pasos que realiza son:

1. Crear el objeto *vista* y asignar algunos atributos: referencia al formato y al objeto representados

2. Crear una *pantalla virtual*. Se verá la función de ésta posteriormente; de momento diremos que es un *array* de dimensiones fijas que representa al panel de texto utilizado

3. Crear la jerarquía de cajas asociada, llamando a `Build`

4. Crear el panel de texto y construir sobre él la representación, llamando para ello a la función `BuildScreen`. Además, copia la representación física en la virtual, llamando a `BuildVScreen`. El panel de texto tendrá como dimensiones virtuales, las solicitadas por el usuario; y reales (físicas) las solicitadas si no son 0. En caso de serlo, se establecerán como tales las dimensiones de la caja raíz generada.

El subprograma `BuildScreen` realiza un recorrido por la jerarquía de cajas, enviando al panel de texto el contenido. Opera de la siguiente manera:

1. Si la caja es terminal, pintar en las posiciones absolutas *(absX,absY)* actuales el texto contenido

2. En otro caso, para cada caja hija, hacer:

   - Obtener coordenadas absolutas de la caja hija (por ejemplo, la coordenada X será $xPos + absX$)
   - Llamar a BuildScreen con la caja hija y esas coordenadas

El subprograma `BuildVScreen` hace un recorrido similar al anterior, aunque al llegar a una caja terminal, en lugar de llamar a la función del terminal virtual `TextPWriteStr`, llama a un procedimiento interno `PaintStringatXY`, que hace lo mismo que la anterior pero sobre la *pantalla virtual*.

El procedimiento `PaintStringatXY` no se limita a enviar a la fila Y columna X la cadena deseada, sino que también es capaz de detectar la situacion de haber llegado al margen derecho, para continuar entonces en la línea siguiente empezando por el margen izquierdo. Tratamos así de garantizar que la *pantalla virtual* contenga siempre la misma información que el panel de texto.

Borrado

Para borrar una vista y toda la información asociada, se usa, como hemos visto, el procedimiento `Delete`.

Así pues, lo que hace es simple: en primer lugar, llama al procedimiento `Destroy` de `VisBox` para eliminar las cajas y toda referencia a éstas. A continuación, destruye el panel de texto utilizado así como la *pantalla virtual*. Finalmente, borra el objeto *vista*.

5.4.3 Refresco de vistas

El refresco de vistas se ha planteado, en general, mediante procedimientos a llamar desde los módulos de gestión de objetos, cuando se produzca una modificación de alguno. La interfaz exportada desde `VisView` para estas funciones se simplifica al máximo de modo que las modificaciones de los módulos de los trabajos de Arana [1] y Lorenzana [10] sea mínima.

Así pues, se establece que cuando se modifique un objeto `O`, se haga una llamada a `RefreshViews(O)`. Internamente, el procedimiento refrescará toda vista que contenga cajas que representen a `O`. Pero además, el refresco tratará de minimizar las operaciones, para lo que usará funciones como `ReAlign`, descrita en la sección 5.3. Por este motivo surge la necesidad (luego veremos por qué) de crear nuevas operaciones específicas, para el caso de que el refresco se produzca por inserción, borrado o reemplazo de componentes de objetos compuestos, dejando la operación `RefreshViews` exclusivamente para modificaciones de objetos simples (lo que no significa que la caja que lo representa sea simple, de ahí que la implementación de `RefreshViews` sea algo más complicada de lo previsto, como veremos).

Procedimiento `RefreshViews`

Lo primero que hace esta operación es obtener la lista de cajas que representan al objeto, y para cada una de ellas, hace lo siguiente:

1. Obtiene de la caja, el formato, y otros atributos necesarios, para poder llamar a `Build` y construir una nueva caja que represente a ese objeto
2. Enlaza esa nueva caja a la caja madre (con el atributo `fatherRef`)
3. Establece también otro atributo conocido: el `index`, idéntico al de la caja que va a sustituir
4. Reemplaza la antigua caja con esta nueva caja en la jerarquía correspondiente
5. Alinea la jerarquía
6. Repinta su representación física

Hay que decir que la alineación de la jerarquía no se hace llamando a `Align`, sino a `ReAlign`, pero solo en caso de que la nueva caja no sea raíz. En caso de que sea raíz, lo que hacemos es destruir toda la antigua jerarquía y poner como nueva jerarquía de cajas asociada a la vista, la recién creada. En este caso, la alinearemos mediante `Align`.

Vemos que aunque esta operación se utilice habitualmente sobre objetos simples, en realidad es de uso universal, válido para cualquier tipo de objeto y representación. Lo que sucede es que los casos de inserción, borrado o reemplazo de componentes en objetos compuestos no estarían suficientemente optimizados, ya que la operación `RefreshViews` recibiría como parámetro la caja madre de la insertada, borrada o reemplazada, lo que haría que se construyera y realineara, no solo la caja que nos interesa, sino todas sus "*hermanas*". Por ello se desarrollan operaciones específicas para los casos que hemos indicado. Estas operaciones se comentan en las siguientes líneas.

Procedimiento `RefreshViewsByInsertedComponent`

Este procedimiento, y los tres siguientes, tienen en común el estar preparados para tratar inserciones, borrados y reemplazo de componentes de objetos compuestos, de una forma mejorada frente al procedimiento anterior, más genérico.

Veremos en la implementación que solo se hace un tratamiento especial si el formato que generó la caja compuesta es de tipo alineación tupla. Efectivamente, este es el único caso en el que se genera una caja hija por cada componente del objeto. En los demás casos, al no suceder esto, no hay nada mejorable que no recomiende operar como en `RefreshViews`.

Así pues, lo primero que hace este procedimiento es, como el anterior, obtener la lista de cajas que representan al objeto padre de la componente modificada, y aplicar los siguientes pasos a cada caja de la lista:

1. Obtener el formato que generó la caja
2. Si este formato no es de tipo alineación lista, operar de modo similar a `RefreshViews`
3. Si por el contrario, sí es alineación lista, hacer lo siguiente:
   - Obtener el formato a aplicar a cada componente
   - Con ese formato y la nueva componente insertada en el objeto, construir una nueva caja (llamando a `Build`)
   - Enlazar esta caja a través del atributo `fatherRef` con la que va a ser su caja madre
   - Si hay separador, crear caja con el texto del separador (operación muy similar a la que hacíamos en `TreatIteFormat`)
   - Colocar ambas cajas en su sitio y alinear

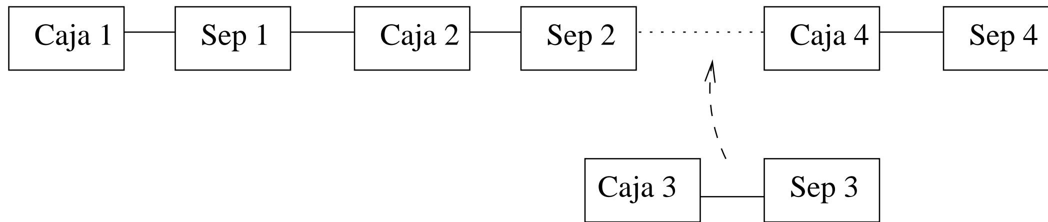

Figura 5.4: Inserción de componente intermedia

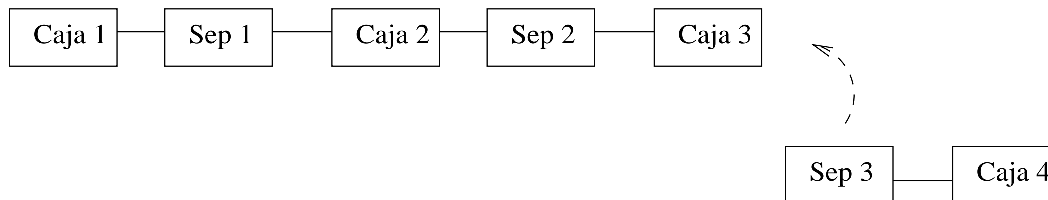

Figura 5.5: Inserción de la última componente

Como antes, hay que hacer algunas matizaciones. En primer lugar, la operación `SubFormatType` permite obtener si el formato contenido es de tipo alineación lista.

Para obtener el formato a aplicar a cada componente también se recurre a un procedimiento auxiliar, `ObtainIteSubFormat`, cuya implementación puede comprenderse directamente si se ha estudiado la sección sobre el módulo `VisBox`.

Finalmente, cuando hay separador hay que tener cierto cuidado: si la componente del objeto insertada no es la última, insertaremos esa caja entre el separador de la caja anterior, y la siguiente. Y por lo tanto el separador deberá insertarse a continuación de la nueva caja (ver figura 5.4).

Pero si la componente fuera la última, no necesitaremos el separador a continuación, pero sí delante de la caja con el contenido, ya que si no, no tendríamos separador entre la antes última caja y la nueva (ver figura 5.5).

También tenemos que decir que, como sucedía en `RefreshViews`, llamaremos a `ReAlign` solo si la caja no es la raíz, pues si la es la alinearemos totalmente con `Align` y reemplazaremos la jerarquía de cajas completa. Obsérvese que a una función u otra la llamamos a nivel de la caja madre de la insertada. En efecto,

la caja recién insertada ya está alineada (al construirla mediante `Build`) y lo que hay que hacer es colocarla junto a sus "*hermanas*", calcular las dimensiones de la caja madre y propagar la alineación.

Por último en el código existe una opción final, para el caso de que el formato no sea de alineación lista, en el que el comportamiento es similar al de `RefreshViews`.

Procedimiento `RefreshViewsByDeletedComponent`

Este procedimiento se aplica, como el anterior, a objetos compuestos (en este caso por borrado de componentes) y solo actua de manera especial cuando el formato es de tipo alineación lista.

Como puede verse en el código fuente, también hay que quitar la caja separadora en caso de que haya código separador. Y hay que actuar de distinta forma, según estemos en la última componente o no, como vimos en el procedimiento anterior.

En los demás aspectos, este procedimiento es muy similar al descrito en los párrafos anteriores.

Procedimiento `RefreshViewsByReplacedComponent`

En este último procedimiento, se actúa de manera similar a los dos anteriores, si bien nos ahorramos tratar con cajas separadoras ya que lógicamente no hay cambios en ninguna de ellas. Puede verse, pues, que lo único que se hace, siempre que tenga sentido (que el formato sea de tipo alineación lista, como siempre) es reemplazar la caja en posición *N* (o *2*N - 1* en caso de haber separadores) con la nueva caja, y luego alinear o realinear la jerarquía completa.

También, como en los dos anteriores procedimientos, se prevé que el formato no sea alineación lista, en cuyo caso se actúa de manera parecida a como lo hace `RefreshViews`.

5.4.4 Refresco del panel de texto

En las funciones vistas para el refresco de vistas, se puede observar que siempre llamamos a un procedimiento `RePaint`, una vez que tenemos la jerarquía de cajas reconstruida y alineada.

Este procedimiento es el que se ocupa de trasladar los resultados obtenidos sobre las jerarquías de cajas, al panel de texto donde se encuentre representada.

En general, nuestra intención es minimizar tanto los cálculos internos de actualización como las operaciones de *entrada/salida* que se tengan que realizar con el panel de texto, es decir, respetar aquellos puntos que no cambien de una situación de la vista a otra.

En esta implementación, dado que se trata de un prototipo donde no tiene excesiva importancia este punto, nos vamos a limitar a minimizar las operaciones de entrada y salida, dejando como líneas de ampliación las ideas para ir más allá en esta minimización.

En definitiva pues, los pasos que seguiremos en el proceso de volcado de la nueva jerarquía de cajas al panel de texto serán:

1. Enviar a la pantalla los caracteres generados por la nueva versión de la vista si son diferentes a los de la anterior versión

2. Limpiar la “*basura*”, es decir, borrar caracteres de la antigua versión que ahora son espacios en blanco

Para ambas operaciones utilizamos la “*pantalla virtual*” que hemos venido nombrando en párrafos anteriores. El procedimiento `RePaint` empieza construyendo una nueva versión de la pantalla virtual (la que previamente rellena de espacios en blanco puesto que el *array* utilizado puede contener *basura* inicialmente) para lo que llama a `BuildVScreen`, función ya comentada. Con esta versión de la pantalla virtual, se compara carácter a carácter con la antigua, y

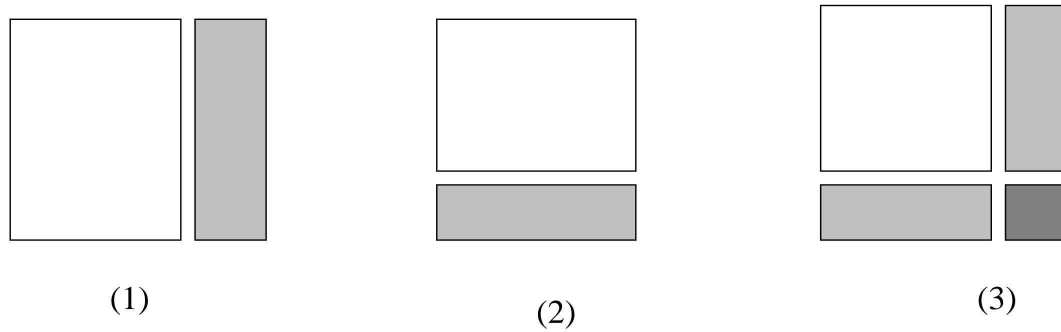

Figura 5.6: Tres casos para eliminar *basura*

se vuelca a la pantalla todo carácter de la nueva versión que sea diferente al de la antigua versión en la misma posición.

Con la operación anterior tendremos correctamente construida la zona del panel de texto que ocupa la nueva caja. Sin embargo, aun puede quedar "*basura*", ya que la antigua caja raíz pudo ser más grande que la nueva. Para limpiar este resto de la antigua caja raíz hacemos tres bucles, correspondientes a tres casos:

1. La nueva caja raíz tiene un ancho menor que la antigua

2. La nueva caja raíz tiene un alto menor que la antigua

3. La nueva caja raíz tiene alto y ancho menor que la antigua

En el primer caso, quedará un espacio de "*basura*" a la derecha de la caja nueva que habrá que limpiar. En el segundo, quedará ese espacio debajo de la caja nueva. En el tercero, ese espacio quedará no solo a la derecha o debajo, sino también en la zona inferior derecha que puede verse en la figura 5.6-(3).

Para programar estos bucles nos resultan de utilidad los atributos `oldHeight` y `oldWidth` que comentábamos unos párrafos antes: con ellos evitamos perder las antiguas dimensiones de la caja raíz antes de que ésta se reconstruyera.

Hecha la operación de limpieza, tendremos ya una pantalla coherente con la nueva pantalla virtual, con lo que destruiremos la antigua y pondremos la

nueva en el lugar que ocupaba ésta.

5.4.5 Otras funciones de `VisView`

Procedimiento `GetBoxAtXY`

Como hemos indicado, con este procedimiento podemos obtener la caja más profunda en una determinada posición. Para ello, hacemos una función recursiva auxiliar que trabaja con la caja y no con la vista que recibe como argumento el procedimiento que nos ocupa.

Este otro procedimiento, llamado `FindXYBox`, funciona de la siguiente forma:

1. Si la caja es terminal, entonces es la que buscamos (terminar)
2. Si no, para cada caja hija, hacer:
   - Si el punto (X, Y) buscado está dentro de la caja, entonces parar el bucle y llamar `FindXYBox` para esta nueva caja.

Comprobar si un punto (X, Y) está dentro de la caja exige conocer sus coordenadas absolutas. En este caso, lo que hemos hecho es modificar (X, Y) para adaptarlo a las coordenadas relativas de la caja. Por otro lado, el problema de conocer si un punto (X, Y) pertenece a una rectángulo es ya un problema geométrico trivial.

Procedimiento `InsertBoxInList`

Este procedimiento, destinado a tratar la lista de cajas asociadas a cada objeto, inserta en la cabecera de la lista la caja pasada como argumento (el objeto lo obtiene de las propiedades de la caja, de modo que ésta debería estar ya correctamente construida).

Es este el momento de explicar un poco cómo se han implementado las listas de cajas asociadas a cada objeto.

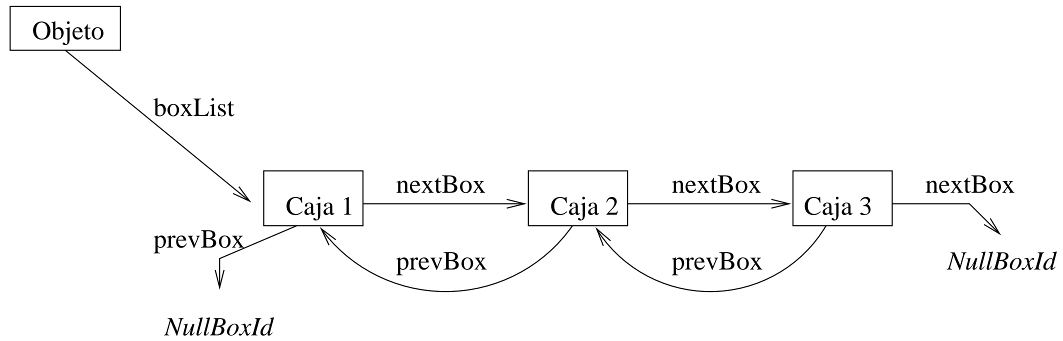

Figura 5.7: Listas de cajas asociadas a un objeto

Se trata de una lista doblemente enlazada, a cuya cabecera accedemos mediante un puntero (en realidad, identificador de caja) que es un nuevo atributo añadido al descriptor de un objeto (ver sección 5.5), llamado `boxList`.

Cada caja, a su vez, tiene dos atributos, también punteros a caja, llamados `nextBox` y `prevBox`. De esta forma podemos recorrer la lista fácilmente, mientras que el hacer la lista doblemente enlazada nos permite mejorar la operación de borrar un elemento, como veremos al tratar el procedimiento siguiente. En la figura 5.7 se muestra un esquema de lo descrito en estas líneas.

Volviendo al procedimiento `InsertBoxInList`, puede verse que resulta muy sencillo. En primer lugar accede a la lista de cajas mediante una función específica exportada desde el módulo `Object` de la que luego hablaremos. Después, y para acabar, hace la inserción directa del nuevo elemento (lo cual consiste simplemente en modificar los punteros de los objetos, no hay que reservar memoria).

Procedimiento `DeleteBoxFromList`

Este procedimiento empieza comprobando que la caja tenga objeto representado (por ejemplo, una caja con un separador no representa ningún objeto). Hecho esto, obtenemos del objeto la lista de cajas, y gracias a que en la caja deseada tenemos punteros a la siguiente y a la anterior, no tenemos que realizar una búsqueda, sino simplemente enganchar la anterior a la siguiente directamente.

Hay que tener cuidado con los casos extremos: si estamos en la última caja de la lista (`nextBox` es `NullBoxId`), solo hay que enganchar la penúltima a `NullBoxId`. Y si es la primera, solo habrá que enganchar el puntero a caja previa de la siguiente, a `NullBoxId`, además de cambiar en las propiedades del objeto, el puntero a la primera caja de la lista, que ahora deberá apuntar a la siguiente.

5.5 Modificaciones a los trabajos de Arana [1] y Lorenzana [10]

Como hemos venido diciendo, para mantener automáticamente la coherencia entre los objetos de la base de datos y las posibles representaciones visuales de éstos, hay que modificar ligeramente el trabajo realizado en Arana [1] y Lorenzana [10]. También hay otras modificaciones necesarias a raíz de la creación de los nuevos tipos de objetos (las cajas y las vistas) así como por la declaración de formatos. Veamos brevemente qué modificaciones se proponen.

1. **Nuevos símbolos**. El establecimiento de nuevos identificadores de esquema estándar en la base de datos nos lleva a definir nuevos símbolos predefinidos, lo que implica modificar el módulo `ObjSymb` del trabajo de Arana [1]. En concreto, se añaden nuevas constantes a `ObjSymb.def`, definiendo nombres y valores, como sigue:

```
(**)
   ValueSymbolName        ="value";
   LabelSymbolName        ="label";
   TypeSymbolName         ="type";

   ... etc ...
```

```
    VerRightSymbolName    ="|>";
    VerLeftSymbolName     ="|<";
    VerCentSymbolName     ="||";

(**)
    ValueSymbol           = -11;
    LabelSymbol           = -12;
    TypeSymbol            = -13;

    ... etc ...

    VerCentSymbol         = -35;
    VerRightSymbol        = -36;
    VerLeftSymbol         = -37;
```

La modificación de `ObjSymb.mod` consiste en iniciar las nuevas entradas del *array* de símbolos predefinidos con los que acabamos de introducir. Es decir, añadir líneas como la siguiente:

```
PredefinedSymbols[ConvFormatSymbol]  := ConvFormatSymbolName;
```

para definir el símbolo predefinido para identificación del formato de conversión. Por supuesto, hay que modificar la constante `MaxPredefinedSymbols` para reflejar los nuevos símbolos predefinidos.

2. **Nuevo atributo para el descriptor de objetos**. El descriptor de objetos se declara en el módulo `StoTbl`, perteneciente al trabajo de Lorenzana [10]. Dicho descriptor define los atributos de cada entrada de la

tabla de objetos. El atributo que añadimos es un identificador de caja al que damos el nombre de `boxList`, y representa el inicio de la lista de cajas que representan al objeto, lista que se forma con los atributos `nextBox` y `prevBox` de las propias cajas.

De este modo el descriptor de objeto queda ahora así:

```
TYPE
  TypeObjDescriptor =
    RECORD
      memAddress  : ADDRESS;
      diskAddress : LibSys.TypeFilePos;
          (* The preceding fields have to be defined contiguously and in
             this order *)
      size        : CARDINAL;
      modified    : BOOLEAN;      (* Modified in memory, but not in disk *)
      objId       : TypeObjId;    (* For transactions and free chain list *)
      transCount  : CARDINAL;     (* Number of simultaneous transactions *)
      boxList     : TypeBoxId;    (* First box of linked boxes *)
    END;
```

Hay que decir que el atributo `boxList`, al igual que sucede con `memAddress`, no tiene significado cuando se lee del disco, puesto que las cajas son objetos temporales que no sobreviven entre distintas sesiones de trabajo de Lope.

Hay otra modificación del fichero `StoTbl.def`: la definición de los identificadores de caja nula (`NullBoxId`, que queda en el valor `0`) y de vista nula (`NullViewId`, también con el valor `0`). Las definiciones se aplazan hasta este módulo de la misma forma que se hace con el objeto nulo, `NullObjId`.

3. **Modificaciones adicionales de `StoTbl.mod`**. Ha sido necesario modificar también el procedimiento interno `EscribirTDO` con el fin de guardar los objetos con listas de cajas vacías, al igual que la dirección `memAddress` se guarda al valor `NullAddress`: porque son valores que no tienen significado en nuevas sesiones de trabajo.

4. **Creación de los esquemas de formatos**. Los esquemas de formatos deben ser creados automáticamente al crear una nueva base de datos. Asímismo el directorio "*format*" debe existir. Para ello, se modifica el módulo `ObjLocal`, añadiendo un nuevo procedimiento `CreateFormatSchemas`, que crea los esquemas de formatos. Además se modifica `CreateRootDirectory` para incluir la creación del directorio de formatos. La constante exportada `IdentifierObjectFormat` apuntará a ese directorio.

   También modificamos el procedimiento `Create` de `ObjBase.mod` para que llame a `ObjLocal.CreateFormatSchemas` al crearse una nueva base de datos.

5. **Modificaciones a `Object`**. En el módulo `Object` se producen dos modificaciones importantes. La primera de ellas se refiere al manejo del nuevo atributo comentado antes, la lista de caja, con la exportación de nuevas funciones para tratar la lista: la función `GetBoxList` y la función `SetBoxList`, que permiten, respectivamente, obtener o cambiar la lista (de recorrerla se ocupan directamente los módulos diseñados específicamente para este trabajo). Hay otras modificaciones que tienen lugar, pero de poca importancia: el hecho de definir los identificadores de caja y de vista nulos, que se importan de `Store`, al igual que se hace con el identificador de objeto nulo. Sin embargo, la modificación más importante del módulo `Object` es la que se realiza en todos los procedimientos de cambio de la base de datos, con el fin de actualizar automáticamente las posibles vistas relacionadas.

Las modificaciones de este módulo son en general, añadir llamadas a `RefreshViews` u otra más apropiada, después de realizar las otras operaciones para actualizar el objeto. Concretamente, se añaden:

- En `AssignValue(o,v)`, llamada a `RefreshViews(o)`
- En `AssignObjectValue(o,p)`, llamada a `RefreshViews(o)`
- En `ReplaceNthComponent(father,n,child)`, llamada a `RefreshViewsByReplacedComponent(father,n)`
- En `ReplaceTypedComponent` una llamada similar a la anterior
- En `InsertNthComponent(father,n,child)`, una llamada a `RefreshViewsByInsertedComponent(father,n)`
- En `ExtractNthComponent(father,n)`, una llamada a `RefreshViewsByDeletedComponent(father,n)`
- En `ReplaceNthName`, llamada a `RefreshViews(child)`
- En `ReplaceName`, llamada similar a la anterior
- En `InsertNamedComponent(father,name,child)`, una llamada `RefreshViewsByInsertedComponent(father,n)`, donde n ha de obtenerse buscando la posición de la componente nombrada en el objeto padre
- En `ReplaceNamedComponent` una llamada a `RefreshViewsByReplacedComponent`, buscando el índice n como en el punto anterior
- En `ExtractNamedComponent` una llamada a `RefreshViewsByDeletedComponent` también obteniendo el índice n

6. **Modificaciones a `Store`**. Las modificaciones son simplemente añadir las implementaciones a bajo nivel para obtener o modificar la lista de cajas de cada objeto (procedimientos `ObjGetBoxList` y `ObjSetBoxList`).

Además, los identificadores de caja y vista nulos, se declaran importados de `StoTbl`, al igual que sucede con el objeto nulo.

Capítulo 6

Ejemplo de utilización

En este capítulo mostraremos un ejemplo sencillo que muestra el funcionamiento del subsistema desarrollado.

El ejemplo, básicamente, lo único que hace es crear objetos y modificarlos; con la particularidad de que en todo momento va a existir una vista que represente al objeto que estamos tocando, de tal forma que se observará que, paralelamente a la modificación de los objetos se van modificando automáticamente las representaciones existentes.

Veamos, ante todo, el código fuente del ejemplo, para comentar a continuación sus resultados además de algunos aspectos del programa.

6.1 Código fuente

6.1.1 Código del programa

```
(* $Id:$ *)
(*****************************************************************************
*                              PROJECT  Lope
*============================================================================*
```

```
*
* SUBSYSTEM:  Visual     Visualisation Subsystem
* MODULE:  VisTest
* FILE:    VisTest.mod
* FILE TYPE:         ( ) Definition       ( ) Foreign definition
*                    ( ) Implementation   (x) Main program
*
* HISTORY:
*     DATE         AUTHOR                  DESCRIPTION
*   ---------    -------------    ----------------------------------------
*   17-Aug-96    Juan J. Amor     Initial Version  (file creation)
*
* DESCRIPTION:
*   Test program of the Visualisation Subsystem. It is based on the
*   document format (formDoc) and a sample document object. These objects
*   must exist in Lope Database. For this purpose, ObjXport and files
*   schemas.in, formats.in and objects.in can be used.
*
* SEE ALSO:
*   -
*
***************************************************************************)

MODULE VisTest;

IMPORT VisView;
IMPORT VisViewL;
IMPORT VisBox;
IMPORT VisBoxL;
IMPORT LibErr;
IMPORT LibIO;
IMPORT ObjBase;
IMPORT VT;
IMPORT ObjSymb;
IMPORT Object;
IMPORT ObjLocal;

VAR ot1, ot2, format,newobj,docum1,object,direj1,
       sections,subsect: Object.ObjectId;
    vscreen: VT.TypeTextP;
    view: VisViewL.TypeViewId;
    viewName: ObjSymb.Symbol;

    viewProp: VisViewL.TypeView;

    box: VisBoxL.TypeBoxId;
    boxProp: VisBoxL.TypeBox;

    IdentifierObjectFormat: Object.ObjectId;
    (* In the future, this variable must be defined in ObjLocal *)

    c:CHAR;
```

```
    dia,mes,anyo: CARDINAL;
    x,y: CARDINAL;

    nameOfObject,nameOfFormat: ARRAY[0..63] OF CHAR;
BEGIN

  (* Avoid errors *)
  LibErr.StopOnError := FALSE;
  LibErr.TraceErrors := TRUE;

  ObjBase.Open("TESTBASE");

  (* Get the value for IdentifierObjectFormat. In the future, this can
     be defined in ObjLocal *)
  Object.GetNamedComponent(ObjLocal.IdentifierObjectRoot,
    ObjSymb.Code("format"),IdentifierObjectFormat);

  (* Obtain the directory with the object, and then the object *)
  Object.GetNamedComponent(ObjLocal.IdentifierObjectRoot,
    ObjSymb.Code("objects"),direj1);

  Object.GetNamedComponent(direj1,ObjSymb.Code("docum1"),object);

  (* Obtain the format *)
  Object.GetNamedComponent(IdentifierObjectFormat, ObjSymb.Code("formDoc"),
    format);

(** **)

  (* TEST 1: Simple creation, modification and destroy of a view *)

  (* Create the view *)
  VisView.Create(object,format,500,500,0,0,vscreen,view);

  VisViewL.GetProperties(view,viewProp);

(*
  VisBox.DumpBox(viewProp.boxId);
  ObjBase.Close;
  HALT;
*)

  (* Now, wait for key ... *)
  LibIO.ReadChar(c);

  (* First section (introduction), it's erroneos... word "la" expected *)
  Object.GetNamedComponent(ObjLocal.IdentifierObjectRoot,
    ObjSymb.Code("objects"),direj1);

  (* Locate the desired paragraph line *)
  Object.GetNamedComponent(direj1,ObjSymb.Code("docum1"),docum1);
  Object.GetTypedComponent(docum1,ObjSymb.Code("secciones"),ot2);
  Object.GetNthComponent(ot2,1,ot1);
```

```
  Object.GetTypedComponent(ot1,ObjSymb.Code("Cuerpo"),ot2);
  Object.GetTypedComponent(ot2,ObjSymb.Code("Parrafos"),ot1);
  Object.GetNthComponent(ot1,1,ot2);
  Object.GetNthComponent(ot2,1,ot1);

  (* Prepare the new Palabra object *)
  Object.Create(ObjSymb.Code("Palabra"),newobj);

  Object.AssignValue(newobj,"erronea");

  (* Insert into the Linea object *)
  Object.InsertNthComponent(ot1,4,newobj);

  (* Refresh the screen -- needed until new VT running ... *)
  VT.TextPRedrawClientArea(viewProp.textPanel,VT.Normal);

  (* Wait for a key ... *)
  LibIO.ReadChar(c);

  (* Change the value: its correct value is "la" *)
  Object.AssignValue(newobj,"la");

  (* Refresh the screen *)
  VT.TextPRedrawClientArea(viewProp.textPanel,VT.Normal);

  (* Wait for a key ... *)
  LibIO.ReadChar(c);

  (* Destroy the view *)
  VisView.Delete(view);

(** **)

  (* TEST 2: Change the month of Fecha object *)

  (* Create the view again *)
  VisView.Create(object,format,500,500,0,0,vscreen,view);

  VisViewL.GetProperties(view,viewProp);

  (* Wait for a key ... *)
  LibIO.ReadChar(c);

  (* Locate the Fecha object, we will modify the month, from July (7) to
     August (8) *)
  Object.GetTypedComponent(docum1,ObjSymb.Code("Fecha"),ot1);
  Object.GetTypedComponent(ot1,ObjSymb.Code("mes"),ot2);

  mes := 8;

  Object.AssignValue(ot2,mes);

  (* Refresh the screen *)
```

```
VT.TextPRedrawClientArea(viewProp.textPanel,VT.Normal);

(* Wait for a key ... *)
LibIO.ReadChar(c);

(* TEST 3: Replace the entire Fecha object *)

(* Prepare a new Fecha object *)

Object.Create(ObjSymb.Code("Fecha"),newobj);
Object.GetTypedComponent(newobj,ObjSymb.Code("dia"),ot1);
dia := 15;
Object.AssignValue(ot1,dia);
Object.GetTypedComponent(newobj,ObjSymb.Code("mes"),ot1);
mes := 10;
Object.AssignValue(ot1,mes);
Object.GetTypedComponent(newobj,ObjSymb.Code("an~o"),ot1);
anyo := 1997;
Object.AssignValue(ot1,anyo);

(* Replace the old Fecha object with new *)
Object.ReplaceTypedComponent(docum1,newobj);

(* Refresh the screen *)
VT.TextPRedrawClientArea(viewProp.textPanel,VT.Normal);

(* Wait for a key ... *)
LibIO.ReadChar(c);

(* TEST 4: Extracting a subsection *)

(* Extract a subsection *)
Object.GetTypedComponent(docum1,ObjSymb.Code("secciones"),ot1);
Object.GetNthComponent(ot1,2,ot2);
Object.GetTypedComponent(ot2,ObjSymb.Code("Cuerpo"),ot1);
Object.GetTypedComponent(ot1,ObjSymb.Code("secciones"),sections);
Object.ExtractNthComponent(sections,1,subsect);

(* Refresh the screen *)
VT.TextPRedrawClientArea(viewProp.textPanel,VT.Normal);

(* Wait for a key ... *)
LibIO.ReadChar(c);

(* TEST 5: Moving (extract and reinsert) a section *)

(* Extract a section *)
Object.GetTypedComponent(docum1,ObjSymb.Code("secciones"),ot2);
Object.ExtractNthComponent(ot2,1,ot1);

(* Refresh the screen *)
VT.TextPRedrawClientArea(viewProp.textPanel,VT.Normal);
```

```
  (* Wait for a key ... *)
  LibIO.ReadChar(c);

  (* Insert at bottom of document *)
  Object.InsertNthComponent(ot2,2,ot1);

  (* Refresh the screen *)
  VT.TextPRedrawClientArea(viewProp.textPanel,VT.Normal);

  (* Wait for a key ... *)
  LibIO.ReadChar(c);

  (* TEST 6: Reinsert the subsection *)

  (* Reinsert the subsection *)
  Object.InsertNthComponent(sections,1,subsect);

  (* Refresh the screen *)
  VT.TextPRedrawClientArea(viewProp.textPanel,VT.Normal);

  (* Wait for a key ... *)
  LibIO.ReadChar(c);

  (* Delete the view *)
  VisView.Delete(view);

  (* Close the database *)
  ObjBase.Close;

END VisTest.
```

6.1.2 Objeto de prueba

```
objects ;

(%) [%

  docum1 (DocRequisitos) [+
    (Titulo) [+
      (Referencias) [*
      ]
      (.text.) "Lope-Visualizacion"
    ]
    (Autor) [+
      (Referencias) [*
      ]
      (.text.) "Juan\ Jose\ Amor"
    ]
    (Fecha) [+
```

```
      (dia) 17
      (mes) 7
      (an~o) 1997
      ]
    (secciones) [*
      (Seccion) [+
        (Titulo) [+
          (Referencias) [*
          ]
          (.text.) "Introduccion"
        ]
        (Cuerpo) [|
          (Parrafos) [*
            (Parrafo) [*
              (Linea) [*
                (Palabra) "Este"
                (Palabra) "trabajo"
                (Palabra) "cubre"
                (Palabra) "visualizacion"
                (Palabra) "de"
                (Palabra) "Lope"
                ]
              (Linea) [*
                (Palabra) "Es"
                (Palabra) "un"
                (Palabra) "subsistema"
                (Palabra) "imprescindible"
                ]
              (Linea) [*
                (Palabra) "Permite"
                (Palabra) "visualizar"
                (Palabra) "cualquier"
                (Palabra) "objeto"
                ]
            ]
          ]
        ]
      ]
      (Seccion) [+
        (Titulo) [+
          (Referencias) [*
          ]
          (.text.) "Especificaciones"
        ]
        (Cuerpo) [|
          (secciones) [*
            (Seccion) [+
              (Titulo) [+
                (Referencias) [*
                ]
                (.text.) "Ideas\ Generales"
              ]
              (Cuerpo) [|
```

```
      (Parrafos) [*
        (Parrafo) [*
          (Linea) [*
            (Palabra) "El"
            (Palabra) "elemento"
            (Palabra) "clave"
            (Palabra) "es"
            (Palabra) "el"
            (Palabra) "formato"
          ]
          (Linea) [*
            (Palabra) "Todo"
            (Palabra) "se"
            (Palabra) "organiza"
            (Palabra) "en"
            (Palabra) "jerarquias"
            (Palabra) "de"
            (Palabra) "cajas"
          ]
        ]
      ]
    ]
  ]
  (Seccion) [+
    (Titulo) [+
      (Referencias) [*
      ]
      (.text.) "Desarrollo"
    ]
    (Cuerpo) [|
      (secciones) [*
        (Seccion) [+
          (Titulo) [+
            (Referencias) [*
            ]
            (.text.) "Los\ formatos"
          ]
          (Cuerpo) [|
            (Parrafos) [*
              (Parrafo) [*
                (Linea) [*
                  (Palabra) "Explicar"
                  (Palabra) "los"
                  (Palabra) "formatos"
                ]
              ]
            ]
          ]
        ]
        (Seccion) [+
          (Titulo) [+
            (Referencias) [*
            ]
```

```
                      (.text.) "Esquemas"
                    ]
                    (Cuerpo) [|
                      (Parrafos) [*
                        (Parrafo) [*
                          (Linea) [*
                            (Palabra) "Esquemas"
                            (Palabra) "de"
                            (Palabra) "formatos"
                          ]
                        ]
                      ]
                    ]
                  ]
                ]
              ]
            ]
          ]
        ]
      ]
    ]
  ]

]
```

6.2 Resultados

6.2.1 Comentarios previos

Se puede observar que el programa anterior no contiene procedimientos, es directamente un programa principal que realiza diversas modificaciones a una serie de objetos que deben existir. Como se indica en el propio listado, dichos objetos son los que se incluyen junto al mismo en la sección anterior, y pueden introducirse en la base de datos utilizando la aplicación `ObjXport` que hemos mencionado varias veces a lo largo de este trabajo. Además, deberán haberse definido los correspondientes esquemas y formatos, que son los mismos que se mostraron en la sección 4.4.

Adicionalmente hay que comentar el hecho de que en este listado, cada vez que haya una modificación tengamos que llamar a la función de refresco del

terminal virtual. Esto es así pues en el momento de terminar este trabajo aun no se había desarrollado lo suficiente el nuevo terminal virtual, con lo que hubo que implementar unos módulos simuladores con la misma interfaz, pero que no son capaces de refrescar automáticamente las modificaciones.

Por este mismo motivo, la inexistencia de un terminal virtual definitivo, algunas funciones no pueden probarse por no causar efecto alguno en la presentación. Nos estamos refiriendo a las funciones para resaltar zonas de representación.

6.2.2 Resultados en pantalla

Las siguientes figuras (6.1 a 6.9) muestran lo que se va viendo en pantalla conforme la ejecución del programa avanza; justo en el momento en el que se pide al usuario que pulse una tecla.

Como vemos en la primera figura, la 6.1, así es el resultado obtenido nada más crear la vista. Vemos que el objetivo del formato se ha cumplido: se representan las secciones indentadas y numeradas automáticamente.

La primera modificación que se realiza consiste en insertar una palabra en una de las líneas. Se trata de dejar la primera frase de la introducción como "*Este trabajo cubre* **la** *visualización de* LOPE", es decir, insertar el artículo *la*. Inicialmente, se insertará otra palabra para luego reemplazarla por la correcta (ver figuras 6.2 y 6.3). Al hacer esto vemos cómo se mueven correctamente los textos cercanos: esto es el resultado del mecanismo de *realineación* que, recordemos, garantiza una recolocación correcta de todas las cajas haciendo un número mínimo de movimientos.

A continuación, el programa destruye la vista (lo cual provocará la desaparición de la representación) y se simula la necesidad de crear de nuevo la vista sobre el objeto, con lo que generará de nuevo la salida de la figura 6.3. La siguiente modificación ha sido cambiar el mes de la fecha de realización, pasarlo de 7 (Julio) a 8 (Agosto). El resultado se ve en la figura 6.4.

```
Lope-Visualizacion

Juan Jose Amor
17 de Julio de 1997

 1. Introduccion
  Este trabajo cubre visualizacion de Lope
  Es un subsistema imprescindible
  Permite visualizar cualquier objeto

 2. Especificaciones
   2.1. Ideas Generales
    El elemento clave es el formato
    Todo se organiza en jerarquias de cajas

   2.2. Desarrollo
     2.2.1. Los formatos
      Explicar los formatos

     2.2.2. Esquemas
      Esquemas de formatos
```

Figura 6.1: Estado al crear la vista

```
Lope-Visualizacion

Juan Jose Amor
17 de Julio de 1997

 1. Introduccion
  Este trabajo cubre erronea visualizacion de Lope
  Es un subsistema imprescindible
  Permite visualizar cualquier objeto

 2. Especificaciones
   2.1. Ideas Generales
    El elemento clave es el formato
    Todo se organiza en jerarquias de cajas

   2.2. Desarrollo
     2.2.1. Los formatos
      Explicar los formatos

     2.2.2. Esquemas
      Esquemas de formatos
```

Figura 6.2: Inserción de una palabra

```
Lope-Visualizacion


Juan Jose Amor

17 de Julio de 1997


 1. Introduccion

  Este trabajo cubre la visualizacion de Lope

  Es un subsistema imprescindible

  Permite visualizar cualquier objeto


 2. Especificaciones

   2.1. Ideas Generales

    El elemento clave es el formato

    Todo se organiza en jerarquias de cajas


   2.2. Desarrollo

     2.2.1. Los formatos

      Explicar los formatos


     2.2.2. Esquemas

      Esquemas de formatos
```

Figura 6.3: Reemplazo de una palabra

```
Lope-Visualizacion

Juan Jose Amor
17 de Agosto de 1997

 1. Introduccion
  Este trabajo cubre la visualizacion de Lope
  Es un subsistema imprescindible
  Permite visualizar cualquier objeto

 2. Especificaciones
   2.1. Ideas Generales
    El elemento clave es el formato
    Todo se organiza en jerarquias de cajas

   2.2. Desarrollo
     2.2.1. Los formatos
      Explicar los formatos

     2.2.2. Esquemas
      Esquemas de formatos
```

Figura 6.4: Cambio del mes

La siguiente modificación ha sido de reemplazo de la fecha entera por un nuevo objeto de ese tipo. En la figura 6.5 puede verse el resultado. Hay que resaltar la diferencia con el ejemplo anterior: antes se modificó directamente el valor del objeto `mes`. Ahora se ha reemplazado todo el objeto `fecha` con uno nuevo. Por lo tanto, internamente se han usado funciones de refresco distintas: `RefreshViews` y `RefreshViewsByReplacedComponent`, respectivamente.

Ahora queremos mostrar el efecto de la numeración automática de las secciones. Para ello extraemos la sección con las *Ideas Generales* del documento, y el resultado puede verse en la figura 6.6: al desaparecer esa sección (era la 2.1), la siguiente se mueve hacia arriba y además pasa a ser la 2.1.

En el siguiente ejemplo seguimos eliminando secciones: ahora borramos la *Introducción*. El resultado se ve en la figura 6.7: ahora la sección primera es la que antes era la segunda, por lo que se recoloca ésta y todas las subsecciones se renumeran acorde a la nueva situación.

Estamos terminando con las modificaciones. En primer lugar, reinsertamos la introducción antes extraida, pero al final. Por lo tanto, queda como segunda sección (ver figura 6.8). Finalmente, reinsertamos la subsección de *ideas generales* estraida antes, en el mismo sitio que ocupaba. El resultado puede verse en la figura 6.9.

Este ejemplo ha pretendido demostrar el funcionamiento del subsistema en todos los casos posibles de modificación del objeto representado.

La inexistencia de un terminal virtual terminado nos impide, de momento, demostrar el funcionamiento de un editor básico, aunque las ideas de cómo debe ser el mismo se han expuesto ya en capítulos anteriores (véase la página 48, por ejemplo). Por lo tanto, haciendo uso del nuevo terminal virtual y el subsistema desarrollado en este trabajo, sería posible implementar de forma sencilla el *Browser-editor* de objetos realizado en Luque [11].

```
Lope-Visualizacion

Juan Jose Amor
15 de Octubre de 1997

 1. Introduccion
  Este trabajo cubre la visualizacion de Lope
  Es un subsistema imprescindible
  Permite visualizar cualquier objeto

 2. Especificaciones
   2.1. Ideas Generales
    El elemento clave es el formato
    Todo se organiza en jerarquias de cajas

   2.2. Desarrollo
     2.2.1. Los formatos
      Explicar los formatos

     2.2.2. Esquemas
      Esquemas de formatos
```

Figura 6.5: Reemplazo de todo el objeto fecha

```
Lope-Visualizacion

Juan Jose Amor
15 de Octubre de 1997

 1. Introduccion
  Este trabajo cubre la visualizacion de Lope
  Es un subsistema imprescindible
  Permite visualizar cualquier objeto

 2. Especificaciones
   2.1. Desarrollo
     2.1.1. Los formatos
      Explicar los formatos

     2.1.2. Esquemas
      Esquemas de formatos
```

Figura 6.6: Extraccion de la sección *Ideas Generales*

```
Lope-Visualizacion


Juan Jose Amor

15 de Octubre de 1997


 1. Especificaciones

   1.1. Desarrollo

     1.1.1. Los formatos

      Explicar los formatos


     1.1.2. Esquemas

      Esquemas de formatos
```

Figura 6.7: Estracción de la Introducción

```
Lope-Visualizacion

Juan Jose Amor
15 de Octubre de 1997

 1. Especificaciones
   1.1. Desarrollo
     1.1.1. Los formatos
      Explicar los formatos

     1.1.2. Esquemas
      Esquemas de formatos

 2. Introduccion
  Este trabajo cubre la visualizacion de Lope
  Es un subsistema imprescindible
  Permite visualizar cualquier objeto
```

Figura 6.8: Reinserción de la Introducción al final

```
Lope-Visualizacion

Juan Jose Amor
15 de Octubre de 1997

 1. Especificaciones
   1.1. Ideas Generales
    El elemento clave es el formato
    Todo se organiza en jerarquias de cajas

   1.2. Desarrollo
     1.2.1. Los formatos
      Explicar los formatos

     1.2.2. Esquemas
      Esquemas de formatos

 2. Introduccion
  Este trabajo cubre la visualizacion de Lope
  Es un subsistema imprescindible
  Permite visualizar cualquier objeto
```

Figura 6.9: Reinserción de la sección *Ideas Generales*

Capítulo 7

Código fuente

7.1 Módulos de definición

7.1.1 Módulo VisBoxL.def

```
(* $Id$ *)
(*****************************************************************************
*                            PROJECT  Lope
*============================================================================*
*
* SUBSYSTEM:  Visual     Visualisation Subsystem
* MODULE:  Box          Box manager
* FILE:    VisBoxL.def
* FILE TYPE:         (x) Definition       ( ) Foreign definition
*                    ( ) Implementation   ( ) Main program
*
* HISTORY:
*    DATE          AUTHOR                  DESCRIPTION
*   ---------    -------------    ----------------------------------------
*   14-Apr-97    Juan J. Amor     Initial Version  (file creation)
*
* SEE ALSO:
*
*****************************************************************************)

DEFINITION MODULE VisBoxL;
```

```
IMPORT Object;
IMPORT VT;

CONST MaxText = 63; (* Max. lenght of text in a terminal box *)

      MaxBoxId = 131071; (* 128 K-boxes *)

      NullBoxId = Object.NullBoxId;

TYPE TypeBoxId = Object.TypeBoxId;

TYPE TypeAlign = (HorCent,HorUp,HorDown,VerCent,VerRight,VerLeft);

TYPE TypeString = ARRAY[0..MaxText] OF CHAR;

TYPE TBoxListP = POINTER TO TBoxList;

TYPE TypeBoxList = RECORD
                     top: TBoxListP;
                     bottom: TBoxListP;
                   END;

TYPE TypeBox = RECORD
     xPos, yPos: INTEGER;      (* Relative position *)
     width, height: INTEGER;   (* Box size *)
     mode: VT.TypeMode;           (* Presentation mode *)
     CASE isTerminal: BOOLEAN OF
        TRUE: text: TypeString; (* terminal box *)
     |   FALSE: boxList: TypeBoxList;   (* composed box *)
     END;
     numberOfSons: CARDINAL;   (* 0 for terminal boxes *)
     index: CARDINAL;          (* Position at father's list of boxes *)
     fatherRef: TypeBoxId;
     viewRef: Object.ObjectId;
     objectRef: Object.ObjectId;
     formatRef: Object.ObjectId;
     prevBox: TypeBoxId;       (* prevBox and nextBox are used for use *)
     nextBox: TypeBoxId;       (* in lists associated with objects     *)
     sameObjectLink: TypeBoxId;
END;

TYPE TBoxList = RECORD
     box: TypeBoxId;
     alignCode: TypeAlign; (* Align code, of this box, relative to next box.
                              Not used in last box of list *)
     next: TBoxListP;
     prev: TBoxListP;
END;

(*==========================================================================*
*                         FUNCTION:  Create
*-------------------------------------------------------------------------*)
```

```
PROCEDURE Create(
     (* Out *) VAR box: TypeBoxId
);

(**
* ARGUMENTS:
*    box: Box to create
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   -
*
* EFFECT:
*   - Creates a box object, and assigns an identifier, which is returned.
*     Initial values of box properties are:
*
*    xPos = 0, yPos = 0
*    width = 0, height = 0
*    mode = VT.Normal
*    isTerminal = TRUE
*    text = ""
*    numberOfSons = 0
*    index = 0
*    fatherRef = NullBoxId
*    viewRef = Object.NullObjId
*    objectRef: Object.NullObjId
*    formatRef: Object.NullObjId
*    prevBox: NullBoxId
*    nextBox: NullBoxId
*    sameObjectLink: (the identifier assigned)
*
* EXCEPTIONS:
*
* SEE ALSO:
*
*==========================================================================*)

(*==========================================================================*
*                          FUNCTION:  Delete
*--------------------------------------------------------------------------*)

PROCEDURE Delete(
     (* InOut *) VAR box: TypeBoxId
);

(**
* ARGUMENTS:
*   - box: Box to delete
*
* GLOBAL VARIABLES (USED):
```

```
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - The box must exist.
*
* EFFECT:
*   - Deletes the box, freeing memory and the box identifier
*
* EXCEPTIONS:
*   - If the box is NullBoxId or does not exist, raises an error
*
* SEE ALSO:
*
*============================================================================*)

(*============================================================================*
*                         FUNCTION:  GetProperties
*----------------------------------------------------------------------------*)

PROCEDURE GetProperties(
     (* In *) box: TypeBoxId;
     (* Out *) VAR boxProperties: TypeBox
);

(**
* ARGUMENTS:
*   - box: Box to read
*   - boxProperties: Variable with properties
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - The box must exist.
*
* EFFECT:
*   - Return all properties of this box
*
* EXCEPTIONS:
*   - If the box is NullBoxId or does not exist, raises an error
*
* SEE ALSO:
*
*============================================================================*)

(*============================================================================*
*                         FUNCTION:  SetProperties
*----------------------------------------------------------------------------*)

PROCEDURE SetProperties(
     (* In *) box: TypeBoxId;
```

```
    (* In *) boxProperties: TypeBox
);

(**
* ARGUMENTS:
*   - box: Box to modify
*   - boxProperties: Variable with properties
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - The box must exist.
*
* EFFECT:
*   - Modify the box properties with new ones
*
* EXCEPTIONS:
*   - If the box is NullBoxId or does not exist, raises an error
*
* SEE ALSO:
*
*==============================================================================*)

END VisBoxL.
```

7.1.2 Módulo `VisViewL.def`

```
(* $Id$ *)
(*******************************************************************************
*                            PROJECT  Lope
*==============================================================================*
*
* SUBSYSTEM:  Visual      Visualisation Subsystem
* MODULE:  View           View manager
* FILE:    VisViewL.def
* FILE TYPE:          (x) Definition       ( ) Foreign definition
*                     ( ) Implementation   ( ) Main program
*
* HISTORY:
*     DATE          AUTHOR                   DESCRIPTION
*   ---------    -------------    ---------------------------------------
*   21-May-97     Juan J. Amor    Initial version (file creation)
*
* SEE ALSO:
*
*******************************************************************************)
```

```
DEFINITION MODULE VisViewL;

IMPORT Object;

IMPORT VT;

CONST MaxViewId = 255; (* 255 views *)

CONST NullViewId = 0;
CONST NullBoxId = 0;
CONST NullWinId = 0;

TYPE TypeViewId = Object.TypeViewId;

TYPE TypeVirtualScreen = POINTER TO TypeVScreen;

(* This prototype uses only screens of:
     max 200 rows    (maxY)
     max 200 columns (maxX)
*)

CONST maxX = 200;
      maxY = 200;

TYPE TypeVScreen = ARRAY [0..maxY],[0..maxX] OF RECORD
                     value: CHAR;
                     mode: VT.TypeMode;
                   END;

TYPE TypeView = RECORD

   formatRef: Object.ObjectId;
   objectRef: Object.ObjectId;

   boxId: Object.TypeBoxId;

   oldHeight, oldWidth: INTEGER;

   virtualScreen: TypeVirtualScreen;

   textPanel: VT.TypeTextP;

END;


(*=============================================================================*
*                          FUNCTION:  Create
*-----------------------------------------------------------------------------*)

PROCEDURE Create(
     (* Out *) VAR view: TypeViewId
);
```

```
(**
* ARGUMENTS:
*    view: View to create
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   -
*
* EFFECT:
*   - Creates a view object, and assigns an identifier, which is returned.
*     Initial values of view properties are:
*
*    formatRef = NullObjId
*    objectRef = NullObjId
*    boxId = NullBoxId
*    oldHeight = 0
*    oldWidth = 0
*    textPanel = (desconocido)
*    virtualScreen = NIL
*
* EXCEPTIONS:
*
* SEE ALSO:
*
*============================================================================*)

(*============================================================================*
*                          FUNCTION:  Delete
*----------------------------------------------------------------------------*)

PROCEDURE Delete(
     (* InOut *) VAR view: TypeViewId
);

(**
* ARGUMENTS:
*   - view: View to delete
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - The view must exist.
*
* EFFECT:
*   - Deletes the view, freeing memory (not virtualScreen nor textPanel)
*     and the view identifier
*
* EXCEPTIONS:
```

```
*   - If the view is NullViewId or does not exist, raises an error
*
* SEE ALSO:
*
*=============================================================================*)

(*=============================================================================*
*                          FUNCTION:  GetProperties
*-----------------------------------------------------------------------------*)

PROCEDURE GetProperties(
     (* In *) view: TypeViewId;
     (* Out *) VAR viewProperties: TypeView
);

(**
* ARGUMENTS:
*   - view: view to read
*   - viewProperties: Variable with properties
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - The view must exist.
*
* EFFECT:
*   - Return all properties of this view
*
* EXCEPTIONS:
*   - If the view is NullViewId or does not exist, raises an error
*
* SEE ALSO:
*
*=============================================================================*)


(*=============================================================================*
*                          FUNCTION:  SetProperties
*-----------------------------------------------------------------------------*)

PROCEDURE SetProperties(
     (* In *) view: TypeViewId;
     (* In *) viewProperties: TypeView
);

(**
* ARGUMENTS:
*   - view: View to modify
*   - viewProperties: Variable with properties
*
* GLOBAL VARIABLES (USED):
```

```
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - The view must exist.
*
* EFFECT:
*   - Modify the view properties with new ones
*
* EXCEPTIONS:
*   - If the view is NullViewId or does not exist, raises an error
*
* SEE ALSO:
*
*===========================================================================*)

END VisViewL.
```

7.1.3 Módulo `VisBox.def`

```
(* $Id$ *)
(*****************************************************************************
*                            PROJECT  Lope
*============================================================================*
*
* SUBSYSTEM:  Visual     Visualisation Subsystem
* MODULE:  Box          Box manager
* FILE:    VisBox.def
* FILE TYPE:         (x) Definition      ( ) Foreign definition
*                    ( ) Implementation   ( ) Main program
*
* HISTORY:
*     DATE          AUTHOR                 DESCRIPTION
*   ---------    -------------    ----------------------------------------
*   24-Sep-96    Juan J. Amor     Initial Version  (file creation)
*   17-Nov-96    Juan J. Amor     Added Destroy procedure
*   14-Mar-97    Juan J. Amor     Added ReAlign, ReplaceNthBox procedures
*   14-Apr-97    Juan J. Amor     Added low-level box handling interface
*
* SEE ALSO:
*
*****************************************************************************)

DEFINITION MODULE VisBox;

IMPORT Object;
IMPORT VisBoxL;
IMPORT VT;
```

```
(*==========================================================================*
*                          FUNCTION:  DumpBox
*--------------------------------------------------------------------------*)

PROCEDURE DumpBox(
     (* In *) box: VisBoxL.TypeBoxId
);

(* DEBUG FUNCTION *)

(*==========================================================================*
*                          FUNCTION:  Compose
*--------------------------------------------------------------------------*)

PROCEDURE Compose(
     (* In *) object: Object.ObjectId;
     (* In *) format: Object.ObjectId;
     (* In *) view: Object.ObjectId;
     (* In *) mode: VT.TypeMode;
     (* Out *) VAR box: VisBoxL.TypeBoxId
);

(**
* ARGUMENTS:
*    object: Object to view
*    format: Format to use in box hierarchy
*    view: Identifier of view owner of this box
*    mode: Mode of presentation
*    box: Box hierarchy returned
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - The object and format must exist. The box must be NullBoxId.
*
* EFFECT:
*   - Recursively builds the box hierarchy, using the information about
*     the format, and using data of the object.
*
* EXCEPTIONS:
*
* SEE ALSO:
*
*==========================================================================*)

(*==========================================================================*
*                          FUNCTION:  Align
*--------------------------------------------------------------------------*)

PROCEDURE Align(
     (* InOut *) VAR box: VisBoxL.TypeBoxId
```

```
);

(**
* ARGUMENTS:
*    box: Box hierarchy to align
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - The box hierarchy must be built (usually, using the above
*     function "Compose" )
*
* EFFECT:
*   - Aligns the box hierarchy, using the rules specified in its format
*
* EXCEPTIONS:
*
* SEE ALSO:
*
*============================================================================*)

(*============================================================================*
*                          FUNCTION:  ReAlign
*----------------------------------------------------------------------------*)

PROCEDURE ReAlign(
     (* InOut *) VAR box: VisBoxL.TypeBoxId
);

(**
* ARGUMENTS:
*    box: Box hierarchy to realign
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - The box hierarchy must be built (usually, using the above
*     function "Compose" ), and aligned.
*
* EFFECT:
*   - Compute the position of given box in its mother, and then propagate
*     aligns to the top of box hierarchy.
*
* EXCEPTIONS:
*
* SEE ALSO:
*
*============================================================================*)
```

```
(*==========================================================================*
*                         FUNCTION:  Build
*--------------------------------------------------------------------------*)

PROCEDURE Build(
     (* In *) object: Object.ObjectId;
     (* In *) format: Object.ObjectId;
     (* In *) view: Object.ObjectId;
     (* In *) mode: VT.TypeMode;
     (* Out *) VAR box: VisBoxL.TypeBoxId
);

(**
* ARGUMENTS:
*    object: Object to view
*    format: Format to use in box hierarchy
*    box: Box hierarchy returned
*    view: Identifier of view owner of this box
*    mode: mode of presentation desired
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - Both format and object must exist. The box must be NullBoxId.
*
* EFFECT:
*   - Actually calls to Compose and then Align
*
* EXCEPTIONS:
*
* SEE ALSO:
*
*==========================================================================*)

(*==========================================================================*
*                         FUNCTION:  Destroy
*--------------------------------------------------------------------------*)

PROCEDURE Destroy(
     (* InOut *) VAR box:VisBoxL.TypeBoxId
);

(**
* ARGUMENTS:
*    box: Box hierarchy to destroy
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
```

```
*   - The box hierarchy must exist
*
* EFFECT:
*   - Recursively destroys the box hierarchy. Returns NullBoxId. Also delete
*     all boxes from lists on objects viewed.
*
*   - Note that Delete does not recursively destroy any hierarchy,
*     only deletes the requested box. If your box has son boxes,
*     use this function instead of Delete.
*
* EXCEPTIONS:
*
* SEE ALSO:
*
*============================================================================*)

(*============================================================================*
*                          FUNCTION:  Display
*----------------------------------------------------------------------------*)

PROCEDURE Display(
     (* InOut *) VAR box: VisBoxL.TypeBoxId;
     (* In *) mode: VT.TypeMode
);

(**
* ARGUMENTS:
*    box: Box hierarchy to change
*    mode: New mode
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - The box hierarchy must exist
*
* EFFECT:
*   - Changes mode of presentation to the box hierarchy, and the
*     associated virtual terminal.
*
* EXCEPTIONS:
*
* SEE ALSO:
*
*============================================================================*)

(*============================================================================*
*                          FUNCTION:  InsertNthBox
*----------------------------------------------------------------------------*)

PROCEDURE InsertNthBox(
     (* In *) newBox: VisBoxL.TypeBoxId;
```

```
     (* In *) index: CARDINAL;
     (* In *) alignCode: VisBoxL.TypeAlign;
     (* InOut *) VAR father: VisBoxL.TypeBoxId
);

(**
* ARGUMENTS:
*    newBox: Box to insert
*    index: position desired
*    alignCode: Align code desired
*    father: Box where newBox will be inserted
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - Both boxes must exist
*
* EFFECT:
*   - Inserts the new box in the Nth position ( box.index = index) of
*     father box. The value for first box in list is 1.
*
* EXCEPTIONS:
*
* SEE ALSO:
*
*==========================================================================*)

(*==========================================================================*
*                          FUNCTION:  ReplaceNthBox
*--------------------------------------------------------------------------*)

PROCEDURE ReplaceNthBox(
     (* In *) newBox: VisBoxL.TypeBoxId;
     (* In *) index: CARDINAL;
     (* InOut *) VAR father: VisBoxL.TypeBoxId
);

(**
* ARGUMENTS:
*    newBox: Box to replace
*    index: position desired
*    father: Box where newBox will be inserted instead of old nth box.
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - Both boxes must exist
*
* EFFECT:
```

```
*   - Logically equivalent to DeleteNthBox and then InsertNthBox, but more
*     efficiently. Also, uses the old align code.
*
* EXCEPTIONS:
*
* SEE ALSO:
*
*============================================================================*)

(*============================================================================*
*                         FUNCTION:  ReplaceNthBox
*----------------------------------------------------------------------------*)

PROCEDURE ReplaceAlignCode(
   (* In *) alignCode: VisBoxL.TypeAlign;
   (* In *) index: CARDINAL;
   (* InOut *) VAR father: VisBoxL.TypeBoxId
);

(**
* ARGUMENTS:
*    alignCode: new align code
*    index: position desired
*    father: Box where newBox will be inserted instead of old nth box.
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - Both boxes must exist
*
* EFFECT:
*   - Replaces the Nth align code
*
* EXCEPTIONS:
*
* SEE ALSO:
*
*============================================================================*)

(*============================================================================*
*                         FUNCTION:  DeleteNthBox
*----------------------------------------------------------------------------*)

PROCEDURE DeleteNthBox(
     (* In *) index: CARDINAL;
     (* InOut *) VAR father: VisBoxL.TypeBoxId
);

(**
* ARGUMENTS:
*    index: position desired
```

```
*    father: Father of box to be deleted
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - Box must exist, and with 'index' or more components
*
* EFFECT:
*   - Deletes the Nth box, the box which box.index = index, freeing the
*     son box.
*
* EXCEPTIONS:
*
* SEE ALSO:
*
*=========================================================================*)

(*=========================================================================*
*                          FUNCTION:  ChangeText
*-------------------------------------------------------------------------*)

PROCEDURE ChangeText(
     (* In *) newText: VisBoxL.TypeString;
     (* InOut *) VAR box: VisBoxL.TypeBoxId
);

(**
* ARGUMENTS:
*    newText: New text desired
*    box: Terminal box to be changed
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - Box must exist, and must be terminal
*
* EFFECT:
*   - Changes box's text property
*
* EXCEPTIONS:
*
* SEE ALSO:
*
*=========================================================================*)

END VisBox.
```

7.1.4 Módulo VisView.def

```
(* $Id$ *)
(*****************************************************************************
*                            PROJECT  Lope
*============================================================================*
*
* SUBSYSTEM:  Visual     Visualisation Subsystem
* MODULE:  View          View manager
* FILE:    VisView.def
* FILE TYPE:         (x) Definition       ( ) Foreign definition
*                    ( ) Implementation   ( ) Main program
*
* HISTORY:
*     DATE          AUTHOR                  DESCRIPTION
*   ---------    -------------    ----------------------------------------
*   24-Sep-96    Juan J. Amor     Initial Version  (file creation)
*    4-Apr-96    Juan J. Amor     Added support for modification of
*                                   viewed objects
*
* SEE ALSO:
*
*****************************************************************************)

DEFINITION MODULE VisView;

IMPORT Object;
IMPORT VT;
IMPORT VisBoxL;
IMPORT VisViewL;

(*===========================================================================*
*                          FUNCTION:  Create
*---------------------------------------------------------------------------*)

PROCEDURE Create(
     (* In *) object: Object.ObjectId;
     (* In *) format: Object.ObjectId;
     (* In *) virtWidth, virtHeight: CARDINAL;
     (* In *) physWidth, physHeight: CARDINAL;
     (* Out *) VAR textPanel: VT.TypeTextP;
     (* Out *) VAR view: VisViewL.TypeViewId
);

(**
* ARGUMENTS:
*   object: The object to be viewed
*   format: The format to apply
*   view: The view object
*   textPanel: The text panel to use (it will be created here)
*   virtWidth, virtHeight: The total (virtual) dimensions of the text panel
*   physWidth, physHeight: The physical dimensions of the text panel
```

```
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - The object, format and window must exist
*
* EFFECT:
*   - Creates a view related to the object and format passed as arguments.
*     This implies generate and align a box hierarchy, and display it in
*     the window.
*   - If physWidth = physHeight = 0, the panel dimensions will be equal to
*     the root box dimensions
*
* EXCEPTIONS:
*   -
*
* SEE ALSO:
*
*============================================================================*)

(*============================================================================*
*                         FUNCTION:  Delete
*----------------------------------------------------------------------------*)

PROCEDURE Delete(
     (* InOut *) VAR view:VisViewL.TypeViewId
);

(**
* ARGUMENTS:
*  view: The view to delete
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - The view must exist
*
* EFFECT:
*   - Destroy the view and related objects: box hierarchy and so on
*
* EXCEPTIONS:
*   -
*
* SEE ALSO:
*
*============================================================================*)

(*============================================================================*
*                         FUNCTION:  RefreshViews
```

```
*---------------------------------------------------------------------------*)

PROCEDURE RefreshViews(
     (* In *) object: Object.ObjectId
);

(**
* ARGUMENTS:
*  object: The modified object, which has a visual representation
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - The object must exist
*
* EFFECT:
*   - Refresh all representations of this object
*
* EXCEPTIONS:
*   -
*
* SEE ALSO:
*
*===========================================================================*)

(*===========================================================================*
*              FUNCTION:  RefreshViewsByInsertedComponent
*---------------------------------------------------------------------------*)

PROCEDURE RefreshViewsByInsertedComponent(
     (* In *) father: Object.ObjectId;
     (* In *) position: CARDINAL
);

(**
* ARGUMENTS:
*  father: The (composed) and modified object with a visual representation
*  position: The position of component inserted
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - The object father must exist
*
* EFFECT:
*   - If the object father is composed, and it is viewed via a list align
*     format, rebuild the view by inserting the desired
*     component as a son box. Also inserts another son box with separator
*     if needed. If the format is not list align, rebuilds the father box
```

```
*     RefreshViews does.
*
* EXCEPTIONS:
*   -
*
* SEE ALSO:
*
*==========================================================================*)

(*==========================================================================*
*                FUNCTION:  RefreshViewsByDeletedComponent
*--------------------------------------------------------------------------*)

PROCEDURE RefreshViewsByDeletedComponent(
     (* In *) father: Object.ObjectId;
     (* In *) position: CARDINAL
);

(**
* ARGUMENTS:
*  father: The (composed) and modified object with a visual representation
*  position: The position of component deleted
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - The object father must exist
*
* EFFECT:
*   - If the object father is composed, and it is viewed via a list align
*     format, rebuild the view by deleting  the desired
*     component as a son box, and separator box if exists. If it is not
*     a list align format, rebuilds the father box as RefreshViews does.
*
* EXCEPTIONS:
*   -
*
* SEE ALSO:
*
*==========================================================================*)

(*==========================================================================*
*                FUNCTION:  RefreshViewsByReplacedComponent
*--------------------------------------------------------------------------*)

PROCEDURE RefreshViewsByReplacedComponent(
     (* In *) father: Object.ObjectId;
     (* In *) position: CARDINAL
);

(**
```

```
* ARGUMENTS:
*  father: The (composed) and modified object with a visual representation
*  position: The position of component replaced
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - The object father must exist
*
* EFFECT:
*   - If the object father is composed, and it is viewed via a list align
*     format, rebuild the view by replacing the desired
*     component as a son box. If it is not a list align format, rebuilds
*     the father box as RefreshViews does.
*
* EXCEPTIONS:
*   -
*
* SEE ALSO:
*
*============================================================================*)

(*============================================================================*
*                          FUNCTION:  GetBoxAtXY
*----------------------------------------------------------------------------*)

PROCEDURE GetBoxAtXY(
     (* In *) view: VisViewL.TypeViewId;
     (* In *) posX, posY: INTEGER;
     (* Out *) VAR box: VisBoxL.TypeBoxId
);

(**
* ARGUMENTS:
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*
* REQUIRES:
*   - The view must exist
*
* EFFECT:
*   - Obtains the deepest box located at the X,Y position, relative to
*     the window.
*
* EXCEPTIONS:
*   -
*
* SEE ALSO:
*
```

```
*==========================================================================*)

(*==========================================================================*
*                         FUNCTION:  InsertBoxInList
*--------------------------------------------------------------------------*)

PROCEDURE InsertBoxInList(
     (* In *) box: VisBoxL.TypeBoxId
);

(**
* ARGUMENTS:
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*   -
*
* REQUIRES:
*   - The attributes of the box must be specified
*
* EFFECT:
*   - Inserts the box into internal list of boxes associated to its object
*
* EXCEPTIONS:
*   -
*
* SEE ALSO:
*   This function is for internal purposes of this Module (related with
*   controlling modification of viewed objects). See VisView.mod for
*   details.
*
* **NOTE**: These functions (Insert and DeleteBoxFromList) must be defined
*  here and not in Object, because of circular import problems of Modula-2.
*  The purpose is to avoid the need of TypeBox descriptor in Object or other
*  low level modules.
*
*==========================================================================*)

PROCEDURE DeleteBoxFromList(
     (* In *) box: VisBoxL.TypeBoxId
);

(**
* ARGUMENTS:
*
* GLOBAL VARIABLES (USED):
*
* GLOBAL VARIABLES (MODIFIED):
*   -
*
* REQUIRES:
*   - The attributes of the box must be specified
```

```
*
* EFFECT:
*   - Delete box from list of boxes associated to its object
*
* EXCEPTIONS:
*   -
*
* SEE ALSO:
*   This function is for internal purposes of this Module (related with
*   controlling modification of viewed objects). See VisView.mod for
*   details.
*
*=============================================================================*)

END VisView.
```

7.2 Módulos de implementación

7.2.1 Módulo VisBoxL.mod

```
(* $Id:$ *)
(******************************************************************************
*                             PROJECT  Lope
*=============================================================================*
*
* SUBSYSTEM:  Visual     Visualisation Subsystem
* MODULE:  Box          Box manager
* FILE:    VisBoxL.mod
* FILE TYPE:         ( ) Definition       ( ) Foreign definition
*                    (x) Implementation   ( ) Main program
*
* HISTORY:
*    DATE          AUTHOR                  DESCRIPTION
*   ---------    -------------    ----------------------------------------
*   14-Apr-97    Juan J. Amor     Initial version (file creation)
*
* EXPORTED FUNCTIONS:
*   Create, Delete, GetProperties, SetProperties
*
* DESCRIPTION:
*   Low level box object handler
*
* SEE ALSO:
*   -
*
*****************************************************************************)
```

```
IMPLEMENTATION MODULE VisBoxL;

IMPORT SYSTEM;

IMPORT Object;
IMPORT LibErr;
IMPORT LibStr;
IMPORT LibSys;

IMPORT VT;

TYPE TypeBoxPtr = POINTER TO TypeBox;

TYPE TypeBoxSet = ARRAY [1..MaxBoxId] OF TypeBoxPtr;

VAR  boxSetPtr: POINTER TO TypeBoxSet;

     currentMaxBoxId: TypeBoxId;

(*=============================================================================*
*                  PUBLIC FUNCTION: Create
*-----------------------------------------------------------------------------*)

PROCEDURE Create(
     (* Out *) VAR box: TypeBoxId
);

VAR done: BOOLEAN;
BEGIN

  (* Try to allocate an identifier to this box *)

  done := FALSE;

  INC(currentMaxBoxId);

  IF currentMaxBoxId > MaxBoxId THEN

     box := 1;

     WHILE (boxSetPtr^[box] <> NIL) AND (box <= MaxBoxId) DO
       INC(box);
     END;

     IF box > MaxBoxId THEN
        done := FALSE;
     ELSE
        done := TRUE;
     END;

  ELSE
    box := currentMaxBoxId;
    done := TRUE;
```

```
  END;

  LibSys.MemoryGet(boxSetPtr^[box],SIZE(TypeBox),done);

  IF NOT done THEN
     (* done = FALSE iff there is not any box id free or
        can not allocate the requested size of bytes *)
     LibErr.StoreError("VisBoxL.Create",1,"not enough memory");

     box := NullBoxId;

  ELSE

    (* Initialize the box properties *)
    WITH boxSetPtr^[box]^ DO
      xPos := 0; yPos := 0;
      width := 0; height := 0;
      mode := VT.Normal;
      isTerminal := TRUE;
      LibStr.Assign("",text);
      numberOfSons := 0;
      index := 0;
      fatherRef := NullBoxId;
      viewRef := Object.NullObjId;
      formatRef := Object.NullObjId;
      objectRef := Object.NullObjId;
      prevBox := NullBoxId;
      nextBox := NullBoxId;
      sameObjectLink := box;
    END;

  END;

END Create;

(*==========================================================================*
*                  PUBLIC FUNCTION: Delete
*--------------------------------------------------------------------------*)

PROCEDURE Delete(
     (* InOut *) VAR box: TypeBoxId
);

VAR

BEGIN

  IF (box = NullBoxId) OR
     (box > currentMaxBoxId) OR (boxSetPtr^[box] = NIL) THEN
    (* box is greater than current max box identifier or max box id,
       or its slot is not occupied *)
    LibErr.StoreError("VisBoxL.Delete",1,"given box does not exist");
```

```
  ELSE

    LibSys.MemoryFree(boxSetPtr^[box],SIZE(TypeBox));

    boxSetPtr^[box] := NIL;

    IF box = currentMaxBoxId THEN DEC(currentMaxBoxId); END;

    (* Return the Null identifier *)
    box := NullBoxId;

  END;

END Delete;

(*=============================================================================*
*                  PUBLIC FUNCTION: GetProperties
*-----------------------------------------------------------------------------*)

PROCEDURE GetProperties(
     (* In *) box: TypeBoxId;
     (* Out *) VAR boxProperties: TypeBox
);

BEGIN

  IF (box = NullBoxId) OR
     (box > currentMaxBoxId) OR (boxSetPtr^[box] = NIL) THEN
    (* box is greater than current max box identifier or max box id,
       or its slot is not occupied *)
    LibErr.StoreError("VisBoxL.GetProperties",1,"given box does not exist");

  ELSE
    boxProperties := boxSetPtr^[box]^;
  END;

END GetProperties;

(*=============================================================================*
*                         FUNCTION:  SetProperties
*-----------------------------------------------------------------------------*)

PROCEDURE SetProperties(
     (* In *) box: TypeBoxId;
     (* In *) boxProperties: TypeBox
);

BEGIN

  IF (box = NullBoxId) OR
     (box > currentMaxBoxId) OR (boxSetPtr^[box] = NIL) THEN
    (* box is greater than current max box identifier or max box id,
       or its slot is not occupied *)
```

```
    LibErr.StoreError("VisBoxL.SetProperties",1,"given box does not exist");

  ELSE
    boxSetPtr^[box]^ := boxProperties;
  END;

END SetProperties;

VAR ok: BOOLEAN;
    c: TypeBoxId;
BEGIN

  (* Try to get memory for our box set *)
  LibSys.MemoryGet( boxSetPtr, SIZE(TypeBoxSet), ok );

  IF NOT ok THEN
     LibErr.StoreError("VisBoxL.main",1,"Not enough memory to store box set");
  END;

  (* Initialize the box set *)
  currentMaxBoxId := 0;

  FOR c := 1 TO MaxBoxId DO
    boxSetPtr^[c] := NIL;
  END;

END VisBoxL.
```

7.2.2 Módulo `VisViewL.mod`

```
(* $Id:$ *)
(******************************************************************************
*                            PROJECT  Lope
*============================================================================*
*
* SUBSYSTEM:  Visual     Visualisation Subsystem
* MODULE:  View          View manager
* FILE:    VisViewL.mod
* FILE TYPE:         ( ) Definition       ( ) Foreign definition
*                    (x) Implementation   ( ) Main program
*
* HISTORY:
*     DATE          AUTHOR                  DESCRIPTION
*   ---------    -------------    ---------------------------------------
*   21-May-97    Juan J. Amor     Initial version (file creation)
*
* EXPORTED FUNCTIONS:
*   Create, Delete, GetProperties, SetProperties
*
```

```
* DESCRIPTION:
*   Low level view object handler
*
* SEE ALSO:
*   -
*
*****************************************************************************)

IMPLEMENTATION MODULE VisViewL;

IMPORT Object;
IMPORT LibErr;
IMPORT LibSys;

TYPE TypeViewPtr = POINTER TO TypeView;

TYPE TypeViewSet = ARRAY [1..MaxViewId] OF TypeViewPtr;

VAR  viewSetPtr: POINTER TO TypeViewSet;
     currentMaxViewId: TypeViewId;

(*===========================================================================*
*                  PUBLIC FUNCTION: Create
*--------------------------------------------------------------------------*)

PROCEDURE Create(
     (* Out *) VAR view: TypeViewId
);

VAR done: BOOLEAN;
BEGIN

  (* Try to allocate an identifier to this box *)

  done := FALSE;

  INC(currentMaxViewId);

  IF currentMaxViewId > MaxViewId THEN

     view := 1;

     WHILE (viewSetPtr^[view] <> NIL) AND (view <= MaxViewId) DO
       INC(view);
     END;

     IF view > MaxViewId THEN
        done := FALSE;
     ELSE
        done := TRUE;
     END;

  ELSE
```

```
    view := currentMaxViewId;
    done := TRUE;
  END;

  LibSys.MemoryGet(viewSetPtr^[view],SIZE(TypeView),done);

  IF NOT done THEN
     (* done = FALSE iff there is not any box id free or
        can not allocate the requested size of bytes *)
     LibErr.StoreError("VisViewL.Create",1,"not enough memory");

     view := NullViewId;

  ELSE

    (* Initialize the view properties *)
    WITH viewSetPtr^[view]^ DO
      formatRef := Object.NullObjId;
      objectRef := Object.NullObjId;
      boxId := NullBoxId;
      oldHeight := 0;
      oldWidth := 0;
      virtualScreen := NIL;
    END;

  END;

END Create;

(*===========================================================================*
*                  PUBLIC FUNCTION: Delete
*---------------------------------------------------------------------------*)

PROCEDURE Delete(
     (* InOut *) VAR view: TypeViewId
);

BEGIN

  IF (view = NullViewId) OR (view > currentMaxViewId)
    OR (viewSetPtr^[view] = NIL) THEN
    (* view is greater than current max view identifier or max view id,
       or its slot is not occupied *)
    LibErr.StoreError("VisViewL.Delete",1,"given view does not exist");

  ELSE

    LibSys.MemoryFree(viewSetPtr^[view],SIZE(TypeView));

    viewSetPtr^[view] := NIL;

    IF view = currentMaxViewId THEN DEC(currentMaxViewId); END;
```

```
    (* Return the Null identifier *)
    view := NullViewId;

  END;

END Delete;

(*============================================================================*
*                  PUBLIC FUNCTION: GetProperties
*---------------------------------------------------------------------------*)

PROCEDURE GetProperties(
     (* In *) view: TypeViewId;
     (* Out *) VAR viewProperties: TypeView
);

BEGIN

  IF (view = NullViewId) OR
    (view > currentMaxViewId) OR (viewSetPtr^[view] = NIL) THEN
    (* view is greater than current max view identifier or max view id,
       or its slot is not occupied *)
    LibErr.StoreError("VisViewL.GetProperties",1,"given view does not exist");

  ELSE
    viewProperties := viewSetPtr^[view]^;
  END;

END GetProperties;

(*============================================================================*
*                         FUNCTION:  SetProperties
*---------------------------------------------------------------------------*)

PROCEDURE SetProperties(
     (* In *) view: TypeViewId;
     (* In *) viewProperties: TypeView
);

BEGIN

  IF (view = NullViewId) OR
    (view > currentMaxViewId) OR (viewSetPtr^[view] = NIL) THEN
    (* view is greater than current max view identifier or max view id,
       or its slot is not occupied *)
    LibErr.StoreError("VisViewL.SetProperties",1,"given view does not exist");

  ELSE
    viewSetPtr^[view]^ := viewProperties;
  END;

END SetProperties;
```

```
VAR ok: BOOLEAN;
    c: TypeViewId;

BEGIN

  (* Try to get memory for our view set *)
  LibSys.MemoryGet( viewSetPtr, SIZE(TypeViewSet), ok );

  IF NOT ok THEN
    LibErr.StoreError("VisViewL.main",1,"Not enough memory to store view set");
  END;

  (* Initialize the view set *)
  currentMaxViewId := 0;

  FOR c := 1 TO MaxViewId DO
    viewSetPtr^[c] := NIL;
  END;

END VisViewL.
```

7.2.3 Módulo VisView.mod

```
(* $Id:$ *)
(****************************************************************************
*                              PROJECT  Lope
*==========================================================================*
*
* SUBSYSTEM:  Visual     Visualisation Subsystem
* MODULE:  View         View manager
* FILE:    VisView.mod
* FILE TYPE:         ( ) Definition        ( ) Foreign definition
*                    (x) Implementation    ( ) Main program
*
* HISTORY:
*     DATE          AUTHOR                  DESCRIPTION
*   ---------    -------------    ----------------------------------------
*   17-Nov-96    Juan J. Amor     Initial Version  (file creation)
*    2-Apr-96    Juan J. Amor     Added internal lists and InsertBoxInList
*
* EXPORTED FUNCTIONS:
*   Create, Delete, GetBoxAtXY, InsertBoxInList, DeleteBoxFromList,
*   RefreshViewsByInsertedComponent, RefreshViewsByDeletedComponent,
*   RefreshViews
*
* LOCAL FUNCTIONS:
*   PutTextXY, BuildScreen, Refresh, ObtainIteSubformat, FindXYBox, RePaint,
*   ObtainAlignCode, ObtainAlignCode2, PaintStringatXY, BuildVScreen,
*   ObtainIteSeparator, SubFormatType
```

```
*
* DESCRIPTION:
*   View Manager.
*
* SEE ALSO:
*   -
*
****************************************************************************)

IMPLEMENTATION MODULE VisView;

IMPORT Object;
IMPORT ObjSymb;
IMPORT VT;
IMPORT VisBoxL;
IMPORT VisViewL;
IMPORT VisBox;
IMPORT LibErr;
IMPORT LibStr;
IMPORT LibSys;

(*=============================================================================
*  LOCAL FUNCTIONS RELATED WITH Create:
*  BuildScreen, PaintStringatXY, BuildVScreen
*---------------------------------------------------------------------------*)

(*=============================================================================
*                  LOCAL FUNCTION: BuildScreen
*)

PROCEDURE BuildScreen(
     (* In *) box: VisBoxL.TypeBoxId;
     (* In *) xPos, yPos: INTEGER;
     (* In *) panel: VT.TypeTextP
);

(**
*
* DESCRIPTION:
*   Builds the virtual text screen
*
* USED BY:
*   Create
*===========================================================================*)

VAR list: VisBoxL.TBoxListP;
    boxProp, sonBoxProp: VisBoxL.TypeBox;
    sonBox: VisBoxL.TypeBoxId;
    xAbs,yAbs: INTEGER;
BEGIN

  VisBoxL.GetProperties(box,boxProp);
  IF boxProp.isTerminal THEN
```

```
    VT.TextPWriteStr(panel, boxProp.mode,yPos,xPos, boxProp.text);
  ELSE
    xAbs := xPos;
    yAbs := yPos;
    list := boxProp.boxList.top;
    WHILE (list <> NIL) DO
      sonBox := list^.box;
      VisBoxL.GetProperties(sonBox,sonBoxProp);
      xPos := xAbs + sonBoxProp.xPos;
      yPos := yAbs + sonBoxProp.yPos; (* Convert relative positions to absolute *)
      BuildScreen(sonBox,xPos,yPos,panel);
      list := list^.next;
    END;
  END;
END BuildScreen;

(*=============================================================================
*                  LOCAL FUNCTION: PaintStringatXY
*)

PROCEDURE PaintStringAtXY(
   (* In *) text: ARRAY OF CHAR;
   (* In *) yPos, xPos: CARDINAL;
   (* In *) mode: VT.TypeMode;
   (* In *) vscreen: VisViewL.TypeVirtualScreen
);

(**
*
* DESCRIPTION:
*   Paints the given string at the given position of given array
*
* USED BY:
*   Create, BuildDiffScreen
*============================================================================*)

VAR k,c:CARDINAL;
BEGIN
(*
  FOR c := 0 TO LibStr.Length(text) DO
    vscreen^[yPos,xPos+c].value := text[c];
    vscreen^[yPos,xPos+c].mode := mode;
  END;
*)

  IF LibStr.Length(text) + xPos <= VisViewL.maxX THEN

    FOR c := 0 TO LibStr.Length(text) DO

      vscreen^[yPos,xPos+c].value := text[c];
      vscreen^[yPos,xPos+c].mode := mode;

    END;
```

```
  ELSE
    k := 0;
    FOR c := xPos TO VisViewL.maxX DO

      vscreen^[yPos,c].value := text[k];
      vscreen^[yPos,c].mode := mode;
      INC(k);

    END;

    FOR c := k TO LibStr.Length(text) DO

      vscreen^[yPos+1,c].value := text[c];
      vscreen^[yPos+1,c].mode := mode;

    END;

  END;

END PaintStringAtXY;

(*=============================================================================
*                  LOCAL FUNCTION: BuildScreen
*)

PROCEDURE BuildVScreen(
     (* In *) box: VisBoxL.TypeBoxId;
     (* In *) xPos, yPos: INTEGER;
     (* In *) vscreen: VisViewL.TypeVirtualScreen
);

(**
*
* DESCRIPTION:
*   Builds the virtual text screen
*
* USED BY:
*   Create, RePaint
*============================================================================*)

VAR list: VisBoxL.TBoxListP;
    boxProp, sonBoxProp: VisBoxL.TypeBox;
    sonBox: VisBoxL.TypeBoxId;
    xAbs,yAbs: INTEGER;
BEGIN

  VisBoxL.GetProperties(box,boxProp);
  IF boxProp.isTerminal THEN
    PaintStringAtXY(boxProp.text,yPos,xPos,boxProp.mode,vscreen);
  ELSE
    xAbs := xPos;
    yAbs := yPos;
```

```
    list := boxProp.boxList.top;
    WHILE (list <> NIL) DO
      sonBox := list^.box;
      VisBoxL.GetProperties(sonBox,sonBoxProp);
      xPos := xAbs + sonBoxProp.xPos;
      yPos := yAbs + sonBoxProp.yPos; (* Convert relative positions to absolute *)
      BuildVScreen(sonBox,xPos,yPos,vscreen);
      list := list^.next;
    END;
  END;
END BuildVScreen;

(*============================================================================
*                  LOCAL FUNCTION: RePaint
*)

PROCEDURE RePaint(
     (* In *) view: VisViewL.TypeViewId
);

(**
*
* DESCRIPTION:
*   Minimal repaint of physical screen, using internal screen images
*
* USED BY:
*   RefreshViews, RefreshViewsByInsertedComponent,
*   RefreshViewsByDeletedComponent, RefreshViewsByReplacedComponent
*
*============================================================================*)

VAR newVScreen: VisViewL.TypeVirtualScreen;
    done: BOOLEAN;
    viewProp: VisViewL.TypeView;
    boxProp: VisBoxL.TypeBox;
    y,x: INTEGER;
BEGIN

  LibSys.MemoryGet(newVScreen,SIZE(VisViewL.TypeVScreen),done);

  (* Initialize with spaces - Initially vscreen values may be null
     (0x0) and that can be a problem when VT dumps to screen *)
  FOR y := 0 TO VisViewL.maxY DO
    FOR x := 0 TO VisViewL.maxX DO
       newVScreen^[y,x].value := ' ';
       newVScreen^[y,x].mode := VT.Normal;
    END;
  END;

  IF NOT done THEN
     LibErr.StoreError("VisView.Repaint",1,"not enough memory");
     RETURN;
  END;
```

```
  VisViewL.GetProperties(view,viewProp);

  BuildVScreen(viewProp.boxId,0,0,newVScreen);

  VisBoxL.GetProperties(viewProp.boxId,boxProp);

  (* Dump to physical screen only the differences *)
  FOR y := 0 TO boxProp.height DO
    FOR x := 0 TO boxProp.width DO

(*      newVScreen^[y,x].mode := viewProp.virtualScreen^[y,x].mode; *)

      IF newVScreen^[y,x].value <> viewProp.virtualScreen^[y,x].value THEN
         VT.TextPWriteChar(viewProp.textPanel,newVScreen^[y,x].mode,
            y,x,newVScreen^[y,x].value);
      END;

    END;
  END;

  (* Clean screen garbage *)

  FOR y := 0 TO boxProp.height DO
    FOR x := boxProp.width TO viewProp.oldWidth DO
      newVScreen^[y,x].value := " ";
      newVScreen^[y,x].mode := VT.Normal;
      IF viewProp.virtualScreen^[y,x].value <> " " THEN
         VT.TextPWriteChar(viewProp.textPanel,VT.Normal,
            y,x," ");
      END;
    END;
  END;

  FOR y := boxProp.height TO viewProp.oldHeight DO
    FOR x := 0 TO boxProp.width DO
      newVScreen^[y,x].value := " ";
      newVScreen^[y,x].mode := VT.Normal;
      IF viewProp.virtualScreen^[y,x].value <> " " THEN
         VT.TextPWriteChar(viewProp.textPanel,VT.Normal,
            y,x," ");
      END;
    END;
  END;

  FOR y := boxProp.height TO viewProp.oldHeight DO
    FOR x := boxProp.width TO viewProp.oldWidth DO
      newVScreen^[y,x].value := " ";
      newVScreen^[y,x].mode := VT.Normal;
      IF viewProp.virtualScreen^[y,x].value <> " " THEN
         VT.TextPWriteChar(viewProp.textPanel,VT.Normal,
            y,x," ");
      END;
```

```
    END;
  END;

  (* Change the virtual screen: now the valid is the new *)
  LibSys.MemoryFree(viewProp.virtualScreen,SIZE(VisViewL.TypeVScreen));

  viewProp.virtualScreen := newVScreen;

  VisViewL.SetProperties(view,viewProp);

END RePaint;

(*============================================================================*
*                  PUBLIC FUNCTION: Create
*----------------------------------------------------------------------------*)

PROCEDURE Create(
     (* In *) object: Object.ObjectId;
     (* In *) format: Object.ObjectId;
     (* In *) virtWidth, virtHeight: CARDINAL;
     (* In *) physWidth, physHeight: CARDINAL;
     (* Out *) VAR textPanel: VT.TypeTextP;
     (* Out *) VAR view: VisViewL.TypeViewId
);

VAR formatName, objectName: ObjSymb.Symbol;
    frefobject, orefobject: Object.ObjectId;
    box: VisBoxL.TypeBoxId;
    viewProp: VisViewL.TypeView;
    boxProp: VisBoxL.TypeBox;
    done: BOOLEAN;

BEGIN

     (* Create and prepare the object *)

     VisViewL.Create(view);

     VisViewL.GetProperties(view,viewProp);
     viewProp.formatRef := format;
     viewProp.objectRef := object;

     LibSys.MemoryGet(viewProp.virtualScreen,SIZE(VisViewL.TypeVScreen),done);

     IF NOT done THEN
        LibErr.StoreError("VisView.Create",1,"Not enough memory");
        VisViewL.Delete(view);
        RETURN;
     END;

     VisViewL.SetProperties(view,viewProp);

     (* Compose the box *)
```

```
     VisBox.Build(object,format,view,VT.Normal,box);

     viewProp.boxId := box;

     VisBoxL.GetProperties(box,boxProp);

     (* Create the physical text panel *)
     IF (physWidth = 0) AND (physHeight = 0) THEN
       VT.TextPCreate(textPanel,virtHeight,virtWidth,
          0,0,boxProp.height,boxProp.width,VT.Normal);
     ELSE
       VT.TextPCreate(textPanel,virtHeight,virtWidth,
          0,0,physHeight,physWidth,VT.Normal);
     END;

     viewProp.textPanel := textPanel;

     VisViewL.SetProperties(view,viewProp);

     BuildVScreen(viewProp.boxId,0,0,viewProp.virtualScreen);

     (* Refresh the screen *)
     BuildScreen(viewProp.boxId,0,0,textPanel);
     VT.TextPRedrawClientArea(textPanel,VT.Normal);

END Create;

(*============================================================================*
*                  PUBLIC FUNCTION: Delete
*----------------------------------------------------------------------------*)

PROCEDURE Delete(
     (* InOut *) VAR view:VisViewL.TypeViewId
);

VAR viewProp: VisViewL.TypeView;
BEGIN

  VisViewL.GetProperties(view,viewProp);

  (* Destroy the box hierarchy *)
  VisBox.Destroy(viewProp.boxId);

  (* Destroy the associated text panel *)
  VT.TextPDelete(viewProp.textPanel);

  (* Free virtual screen memory *)
  LibSys.MemoryFree(viewProp.virtualScreen,SIZE(VisViewL.TypeVScreen));

  (* Delete view *)
  VisViewL.Delete(view);

END Delete;
```

```
(*===========================================================================
*  LOCAL FUNCTIONS RELATED WITH view refreshing functions:
*  ObtainIteSubFormat, ObtainIteSeparator, ObtainAlignCode, ObtainAlignCode2,
*  SubFormatType, RePaint
*--------------------------------------------------------------------------*)

(*=========================================================================*
*                   PUBLIC FUNCTION: RefreshViews
*-------------------------------------------------------------------------*)

PROCEDURE RefreshViews(
     (* In *) object: Object.ObjectId
);

VAR box, newBox: VisBoxL.TypeBoxId;
    viewProp: VisViewL.TypeView;
    boxProp, newBoxProp, rootProp: VisBoxL.TypeBox;
BEGIN

  Object.GetBoxList(object,box);

  WHILE (box <> VisBoxL.NullBoxId) DO

    VisBoxL.GetProperties(box,boxProp);

    (* Save the old width and height of root box before realign *)
    (* Properties used when cleaning garbage *)
    VisViewL.GetProperties(boxProp.viewRef,viewProp);
    VisBoxL.GetProperties(viewProp.boxId,rootProp);
    viewProp.oldHeight := rootProp.height;
    viewProp.oldWidth := rootProp.width;
    VisViewL.SetProperties(boxProp.viewRef,viewProp);

    (* Build (compose+align) the new box *)
    VisBox.Build(object,boxProp.formatRef,boxProp.viewRef,boxProp.mode,newBox);

    (* Link to its father *)
    VisBoxL.GetProperties(newBox,newBoxProp);
    newBoxProp.fatherRef := boxProp.fatherRef;
    newBoxProp.index := boxProp.index;
    VisBoxL.SetProperties(newBox,newBoxProp);

    (* Replace old box with new box *)
    (* if it is not the root box *)
    IF (boxProp.fatherRef <> VisBoxL.NullBoxId) THEN
      VisBox.ReplaceNthBox(newBox,boxProp.index,boxProp.fatherRef);
      VisBox.ReAlign(newBox);
    ELSE

      VisBox.Destroy(box);

      VisViewL.GetProperties(boxProp.viewRef, viewProp);
```

```
      viewProp.boxId := newBox;
      VisViewL.SetProperties(boxProp.viewRef, viewProp);

    END;

    RePaint(boxProp.viewRef);

    box := boxProp.nextBox;

  END;

END RefreshViews;

(*============================================================================*
*                   LOCAL FUNCTION: ObtainIteSubformat
*)

PROCEDURE ObtainIteSubFormat(
     (* In *) iteFormat: Object.ObjectId;
     (* Out *) VAR subFormat: Object.ObjectId
);

(**
*
* DESCRIPTION:
*   Obtains the format to apply to each component
*
* USED BY:
*   RefreshViewsByInsertedComponent, RefreshViewsByDeletedComponent
*
*============================================================================*)
VAR ftemp: Object.ObjectId;
BEGIN

   Object.GetTypedComponent(iteFormat,ObjSymb.IteFormatSymbol,ftemp);
   Object.GetTypedComponent(ftemp,ObjSymb.FormatSymbol,subFormat);

END ObtainIteSubFormat;

(*============================================================================*
*                   LOCAL FUNCTION: ObtainIteSeparator
*)

PROCEDURE ObtainIteSeparator(
     (* In *) iteFormat: Object.ObjectId;
     (* Out *) VAR subFormat: Object.ObjectId
);

(**
*
* DESCRIPTION:
*   Obtains the separator of an iteration format
*
```

```
* USED BY:
*   RefreshViewsByInsertedComponent, RefreshViewsByDeletedComponent
*
*============================================================================*)
VAR ftemp: Object.ObjectId;
BEGIN

   Object.GetTypedComponent(iteFormat,ObjSymb.IteFormatSymbol,ftemp);
   Object.GetTypedComponent(ftemp,ObjSymb.SeparatorSymbol,subFormat);

END ObtainIteSeparator;

(*============================================================================*
*                  LOCAL FUNCTION: ObtainAlignCode
*)

PROCEDURE ObtainAlignCode(
     (* In *) iteFormat: Object.ObjectId;
     (* Out *) VAR alignCode: VisBoxL.TypeAlign
);

(**
*
* DESCRIPTION:
*   Obtains the align code of an iteration format
*
* USED BY:
*   RefreshViewsByInsertedComponent, RefreshViewsByDeletedComponent
*
*============================================================================*)

VAR ftemp,alignO: Object.ObjectId;
    value: ObjSymb.Symbol;

BEGIN

  Object.GetTypedComponent(iteFormat,ObjSymb.IteFormatSymbol,ftemp);
  Object.GetNthComponent(ftemp,2,alignO);
  Object.GetValue(alignO,value);

  CASE value OF
    ObjSymb.HorCentSymbol: alignCode := VisBoxL.HorCent;
  | ObjSymb.HorUpSymbol: alignCode := VisBoxL.HorUp;
  | ObjSymb.HorDownSymbol: alignCode := VisBoxL.HorDown;

  | ObjSymb.VerCentSymbol: alignCode := VisBoxL.VerCent;
  | ObjSymb.VerLeftSymbol: alignCode := VisBoxL.VerLeft;
  | ObjSymb.VerRightSymbol: alignCode := VisBoxL.VerRight;
  END;

END ObtainAlignCode;

(*============================================================================*
```

```
*                  LOCAL FUNCTION: ObtainAlignCode2
*)

PROCEDURE ObtainAlignCode2(
     (* In *) iteFormat: Object.ObjectId;
     (* Out *) VAR alignCode: VisBoxL.TypeAlign
);

(**
*
* DESCRIPTION:
*   Obtains the second align code of an iteration format
*
* USED BY:
*   RefreshViewsByInsertedComponent, RefreshViewsByDeletedComponent
*
*============================================================================*)

VAR ftemp, alignO: Object.ObjectId;
    value: ObjSymb.Symbol;

BEGIN

  Object.GetTypedComponent(iteFormat,ObjSymb.IteFormatSymbol,ftemp);
  Object.GetNthComponent(ftemp,4,alignO);
  Object.GetValue(alignO,value);

  CASE value OF
    ObjSymb.HorCentSymbol: alignCode := VisBoxL.HorCent;
  | ObjSymb.HorUpSymbol: alignCode := VisBoxL.HorUp;
  | ObjSymb.HorDownSymbol: alignCode := VisBoxL.HorDown;

  | ObjSymb.VerCentSymbol: alignCode := VisBoxL.VerCent;
  | ObjSymb.VerLeftSymbol: alignCode := VisBoxL.VerLeft;
  | ObjSymb.VerRightSymbol: alignCode := VisBoxL.VerRight;
  END;

END ObtainAlignCode2;

(*===========================================================================*
*                  LOCAL FUNCTION: SubFormatType
*)

PROCEDURE SubFormatType(
     (* In *) format: Object.ObjectId
): ObjSymb.Symbol;

(**
*
* DESCRIPTION:
*   Obtains the type of given format
*
* USED BY:
```

```
*   RefreshViewsByInsertedComponent, RefreshViewsByDeletedComponent
*
*=============================================================================*)

VAR subFormat: Object.ObjectId;
BEGIN

   Object.GetNthComponent(format,1,subFormat);

   RETURN Object.Type(subFormat);

END SubFormatType;

(*=============================================================================*
*             PUBLIC FUNCTION: RefreshViewsByInsertedComponent
*-----------------------------------------------------------------------------*)

PROCEDURE RefreshViewsByInsertedComponent(
     (* IN *) father: Object.ObjectId;
     (* IN *) position: CARDINAL
);

VAR
    box, newBox, sepBox: VisBoxL.TypeBoxId;
    boxProp, sepBoxProp, newBoxProp, rootProp: VisBoxL.TypeBox;
    viewProp: VisViewL.TypeView;
    formatType: ObjSymb.Symbol;
    son, subFormat, separator: Object.ObjectId;
    sepText: VisBoxL.TypeString;
    alignCode1, alignCode2: VisBoxL.TypeAlign;
BEGIN

  IF NOT Object.IsAtom(father) THEN

    Object.GetBoxList(father,box);

    WHILE (box <> VisBoxL.NullBoxId) DO

      VisBoxL.GetProperties(box,boxProp);

      (* Save the old width and height of root box before realign *)
      (* Properties used when cleaning garbage *)
      VisViewL.GetProperties(boxProp.viewRef,viewProp);
      VisBoxL.GetProperties(viewProp.boxId,rootProp);
      viewProp.oldHeight := rootProp.height;
      viewProp.oldWidth := rootProp.width;
      VisViewL.SetProperties(boxProp.viewRef,viewProp);

      formatType := SubFormatType(boxProp.formatRef);

      IF ( formatType = ObjSymb.IteFormatSymbol ) THEN
      (* This operation only has special effect when the format is
         an iteration *)
```

```
ObtainIteSubFormat (boxProp.formatRef, subFormat);
Object.GetNthComponent(father,position,son);

VisBox.Build(son,subFormat,boxProp.viewRef,boxProp.mode,newBox);

(* Link the new box with its father box *)
VisBoxL.GetProperties(newBox,newBoxProp);
newBoxProp.fatherRef := box;
VisBoxL.SetProperties(newBox,newBoxProp);

(* Test separator *)
ObtainIteSeparator(boxProp.formatRef, separator);

Object.GetValue(separator, sepText);

IF LibStr.Length (sepText) <> 0 THEN

   (* When there is separator, the position is really
      2*position - 1 *)
   position := 2*position - 1 ;

   (* Build a new box with a separator text, and link with its
      father box *)
   VisBoxL.Create(sepBox);
   VisBoxL.GetProperties(sepBox,sepBoxProp);
   LibStr.Assign(sepText,sepBoxProp.text);
   sepBoxProp.fatherRef := box;
   sepBoxProp.width := LibStr.Length (sepText);
   sepBoxProp.height := 1;
   sepBoxProp.formatRef := newBoxProp.formatRef;
   VisBoxL.SetProperties(sepBox,sepBoxProp);

   (* Obtain the align codes *)
   ObtainAlignCode(boxProp.formatRef, alignCode1);
   ObtainAlignCode2(boxProp.formatRef, alignCode2);

   VisBoxL.SetProperties(box, boxProp);

   (* If the position is the *last* position, the separator must
      be inserted before of the main new box. However, if the
      position is not the last, the separator must be inserted
      after the main new box. *)

   IF boxProp.numberOfSons >= position THEN
      VisBox.InsertNthBox(newBox,position,alignCode1,box);
      VisBox.InsertNthBox(sepBox,position+1,alignCode2,box);

   ELSE
      position := boxProp.numberOfSons+1;
      VisBox.InsertNthBox(sepBox,position,alignCode2,box);
      VisBox.InsertNthBox(newBox,position+1,alignCode1,box);
```

```
        END;

      ELSE

        VisBoxL.SetProperties(box, boxProp);

        (* Obtain the align code *)
        ObtainAlignCode(boxProp.formatRef, alignCode1);

        VisBox.InsertNthBox(newBox,position,alignCode1,box);

      END;

      VisBoxL.GetProperties(box, boxProp);
      (* If our box is root box, totally align it *)
      IF (boxProp.fatherRef = VisBoxL.NullBoxId) THEN
        VisBox.Align(box);
      ELSE
        VisBox.ReAlign(box);
      END;

    ELSE

      (* Very similar to RefreshViews *)
      VisBox.Build(father, boxProp.formatRef, boxProp.viewRef,
         boxProp.mode, newBox);

      VisBoxL.GetProperties(newBox,newBoxProp);
      newBoxProp.fatherRef := boxProp.fatherRef;
      newBoxProp.index := boxProp.index;
      VisBoxL.SetProperties(newBox,newBoxProp);

      IF (boxProp.fatherRef <> VisBoxL.NullBoxId) THEN
        VisBox.ReplaceNthBox(newBox,boxProp.index,boxProp.fatherRef);
        VisBox.ReAlign(newBox);
      ELSE

    VisBox.Destroy(box);

    VisViewL.GetProperties(boxProp.viewRef, viewProp);
    viewProp.boxId := newBox;
    VisViewL.SetProperties(boxProp.viewRef, viewProp);


      END;

    END;

    RePaint(boxProp.viewRef);

    box := boxProp.nextBox;

  END; (* WHILE *)
```

```
  END;

END RefreshViewsByInsertedComponent;

(*==========================================================================*
*             PUBLIC FUNCTION: RefreshViewsByDeletedComponent
*--------------------------------------------------------------------------*)

PROCEDURE RefreshViewsByDeletedComponent(
     (* In *) father: Object.ObjectId;
     (* In *) position: CARDINAL
);

VAR newBox, box, sonBox: VisBoxL.TypeBoxId;
    viewProp: VisViewL.TypeView;
    newBoxProp, boxProp, sonBoxProp, rootProp: VisBoxL.TypeBox;
    formatType: ObjSymb.Symbol;
    son,format2, separator: Object.ObjectId;
    sepText: VisBoxL.TypeString;

BEGIN

  Object.GetBoxList(father,box);

  WHILE (box <> VisBoxL.NullBoxId) DO

    VisBoxL.GetProperties(box,boxProp);
    (* Save the old width and height of root box before realign *)
    (* Properties used when cleaning garbage *)
    VisViewL.GetProperties(boxProp.viewRef,viewProp);
    VisBoxL.GetProperties(viewProp.boxId,rootProp);
    viewProp.oldHeight := rootProp.height;
    viewProp.oldWidth := rootProp.width;
    VisViewL.SetProperties(boxProp.viewRef,viewProp);

    formatType := SubFormatType(boxProp.formatRef);

    IF (formatType = ObjSymb.IteFormatSymbol) THEN

       (* Test separator *)
       ObtainIteSeparator(boxProp.formatRef, separator);
       Object.GetValue(separator, sepText);

       IF LibStr.Length(sepText) > 0 THEN

          IF (boxProp.numberOfSons > 1) THEN

             IF position = boxProp.numberOfSons THEN
                position := 2*position - 1;
                VisBox.DeleteNthBox(position,box);
                VisBox.DeleteNthBox(position-1,box);
             ELSE
```

```
                position := 2*position - 1;
                VisBox.DeleteNthBox(position,box);
                VisBox.DeleteNthBox(position,box); (* pos+1 *)
             END;
          ELSE
             VisBox.DeleteNthBox(position,box);
          END;

       ELSE
         VisBox.DeleteNthBox(position,box);

       END;

       IF ( boxProp.fatherRef = VisBoxL.NullBoxId ) THEN
         VisBox.Align(box);
       ELSE
         VisBox.ReAlign(box);
       END;

    ELSE
         (* Very similar to RefreshViews *)
         VisBox.Build(father, boxProp.formatRef, boxProp.viewRef,
            boxProp.mode, newBox);
         VisBoxL.GetProperties(newBox,newBoxProp);
         newBoxProp.fatherRef := boxProp.fatherRef;
         newBoxProp.index := boxProp.index;
         VisBoxL.SetProperties(newBox,newBoxProp);

         IF (boxProp.fatherRef <> VisBoxL.NullBoxId) THEN
           VisBox.ReplaceNthBox(newBox,boxProp.index,boxProp.fatherRef);
           VisBox.ReAlign(newBox);
         ELSE

      VisBox.Destroy(box);

      VisViewL.GetProperties(boxProp.viewRef, viewProp);
      viewProp.boxId := newBox;
      VisViewL.SetProperties(boxProp.viewRef, viewProp);

         END;
    END;

    RePaint(boxProp.viewRef);

    box := boxProp.nextBox;

  END;

END RefreshViewsByDeletedComponent;

(*=========================================================================*
*             PUBLIC FUNCTION: RefreshViewsByReplacedComponent
*-------------------------------------------------------------------------*)
```

```
PROCEDURE RefreshViewsByReplacedComponent(
     (* IN *) father: Object.ObjectId;
     (* IN *) position: CARDINAL
);

VAR
    box, newBox: VisBoxL.TypeBoxId;
    boxProp, newBoxProp, rootProp: VisBoxL.TypeBox;
    viewProp: VisViewL.TypeView;
    formatType: ObjSymb.Symbol;
    son, subFormat, separator: Object.ObjectId;
    sepText: VisBoxL.TypeString;
    alignCode: VisBoxL.TypeAlign;

BEGIN

   IF NOT Object.IsAtom(father) THEN

     Object.GetBoxList(father,box);

     WHILE (box <> VisBoxL.NullBoxId) DO

       VisBoxL.GetProperties(box, boxProp);

       (* Save the old width and height of root box before realign *)
       (* Properties used when cleaning garbage *)
       VisViewL.GetProperties(boxProp.viewRef,viewProp);
       VisBoxL.GetProperties(viewProp.boxId,rootProp);
       viewProp.oldHeight := rootProp.height;
       viewProp.oldWidth := rootProp.width;
       VisViewL.SetProperties(boxProp.viewRef,viewProp);

       formatType := SubFormatType(boxProp.formatRef);

       IF ( formatType = ObjSymb.IteFormatSymbol ) THEN

         ObtainIteSubFormat (boxProp.formatRef, subFormat);
         Object.GetNthComponent(father,position,son);

         VisBox.Build(son,subFormat,boxProp.viewRef,boxProp.mode,newBox);

         VisBoxL.GetProperties(newBox,newBoxProp);
         newBoxProp.fatherRef := box;
         VisBoxL.SetProperties(newBox,newBoxProp);

         ObtainAlignCode(boxProp.formatRef, alignCode);

         (* Test separator *)
         ObtainIteSeparator(boxProp.formatRef, separator);

         Object.GetValue(separator, sepText);
```

```
         IF LibStr.Length (sepText) <> 0 THEN
            position := 2*position - 1;
         END;

         VisBox.ReplaceNthBox(newBox,position,box);
         VisBox.ReplaceAlignCode(alignCode,position,box);

         IF boxProp.fatherRef = VisBoxL.NullBoxId THEN
            VisBox.Align(box);
         ELSE
            VisBox.ReAlign(box);
         END;

       ELSE

         (* Very similar to RefreshViews *)
         VisBox.Build(father, boxProp.formatRef, boxProp.viewRef,
            boxProp.mode, newBox);

         VisBoxL.GetProperties(newBox,newBoxProp);
         newBoxProp.fatherRef := boxProp.fatherRef;
         newBoxProp.index := boxProp.index;
         VisBoxL.SetProperties(newBox,newBoxProp);

         IF (boxProp.fatherRef <> VisBoxL.NullBoxId) THEN
           VisBox.ReplaceNthBox(newBox,boxProp.index,boxProp.fatherRef);
           VisBox.ReAlign(newBox);
         ELSE

      VisBox.Destroy(box);

      VisViewL.GetProperties(boxProp.viewRef, viewProp);
      viewProp.boxId := newBox;
      VisViewL.SetProperties(boxProp.viewRef, viewProp);

         END;

        END;

        RePaint(boxProp.viewRef);

        box := boxProp.nextBox;

     END; (* WHILE *)

  END;

END RefreshViewsByReplacedComponent;

(*==============================================================================
*                   LOCAL FUNCTION: FindXYBox
*)
```

```
PROCEDURE FindXYBox(
     (* In *) father: VisBoxL.TypeBoxId;
     (* In *) xPos, yPos: INTEGER;
     (* Out *) VAR son: VisBoxL.TypeBoxId
);

(**
*
* DESCRIPTION:
*   Auxiliary function which recursively navegates trough boxes to
* find the (X,Y) box.
*
* USED BY:
*   GetBoxAtXY
*
*============================================================================*)

VAR found: BOOLEAN;
    auxBox: VisBoxL.TypeBoxId;
    fatherProp,boxProp : VisBoxL.TypeBox;
    list: VisBoxL.TBoxListP;
BEGIN

  VisBoxL.GetProperties(father,fatherProp);
  IF fatherProp.isTerminal OR (fatherProp.boxList.top = NIL) THEN
    son := father;
  ELSE
    found := FALSE;
    list := fatherProp.boxList.top;
    WHILE (list <> NIL) AND (NOT found) DO
      auxBox := list^.box;

      (* Check if auxBox contains the pixel (xPos,yPos) *)
      VisBoxL.GetProperties(auxBox,boxProp);
      IF (boxProp.xPos <= xPos) AND (boxProp.yPos <= yPos) AND
          (xPos < boxProp.xPos + boxProp.width)
          AND (yPos < boxProp.yPos + boxProp.height) THEN
        found := TRUE;
      END;

      list := list^.next;
    END;

    (* Compute the relative coordinates of (xPos, yPos) into the found box *)
    xPos := xPos - boxProp.xPos;
    yPos := yPos - boxProp.yPos;

    (* Search the box at (X,Y) into this new box *)
    FindXYBox(auxBox,xPos,yPos,son);

  END; (* IF *)
END FindXYBox;
```

```
(*============================================================================*
*                  PUBLIC FUNCTION: InsertBoxInList
*----------------------------------------------------------------------------*)

PROCEDURE InsertBoxInList(
     (* In *) box: VisBoxL.TypeBoxId
);

VAR objBoxList: VisBoxL.TypeBoxId;
    boxProp, nextBoxProp: VisBoxL.TypeBox;
BEGIN

 VisBoxL.GetProperties(box,boxProp);

 Object.GetBoxList(boxProp.objectRef, objBoxList);

 boxProp.nextBox := objBoxList;
 boxProp.prevBox := VisBoxL.NullBoxId;

 IF (objBoxList <> VisBoxL.NullBoxId) THEN
   VisBoxL.GetProperties(objBoxList,nextBoxProp);

   nextBoxProp.prevBox := box;

   VisBoxL.SetProperties(objBoxList,nextBoxProp);
 END;

 VisBoxL.SetProperties(box,boxProp);

 objBoxList := box;

 Object.SetBoxList(boxProp.objectRef, objBoxList);

END InsertBoxInList;

(*============================================================================*
*                  PUBLIC FUNCTION: DeleteBoxFromList
*----------------------------------------------------------------------------*)

PROCEDURE DeleteBoxFromList(
     (* In *) box: VisBoxL.TypeBoxId
);

VAR boxList: VisBoxL.TypeBoxId;
    boxProp, nextBoxProp, prevBoxProp: VisBoxL.TypeBox;
BEGIN
  VisBoxL.GetProperties(box,boxProp);

  IF ( boxProp.objectRef <> Object.NullObjId) THEN
  (* A box can be a separator box: then, the object reference may be null *)

    Object.GetBoxList(boxProp.objectRef, boxList);
```

```
    IF (boxProp.nextBox <> VisBoxL.NullBoxId) THEN

       VisBoxL.GetProperties(boxProp.nextBox,nextBoxProp);

       nextBoxProp.prevBox := boxProp.prevBox;

       VisBoxL.SetProperties(boxProp.nextBox,nextBoxProp);

    END;

    IF (boxProp.prevBox <> VisBoxL.NullBoxId) THEN

       VisBoxL.GetProperties(boxProp.prevBox,prevBoxProp);

       prevBoxProp.nextBox := boxProp.nextBox;

       VisBoxL.SetProperties(boxProp.prevBox,prevBoxProp);

    ELSE

      boxList := boxProp.nextBox;
      Object.SetBoxList(boxProp.objectRef,boxList);
    END;

  END;

END DeleteBoxFromList;

(*============================================================================*
*                   PUBLIC FUNCTION: GetBoxAtXY
*----------------------------------------------------------------------------*)

PROCEDURE GetBoxAtXY(
     (* In *) view: VisViewL.TypeViewId;
     (* In *) xPos, yPos: INTEGER;
     (* Out *) VAR box: VisBoxL.TypeBoxId
);
VAR viewProp: VisViewL.TypeView;
BEGIN
  VisViewL.GetProperties(view,viewProp);
  FindXYBox(viewProp.boxId,xPos,yPos,box);
END GetBoxAtXY;

END VisView.
```

7.2.4 Módulo `VisBox.mod`

```
(* $Id:$ *)
(*****************************************************************************
```

```
*                             PROJECT  Lope
*============================================================================*
*
* SUBSYSTEM:  Visual     Visualisation Subsystem
* MODULE:  Box          Box manager
* FILE:    VisBox.mod
* FILE TYPE:         ( ) Definition       ( ) Foreign definition
*                    (x) Implementation   ( ) Main program
*
* HISTORY:
*     DATE          AUTHOR                  DESCRIPTION
*   ---------    -------------    ----------------------------------------
*    7-Oct-96    Juan J. Amor     Initial Version  (file creation)
*   17-Nov-96    Juan J. Amor     Added Destroy procedure
*   14-Mar-97    Juan J. Amor     Added ReAlign, ReplaceNthBox procedures
*   14-Apr-97    Juan J. Amor     Added low-level box handling support
*
* EXPORTED FUNCTIONS:
*   Create, Delete, GetProperties, SetProperties, Align, Build,
*   Destroy, Display, InsertNthBox, DeleteNthBox,
*   ReplaceNthBox, ReAlign, ChangeText
*
* LOCAL FUNCTIONS:
*   BoxListCreate, BoxListInsert, BoxListDelete, BoxListEmpty, BoxListTop,
*   BoxListInsertBottom, BoxListDeleteBottom, BoxListBottom, ObtainFormat,
*   BuildHierarchy, TreatNamedFormat, ObtainDefaultFormat, TreatCompFormat,
*   TreatConvFormat, CopyStrAtPos, Int2Str, Expand, TreatAltFormat,
*   TreatSeqFormat, TreatIteFormat, GetNumber, ApplySelectors, AlignHierarchy,
*   BuildDisplay, ObtainAbsXY
*
* DESCRIPTION:
*   -
*
* SEE ALSO:
*   -
*
*****************************************************************************)

IMPLEMENTATION MODULE VisBox;

IMPORT SYSTEM;

IMPORT Object;
IMPORT ObjSymb;
IMPORT ObjLocal;
IMPORT LibErr;
IMPORT LibStr;
IMPORT LibSys;
IMPORT LibIO;
IMPORT VT;

IMPORT VisBoxL;
IMPORT VisViewL;
```

```
IMPORT VisView;

(*==============================================================================
*  LOCAL FUNCTIONS RELATED WITH Handling Box Lists:
*  BoxListCreate, BoxListInsert, BoxListDelete, BoxListInsertBottom,
*  BoxListDeleteBottom, BoxListTop, BoxListBottom, BoxListEmpty
*------------------------------------------------------------------------------*)

(*==============================================================================
*                  LOCAL FUNCTION: BoxListCreate
*)

PROCEDURE BoxListCreate(
     (* Out *) VAR boxList: VisBoxL.TypeBoxList
);

(**
*
* DESCRIPTION:
*   Creates an empty box list.
*
*==============================================================================*)

BEGIN
    boxList.top := NIL;
    boxList.bottom := NIL;
END BoxListCreate;

(*==============================================================================
*                  LOCAL FUNCTION: BoxListInsert
*)

PROCEDURE BoxListInsert(
     (* In *) box: VisBoxL.TypeBoxId;
     (* In *) alignCode: VisBoxL.TypeAlign;
     (* InOut *) VAR boxList: VisBoxL.TypeBoxList
);

(**
*
* DESCRIPTION:
*   Inserts the box into the list of boxes. THE LIST MUST BE CREATED!
*
*==============================================================================*)

VAR newElement: VisBoxL.TBoxListP;
    done:BOOLEAN;
BEGIN

  LibSys.MemoryGet(newElement,SIZE(VisBoxL.TBoxList),done);

  IF NOT done THEN
    LibErr.StoreError("VisBox.BoxListInsert",1,"Not enough memory");
```

```
  ELSE
    newElement^.box := box;
    newElement^.alignCode := alignCode;

    newElement^.prev := NIL;
    newElement^.next := boxList.top;

    IF boxList.bottom = NIL THEN (* This is the first element *)
      boxList.bottom := newElement;
    ELSE
      boxList.top^.prev := newElement;
    END;

    boxList.top := newElement;
  END;

END BoxListInsert;

(*=============================================================================
*                 LOCAL FUNCTION: BoxListInsertBottom
*)

PROCEDURE BoxListInsertBottom(
     (* In *) box: VisBoxL.TypeBoxId;
     (* In *) alignCode: VisBoxL.TypeAlign;
     (* InOut *) VAR boxList: VisBoxL.TypeBoxList
);

(**
*
* DESCRIPTION:
*   Inserts the box at bottom of list of boxes. The list MUST EXIST!
*
*============================================================================*)

VAR newElement: VisBoxL.TBoxListP;
    done: BOOLEAN;
BEGIN
  LibSys.MemoryGet(newElement,SIZE(VisBoxL.TBoxList),done);

  IF NOT done THEN
    LibErr.StoreError("VisBox.BoxListInsertBottom",1,"not enough memory");
  ELSE

    newElement^.box := box;
    newElement^.alignCode := alignCode;

    newElement^.next := NIL;
    newElement^.prev := boxList.bottom;

    IF boxList.top = NIL THEN (* This is the first element *)
      boxList.top := newElement;
    ELSE
```

```
      boxList.bottom^.next := newElement;
    END;

    boxList.bottom := newElement;

  END;

END BoxListInsertBottom;


(*==============================================================================
*                   LOCAL FUNCTION: BoxListDelete
*)

PROCEDURE BoxListDelete(
     (* InOut *) VAR boxList: VisBoxL.TypeBoxList
);

(**
*
* DESCRIPTION:
*   Deletes the top box in list. THE LIST MUST EXIST!
*
*==============================================================================*)

VAR oldElement: VisBoxL.TBoxListP;

BEGIN
  IF boxList.top = NIL THEN
    LibErr.StoreError("VisBox",1,"BoxListDelete: empty list");
  ELSE
    oldElement := boxList.top;
    IF boxList.bottom = oldElement THEN (* There is only this element into the list *)
      boxList.top := NIL;
      boxList.bottom := NIL;
    ELSE
      oldElement^.next^.prev := NIL;
      boxList.top := oldElement^.next;
    END;

    LibSys.MemoryFree(oldElement,SIZE(VisBoxL.TBoxList));

  END;
END BoxListDelete;

(*==============================================================================
*                   LOCAL FUNCTION: BoxListDeleteBottom
*)

PROCEDURE BoxListDeleteBottom(
     (* InOut *) VAR boxList: VisBoxL.TypeBoxList
);
```

```
(**
*
* DESCRIPTION:
*   Deletes the bottom box in list. THE LIST MUST EXIST!
*
*============================================================================*)

VAR oldElement: VisBoxL.TBoxListP;

BEGIN
  IF boxList.bottom = NIL THEN
    LibErr.StoreError("VisBox",2,"BoxListDeleteBottom: empty list");
  ELSE

    oldElement := boxList.bottom;
    IF boxList.top = oldElement THEN (* There is only this element into the list *)
      boxList.top := NIL;
      boxList.bottom := NIL;
    ELSE
      oldElement^.prev^.next := NIL;
      boxList.bottom := oldElement^.prev;
    END;

    LibSys.MemoryFree(oldElement,SIZE(VisBoxL.TBoxList));

  END;
END BoxListDeleteBottom;


(*============================================================================
*                  LOCAL FUNCTION: BoxListTop
*)

PROCEDURE BoxListTop(
     (* In *) boxList: VisBoxL.TypeBoxList;
     (* Out *) VAR box: VisBoxL.TypeBoxId;
     (* Out *) VAR alignCode: VisBoxL.TypeAlign
);

(**
*
* DESCRIPTION:
*   Returns the top box in list and its align code. THE LIST MUST BE CREATED!
*
*============================================================================*)

BEGIN
  IF boxList.top <> NIL THEN
    box := boxList.top^.box;
    alignCode := boxList.top^.alignCode;
  ELSE
    LibErr.StoreError("VisBox",3,"BoxListTop: Empty list");
  END;
```

```
END BoxListTop;

(*============================================================================
*                  LOCAL FUNCTION: BoxListBottom
*)

PROCEDURE BoxListBottom(
     (* In *) boxList: VisBoxL.TypeBoxList;
     (* Out *) VAR box: VisBoxL.TypeBoxId;
     (* Out *) VAR alignCode: VisBoxL.TypeAlign
);

(**
*
* DESCRIPTION:
*   Returns the bottom box in list and its align code.
*   THE LIST MUST BE CREATED!
*
*===========================================================================*)

BEGIN
  IF boxList.bottom <> NIL THEN
    box := boxList.bottom^.box;
    alignCode := boxList.bottom^.alignCode;
  ELSE
    LibErr.StoreError("VisBox",4,"BoxListBottom: Empty list");
  END;
END BoxListBottom;

(*============================================================================
*                  LOCAL FUNCTION: BoxListEmpty
*)

PROCEDURE BoxListEmpty(
     (* InOut *) VAR boxList: VisBoxL.TypeBoxList
);

(**
*
* DESCRIPTION:
*   Delete all elements in list. THE LIST MUST EXIST!
*
*===========================================================================*)

BEGIN
  WHILE boxList.top <> NIL DO
    BoxListDelete(boxList);
  END;
END BoxListEmpty;


(*===========================================================================*
*            LOCAL FUNCTIONS related with public function: 'Compose':
```

```
* BuildHierarchy, TreatNamedFormat, ObtainFormat, ObtainDefaultFormat,
* TreatCompFormat, TreatConvFormat, CopyStrAtPos, Int2Str, Expand,
* TreatAltFormat, TreatSeqFormat, TreatIteFormat, GetNumber, ApplySelectors
----------------------------------------------------------------------------*)

(*============================================================================
*                  LOCAL FUNCTION: ObtainFormat
*)

PROCEDURE ObtainFormat(
     (* In *) nameOfFormat: ObjSymb.Symbol;
     (* Out *) VAR format: Object.ObjectId
);

(**
*
* DESCRIPTION:
*   Obtains the format known as "nameOfFormat"
*
* USED BY:
*   TreatNamedFormat, ObtainDefaultFormat
*
*===========================================================================*)

VAR IdentifierObjectFormat: Object.ObjectId;
BEGIN
  (* Obtain the format directory *)
  Object.GetNamedComponent(ObjLocal.IdentifierObjectRoot,ObjSymb.Code("format"),IdentifierObjectForm

  (* Obtain the format in that directory *)
  Object.GetNamedComponent(IdentifierObjectFormat,nameOfFormat,format);
END ObtainFormat;

(*============================================================================
*                  LOCAL FUNCTION: TreatNamedFormat
*)

PROCEDURE TreatNamedFormat(
     (* In *) object: Object.ObjectId;
     (* In *) format: Object.ObjectId;
     (* In *) mode: VT.TypeMode;
     (* In *) xPos, yPos: CARDINAL;
     (* Out *) VAR box: VisBoxL.TypeBoxId
);

(**
*
* DESCRIPTION:
*   Obtains the referenced format and calls BuildHierarchy again
*
* USED BY:
*   BuildHierarchy
*
```

```
*==============================================================================*)

VAR newFormatName: ObjSymb.Symbol;
    newFormat: Object.ObjectId;
BEGIN
  (* The value of the format object is the name of the format desired *)
  Object.GetValue(format,newFormatName);

  ObtainFormat(newFormatName,newFormat);

  BuildHierarchy(object,newFormat,mode,xPos,yPos,box);
END TreatNamedFormat;

(*==============================================================================
*                  LOCAL FUNCTION: CopyStrAtPos
*)

PROCEDURE CopyStrAtPos(
     (* InOut *) VAR pos: CARDINAL;
     (* In *) source: ARRAY OF CHAR;
     (* Out *) VAR destination: ARRAY OF CHAR
);

(**
* DESCRIPTION:
*   Copies the sources string onto destination string starting at
*   position 'pos' of destination string. Also return pos pointing
*   at the end of destination string.
*
* USED BY:
*   Expand
*
*==============================================================================*)

VAR i:CARDINAL;
BEGIN
  i := 0;
  WHILE source[i] <> 0C DO
    destination[pos] := source[i];
    INC(pos);
    INC(i);
  END;
END CopyStrAtPos;

(*==============================================================================
*                  LOCAL FUNCTION: Int2Str
*)

PROCEDURE Int2Str(
     (* In *) number: INTEGER;
     (* Out *) VAR string: ARRAY OF CHAR
);
```

```
(**
* DESCRIPTION:
*   Converts the number to string.
*
* USED BY:
*   Expand, TreatAltFormat
*
*============================================================================*)

VAR i,j,k:CARDINAL;
    buffer: VisBoxL.TypeString;
BEGIN

  i := 0;

  IF number < 0 THEN
    buffer[i] := '-';
    INC(i);
    number := -number;
  END;

  REPEAT
    buffer[i] := CHR( ORD('0') + (number MOD 10) );
    INC(i);
    number := number DIV 10;
  UNTIL (number = 0) OR (i > HIGH(string));

  IF i < HIGH(string) THEN
    string[i] := 0C;
    DEC(i);
  END;

  k := 0;
  FOR j := i TO 0 BY -1 DO
    string[k] := buffer[j];
    INC(k);
  END;

END Int2Str;

(*=============================================================================
*                  LOCAL FUNCTION: Expand
*)

PROCEDURE Expand(
     (* In *) text: ARRAY OF CHAR;
     (* In *) object: Object.ObjectId;
     (* Out *) VAR textExpanded: ARRAY OF CHAR
);

(**
* DESCRIPTION:
*   Expands the attributes '%X' in text, using the object.
```

```
*
* USED BY:
*   TreatConvFormat
*
*==========================================================================*)

VAR temp: VisBoxL.TypeString;
    tempInt: INTEGER;
    tempType: ObjSymb.Symbol;
    name: ObjSymb.Symbol;
    i,j,num: CARDINAL;

BEGIN

  j := 0;
  i := 0;

  WHILE (i <= HIGH(text)) AND (text[i] # 0C) DO
    IF text[i] <> '%' THEN

      textExpanded[j] := text[i];
      INC(j);

    ELSE

      INC(i);
      CASE text[i] OF
        '%': (* Wants to output '%' ... *)
             CopyStrAtPos(j,"%",textExpanded);

      | 'v': IF (Object.ValueType(object) = ObjSymb.IntegerSymbol) OR
                (Object.ValueType(object) = ObjSymb.ReferenceSymbol) THEN
               Object.GetValue(object,tempInt);
               Int2Str(tempInt,temp);
             ELSIF (Object.ValueType(object) = ObjSymb.SymbolSymbol) THEN
               Object.GetValue(object,name);
               ObjSymb.Name(name,temp);
             ELSIF Object.IsAtom(object) THEN
               Object.GetValue(object,temp);
             ELSE
               (* If the object has not value, return a null string *)
               temp[0] := 0C;
             END;
             CopyStrAtPos(j,temp,textExpanded);
      | 'n': name := Object.Label(object);
             IF name <> ObjSymb.NullSymbol THEN
               ObjSymb.Name(name,temp);
             ELSE
               (* Return a null string if the object is not a component of
                  a directory *)
               temp[0] := 0C;
             END;
             CopyStrAtPos(j,temp,textExpanded);
```

```
      | 's': name := Object.Type(object);
             ObjSymb.Name(name,temp);
             CopyStrAtPos(j,temp,textExpanded);
      | 't': name := Object.ValueType(object);
             ObjSymb.Name(name,temp);
             CopyStrAtPos(j,temp,textExpanded);
      | '*': num := Object.Index(object);
             Int2Str(num,temp);
             CopyStrAtPos(j,temp,textExpanded);
      | '#': num := Object.NumberOfComponents(object);
             Int2Str(num,temp);
             CopyStrAtPos(j,temp,textExpanded);

      ELSE LibErr.StoreError("VisBox",5,"TreatConvFormat: Illegal conversion code");
      END;  (* CASE *)

    END;  (* IF *)

    INC(i);
  END;  (* WHILE *)

  IF j < HIGH(textExpanded) THEN
    textExpanded[j] := 0C; (* Terminate the string *)
  END;

END Expand;

(*=============================================================================
*                  LOCAL FUNCTION: TreatConvFormat
*)

PROCEDURE TreatConvFormat(
     (* In *) object: Object.ObjectId;
     (* In *) format: Object.ObjectId;
     (* In *) mode: VT.TypeMode;
     (* In *) xPos, yPos: CARDINAL;
     (* Out *) VAR box: VisBoxL.TypeBoxId
);

(**
* DESCRIPTION:
*   Expands the conversion code (text+attributes) and build the terminal box
*   with that text.
*
* USED BY:
*   BuildHierarchy
*
*============================================================================*)

VAR text: VisBoxL.TypeString;
    boxProp: VisBoxL.TypeBox;
BEGIN
```

```
  VisBoxL.GetProperties(box,boxProp);

  (* Prepare the box attributes *)
  boxProp.xPos := xPos;
  boxProp.yPos := yPos;
  boxProp.mode := mode;

  WITH boxProp DO
    height := 1;
    isTerminal := TRUE;
    numberOfSons := 0;
  END;

  VisBoxL.SetProperties(box,boxProp);

  (* As this box visualizes the object, insert in view list *)

  VisView.InsertBoxInList(box);

  (* After inserting, the box properties are modified: get them again *)
  VisBoxL.GetProperties(box,boxProp);

  (* Obtain the text to expand, and expands its attributes *)
  Object.GetValue(format,text);

  Expand(text,object,boxProp.text);

  (* Obtain the lenght of the text and assigns to width of box *)
  boxProp.width := LibStr.Length(boxProp.text);

  (* Save the box properties *)

  VisBoxL.SetProperties(box,boxProp);

END TreatConvFormat;

(*==============================================================================
*                  LOCAL FUNCTION: ObtainDefaultFormat
*)

PROCEDURE ObtainDefaultFormat(
     (* In *) object: Object.ObjectId;
     (* Out *) VAR format: Object.ObjectId
);

(**
* DESCRIPTION:
*   Obtains the default format for the object (if exist, known as the
*   name of the type of the object in the format directory; or else,
*   the universal (generic) format.
*
* USED BY:
*   TreatAltFormat, BuildHierarchy
```

```
*
*===========================================================================*)

VAR schema: ObjSymb.Symbol;
BEGIN
  schema := Object.Type(object);
  ObtainFormat(schema,format);

  IF NOT LibErr.Done THEN
    ObtainFormat(ObjSymb.GenericFormatSymbol,format);
  END;

END ObtainDefaultFormat;

(*============================================================================
*                  LOCAL FUNCTION: TreatAltFormat
*)

PROCEDURE TreatAltFormat(
     (* In *) object: Object.ObjectId;
     (* In *) format: Object.ObjectId;
     (* In *) mode: VT.TypeMode;
     (* In *) xPos, yPos: CARDINAL;
     (* Out *) VAR box: VisBoxL.TypeBoxId
);

(**
* DESCRIPTION:
*   Treatment of alternative format: selects a format from the given
*   directory using the attribute selected of the object.
*
*   If this attribute is the index or number of components (cardinal) of
*   the object, it is converted to text and used as name of format.
*
*   If the attribute is the schema, label or value type of the object,
*   the simbol which names it, is used as name of format.
*
*   If the attribute is the value, and is text, it is used as named of
*   format. If is numeric, it will be converted to text and then used
*   as name of format.
*
*   When we obtain the name of format, we'll search it in the directory
*   included. If the format does not exist, we'll try to search a default
*   format coded as third component of alternative format. If this format
*   does not exist, we will create a empty (null) box.
*
* USED BY:
*   BuildHierarchy
*
*===========================================================================*)

VAR osattr,altDir,newFormat: Object.ObjectId;
    sattr,code: ObjSymb.Symbol;
```

```
    text: VisBoxL.TypeString;
    num: CARDINAL;
    boxProp: VisBoxL.TypeBox;

BEGIN

(* Obtain the first component of the format (the attribute) and then its
   value (the symbol (name) of the attribute) *)
  Object.GetNthComponent(format,1,osattr);
  Object.GetValue(osattr,sattr);

  code := ObjSymb.NullSymbol;

  CASE sattr OF

    ObjSymb.ValueSymbol:

      (* Different actions, if it is a text or a number, or none of above
         (then, it is a composed object and, of course, an error) *)
      IF Object.ValueType(object) = ObjSymb.TextSymbol THEN
         Object.GetValue(object,text);
      ELSIF Object.ValueType(object) = ObjSymb.IntegerSymbol THEN
         Object.GetValue(object,num);
         Int2Str(num,text);
      ELSE (* Composed object: return a null string *)
           text[0] := 0C;
      END;
      code := ObjSymb.Code(text);

  | ObjSymb.LabelSymbol:

      code := Object.Label(object);

  | ObjSymb.SchemaSymbol:

      code := Object.Type(object);

  | ObjSymb.TypeSymbol:

      code := Object.ValueType(object);

  | ObjSymb.IndexSymbol:

      num := Object.Index(object);
      Int2Str(num,text);
      code := ObjSymb.Code(text);

  | ObjSymb.CardinalSymbol:

      num := Object.NumberOfComponents(object);
      Int2Str(num,text);
      code := ObjSymb.Code(text);
```

```
  ELSE LibErr.StoreError("VisBox",7,"TreatAltFormat: Invalid alternative attribute");

  END; (* CASE *)


  (* Now, use the code as name of the new format desired *)
  (* The directory of formats used is the *second* component of alt. format *)
  Object.GetNthComponent(format,2,altDir);

  (* Select the new format in that directory *)
  Object.GetNamedComponent(altDir,code,newFormat);

  IF NOT LibErr.Done THEN

    (* Get the third component of alt. format: the default format *)

    Object.GetNthComponent(format,3,newFormat);

  END;

  IF newFormat <> ObjSymb.NullSymbol THEN
    (* newFormat <> ObjSymb.NullSymbol: the default format exists...
       or there is a specific format *)
    BuildHierarchy(object,newFormat,mode,xPos,yPos,box);
  ELSE

    (* Build a null box *)
    VisBoxL.GetProperties(box,boxProp);
    boxProp.isTerminal := TRUE;
    LibStr.Assign("",boxProp.text);
    VisBoxL.SetProperties(box,boxProp);

  END;

END TreatAltFormat;

(*============================================================================
*                  LOCAL FUNCTION: GetNumber
*)

PROCEDURE GetNumber(
     (* In *) str: ARRAY OF CHAR;
     (* Out *) VAR num: CARDINAL;
     (* Out *) VAR ok: BOOLEAN
);

(**
*
* DESCRIPTION:
*   This function tries to take a number from a string, if possible. If not,
*   ok will be FALSE
*
* USED BY:
```

```
*   ApplySelectors
*
*============================================================================*)

VAR i: CARDINAL;
BEGIN
  i := 0;
  num := 0;

  IF str[i] = '+' THEN
    INC(i);
  END;

  ok := FALSE;

  WHILE ( (i <= HIGH(str)) AND (str[i] >= '0') AND (str[i] <= '9') ) DO
    ok := TRUE;
    num := num*10 + CARDINAL(ORD(str[i]) - ORD('0'));
    INC(i);
  END;

END GetNumber;

(*=============================================================================
*                  LOCAL FUNCTION: ApplySelectors
*)

PROCEDURE ApplySelectors(
     (* In *) selectors: Object.ObjectId;
     (* InOut *) VAR object: Object.ObjectId
);

(**
*
* DESCRIPTION:
*   This function goes through the selectors, and obtains the new object
*   starting the root directory
*
* USED BY:
*   TreatCompFormat
*
*============================================================================*)

VAR i:CARDINAL;
    isNum:BOOLEAN;
    num:CARDINAL;
    selector: Object.ObjectId;
    selValue: ObjSymb.Symbol;
    selName: VisBoxL.TypeString;
    auxObject: Object.ObjectId;
BEGIN

(*  LibIO.WriteString("Entrada en selectors");
```

```
  LibIO.WriteLn;
*)
  auxObject := object;

  FOR i := 1 TO Object.NumberOfComponents(selectors) DO

    Object.GetNthComponent(selectors,i,selector);
    Object.GetValue(selector,selValue);
    CASE selValue OF
      ObjSymb.ThisObjSymbol:
       IF i = 1 THEN
         object := auxObject;
       END;
    | ObjSymb.FathObjSymbol:
       object := Object.Father(object);
    ELSE (* then, the name of symbol must be a number or a label/field type *)
      ObjSymb.Name(selValue,selName);
      GetNumber(selName,num,isNum);

      IF isNum THEN
        (* The selector is a number *)
        Object.GetNthComponent(object,num,auxObject);
      ELSE
        (* The selector is a label or name of component *)
        IF Object.Type(object) = ObjSymb.DirectorySymbol THEN
          Object.GetNamedComponent(object,selValue,auxObject);
        ELSE
          Object.GetTypedComponent(object,selValue,auxObject);
        END;
      END;

      IF NOT LibErr.Done THEN
        LibErr.StoreError("VisBox",8,"TreatCompFormat: Selector select an unexistent component of th
      ELSE
        object := auxObject;
      END;

    END; (* CASE *)

  END; (* FOR *)

END ApplySelectors;

PROCEDURE TreatCompFormat(
     (* In *) object: Object.ObjectId;
     (* In *) format: Object.ObjectId;
     (* In *) mode: VT.TypeMode;
     (* In *) xPos, yPos: CARDINAL;
     (* Out *) VAR box: VisBoxL.TypeBoxId
);

(**
*
```

```
* DESCRIPTION:
*   Treat the composition format: first, applies the selectors to the object
*   and then applies the same format to the selected object.
*
* USED BY:
*   BuildHierarchy
*
*============================================================================*)

VAR selectors, newFormat: Object.ObjectId;
    boxProp: VisBoxL.TypeBox;
BEGIN
  (* Obtain selectors *)
  Object.GetTypedComponent(format,ObjSymb.SelectorsSymbol,selectors);
  (* Apply these selectors to the object *)
  ApplySelectors(selectors,object);
  (* Obtain the new format and apply to the selected object *)
  Object.GetTypedComponent(format,ObjSymb.FormatSymbol,newFormat);
  (* Prepare some components on the box *)
  VisBoxL.GetProperties(box,boxProp);
  boxProp.formatRef := newFormat;
  VisBoxL.SetProperties(box,boxProp);
  BuildHierarchy(object,newFormat,mode,xPos,yPos,box);

END TreatCompFormat;

(*=============================================================================
*                  LOCAL FUNCTION: ObtainAlignCode
*)

PROCEDURE ObtainAlignCode(
     (* In *) align: Object.ObjectId;
     (* Out *) VAR alignCode: VisBoxL.TypeAlign
);

(**
*
* DESCRIPTION:
*   Identify the align symbol, and returns a VisBoxL.TypeAlign object.
*
* USED BY:
*   TreatSeqFormat, TreatIteFormat
*
*============================================================================*)

VAR
  value: ObjSymb.Symbol;
BEGIN
  Object.GetValue(align,value);

  CASE value OF
    ObjSymb.HorCentSymbol: alignCode := VisBoxL.HorCent;
  | ObjSymb.HorUpSymbol: alignCode := VisBoxL.HorUp;
```

```
  | ObjSymb.HorDownSymbol: alignCode := VisBoxL.HorDown;

  | ObjSymb.VerCentSymbol: alignCode := VisBoxL.VerCent;
  | ObjSymb.VerLeftSymbol: alignCode := VisBoxL.VerLeft;
  | ObjSymb.VerRightSymbol: alignCode := VisBoxL.VerRight;
  END;

END ObtainAlignCode;

(*============================================================================
*                  LOCAL FUNCTION: TreatSeqFormat
*)

PROCEDURE TreatSeqFormat(
     (* In *) object: Object.ObjectId;
     (* In *) format: Object.ObjectId;
     (* In *) mode: VT.TypeMode;
     (* In *) xPos, yPos: CARDINAL;
     (* Out *) VAR box: VisBoxL.TypeBoxId
);

(**
*
* DESCRIPTION:
*   Treats the sequence format: applies each format of format list to each
*   object of object list
*
* USED BY:
*   BuildHierarchy
*
*===========================================================================*)

VAR alignList, formList, newFormat, (* newObject, *) align : Object.ObjectId;
    alignCode: VisBoxL.TypeAlign;
    newBox: VisBoxL.TypeBoxId;
    boxProp,newBoxProp: VisBoxL.TypeBox;
    x,y,i: CARDINAL;
BEGIN

  (* Obtain the format list *)
  Object.GetTypedComponent(format,ObjSymb.FormatsSymbol,formList);

  (* Obtain the align code list *)
  Object.GetTypedComponent(format,ObjSymb.SeqsFormatSymbol,alignList);

  (* Insert this box in list of views *)
  VisView.InsertBoxInList(box);

  VisBoxL.GetProperties(box,boxProp);

  (* Prepare the box list *)
  boxProp.isTerminal := FALSE;
  BoxListCreate(boxProp.boxList);
```

```
  boxProp.numberOfSons := Object.NumberOfComponents(formList);

  (* Apply each format to (* each *) the object *)
  (* Note that align codes aren't used here, they will only copied to
     the list of boxes. They will be used while Align procedure *)

  FOR i := 1 TO Object.NumberOfComponents(formList) DO
    Object.GetNthComponent(formList,i,newFormat);

    (* Prepare a new box *)
    VisBoxL.Create(newBox);

    VisBoxL.GetProperties(newBox,newBoxProp);

    (* Prepare the pointers of the new boxes *)
    newBoxProp.fatherRef := box;

    (* For each new box, the view is always the same as father's *)
    newBoxProp.viewRef := boxProp.viewRef;

    (* Set the format *)
    newBoxProp.formatRef := newFormat;

    IF i < Object.NumberOfComponents(formList) THEN
      (* Remember: For N formats, there is N-1 align codes *)
      Object.GetNthComponent(alignList,i,align);
    END;

    (* Convert the Symbol to VisBoxL.TypeAlign *)
    ObtainAlignCode(align,alignCode);

    (* Save the value of index for each box *)
    newBoxProp.index := i;

    VisBoxL.SetProperties(newBox,newBoxProp);

    BoxListInsertBottom(newBox,alignCode,boxProp.boxList);

    BuildHierarchy(object,newFormat,mode,xPos,yPos,newBox);

  END;

  VisBoxL.SetProperties(box,boxProp);

END TreatSeqFormat;

(*============================================================================
*                   LOCAL FUNCTION: TreatIteFormat
*)

PROCEDURE TreatIteFormat(
     (* In *) object: Object.ObjectId;
     (* In *) format: Object.ObjectId;
```

```
     (* In *) mode: VT.TypeMode;
     (* In *) xPos, yPos: CARDINAL;
     (* Out *) VAR box: VisBoxL.TypeBoxId
);

(**
*
* DESCRIPTION:
*   Treats the iteration (list align) format. The separator is another
*   format, coded as null object if it isn't used. In that case, the
*   second align code is also not used.
*
* USED BY:
*   BuildHierarchy
*
*==========================================================================*)

VAR
  x,y,i,boxIndex: CARDINAL;
  newBox: VisBoxL.TypeBoxId;
  textvalue: VisBoxL.TypeString;
  newFormat,sepText,newObject, align1, align2: Object.ObjectId;
  alignCode1, alignCode2: VisBoxL.TypeAlign;
  boxProp, newBoxProp: VisBoxL.TypeBox;
BEGIN

(*  IF Object.IsAtom(object) THEN
    (* The object must be composed *)
    LibErr.StoreError("VisBox",11,"TreatIteFormat: The object must be composed");
*)

  IF Object.IsAtom(object) OR (Object.NumberOfComponents(object) = 0) THEN

     VisBoxL.GetProperties(box,boxProp);
     boxProp.isTerminal := TRUE;
     LibStr.Assign("",boxProp.text);
     VisBoxL.SetProperties(box,boxProp);

  ELSE
    (* Obtain the format to apply to each component *)
    Object.GetTypedComponent(format,ObjSymb.FormatSymbol,newFormat);

    (* Obtain the separator (it's optional: can be the null object) *)
    Object.GetTypedComponent(format,ObjSymb.SeparatorSymbol,sepText);

    (* Obtain the first align code *)
    Object.GetNthComponent(format,2,align1);
    ObtainAlignCode(align1,alignCode1);

    VisBoxL.GetProperties(box,boxProp);

    (* Prepare the box list *)
    boxProp.isTerminal := FALSE;
```

```
BoxListCreate(boxProp.boxList);

Object.GetValue(sepText,textvalue);
IF LibStr.Length (textvalue) = 0 THEN
  boxProp.numberOfSons := Object.NumberOfComponents(object);

ELSE
  (* If there is a separator, the number of boxes is 2N-1 *)
  boxProp.numberOfSons := 2*Object.NumberOfComponents(object) - 1;
  (* ...and there is a second align code *)
  Object.GetNthComponent(format,4,align2);
  ObtainAlignCode(align2,alignCode2);

END;

VisBoxL.SetProperties(box,boxProp);

(* Insert this box in list of views *)
VisView.InsertBoxInList(box);

(* After inserting, the box properties are modified: get them again *)
VisBoxL.GetProperties(box,boxProp);

boxIndex := 1;

FOR i := 1 TO Object.NumberOfComponents(object) DO

  VisBoxL.Create(newBox);
  VisBoxL.GetProperties(newBox,newBoxProp);

  (* Prepare pointers *)
  newBoxProp.fatherRef := box;

  newBoxProp.viewRef := boxProp.viewRef;

  newBoxProp.formatRef := newFormat;

  (* Save the index (position) of this box into its father *)
  newBoxProp.index := boxIndex;
  INC(boxIndex);

  VisBoxL.SetProperties(newBox,newBoxProp);

  Object.GetNthComponent(object,i,newObject);

  BuildHierarchy(newObject,newFormat,mode,xPos,yPos,newBox);

  BoxListInsertBottom(newBox,alignCode1,boxProp.boxList);

  (* Treat the separator, if exist and we aren't on last component *)
    IF (LibStr.Length(textvalue) > 0) AND
     (i < Object.NumberOfComponents(object)) THEN
```

```
        (* Prepare the new box as terminal, with the text of separator *)

        VisBoxL.Create(newBox);
        VisBoxL.GetProperties(newBox,newBoxProp);

        newBoxProp.xPos := xPos;
        newBoxProp.yPos := yPos;
        newBoxProp.mode := mode;

        WITH newBoxProp DO
          height := 1;
          isTerminal := TRUE;
          fatherRef := box;
          numberOfSons := 0;

          (* Save the index (position) of this box into its father *)
          index := boxIndex;
          INC(boxIndex);

        END;

        Object.GetValue(sepText,newBoxProp.text);
        newBoxProp.width := LibStr.Length(newBoxProp.text);

        VisBoxL.SetProperties(newBox,newBoxProp);

        BoxListInsertBottom(newBox,alignCode2,boxProp.boxList);

      END;

    END;

    VisBoxL.SetProperties(box,boxProp);

  END; (* IF *)
END TreatIteFormat;

(*==============================================================================
*                   LOCAL FUNCTION: BuildHierarchy
*)

PROCEDURE BuildHierarchy(
     (* In *) object: Object.ObjectId;
     (* In *) format: Object.ObjectId;
     (* In *) mode: VT.TypeMode;
     (* In *) xPos, yPos: CARDINAL;
     (* Out *) VAR box: VisBoxL.TypeBoxId
);

(**
*
* DESCRIPTION:
*   Main function which purpose is to build the box hierarchy
```

```
*
* USED BY:
*   Compose
*
*=========================================================================*)

VAR formattype: ObjSymb.Symbol;
    selFormat: Object.ObjectId;
    boxProp: VisBoxL.TypeBox;
BEGIN

  (* First, we must test if the desired format is null, and then search
     the default format *)
  IF format = Object.NullObjId THEN
    ObtainDefaultFormat(object,format);
  END;


  (* Test if this object is a format *)
  IF Object.Type(format) <> ObjSymb.FormatSymbol THEN
     LibErr.StoreError("VisBox",12,"BuildHierarchy: Format object expected");
     RETURN;
  END;

  VisBoxL.GetProperties(box,boxProp);

  (* Prepare some components of the box *)
  boxProp.mode := mode;
  boxProp.objectRef := object;

  VisBoxL.SetProperties(box,boxProp);

  (* Obtain the component of alternative object == format *)
  Object.GetNthComponent(format,1,selFormat);

  formattype := Object.Type(selFormat);

  CASE formattype OF
    ObjSymb.NamedFormatSymbol:
      TreatNamedFormat(object,selFormat,mode,xPos,yPos,box);
  | ObjSymb.ConvFormatSymbol:
      TreatConvFormat(object,selFormat,mode,xPos,yPos,box);
  | ObjSymb.AltFormatSymbol:
      TreatAltFormat(object,selFormat,mode,xPos,yPos,box);
  | ObjSymb.CompFormatSymbol:
      TreatCompFormat(object,selFormat,mode,xPos,yPos,box);
  | ObjSymb.SeqFormatSymbol:
      TreatSeqFormat(object,selFormat,mode,xPos,yPos,box);
  | ObjSymb.IteFormatSymbol:
      TreatIteFormat(object,selFormat,mode,xPos,yPos,box);
  ELSE LibErr.StoreError("VisBox",12,"BuildHierarchy: illegal format type");
  END;
```

```
END BuildHierarchy;

(*==============================================================================*
*            LOCAL FUNCTIONS related with public function: 'Align':
* AlignHierarchy
--------------------------------------------------------------------------------*)

(*==============================================================================
*                  LOCAL FUNCTION: AlignHierarchy
*)

PROCEDURE AlignHierarchy(
     (* InOut *) VAR box: VisBoxL.TypeBoxId
);

(**
*
* DESCRIPTION:
*   Main function which purpose is to align the box hierarchy. This function
* DOES NOT use the conditional align codes yet. See also Align.txt
*
* USED BY:
*   Align
*
*==============================================================================*)

VAR i:CARDINAL;
    prevX,prevY,prevWidth,prevHeight: INTEGER;
    minX,minY,maxX,maxY: INTEGER;
    boxPtr: VisBoxL.TBoxListP;
    boxProp, sonBoxProp: VisBoxL.TypeBox;
    relAlign: VisBoxL.TypeAlign;
BEGIN
  (* If the box is terminal, it's height and width was set at
     TreatConvFormat. However, if the box is composed, its dimensions
     will be set using the son boxes and align codes *)

  VisBoxL.GetProperties(box,boxProp);

  IF NOT (boxProp.isTerminal) THEN

    (* First, we'll align all son boxes (that is, apply recursively this
       procedure to each box, to obtain their width and height *)

    boxPtr := boxProp.boxList.top;

    WHILE (boxPtr <> NIL) DO
      AlignHierarchy(boxPtr^.box);
      boxPtr := boxPtr^.next;
    END;

    (* Now, align the boxes (determine their relative (X,Y) position on this
       box *)
```

```
boxPtr := boxProp.boxList.top;

VisBoxL.GetProperties(boxPtr^.box,sonBoxProp);

(* First box: its position is always (1,1) *)
sonBoxProp.xPos := 0;
sonBoxProp.yPos := 0;

VisBoxL.SetProperties(boxPtr^.box,sonBoxProp);

(* Initialize auxiliary variables *)

prevX := 0;
prevWidth := sonBoxProp.width;
prevY := 0;
prevHeight := sonBoxProp.height;

maxX := prevWidth;
maxY := prevHeight;

minX := 0;
minY := 0;

relAlign := boxPtr^.alignCode;

boxPtr := boxPtr^.next;

(* Second box and so on: their positions are computed relative to the
   previous box and the align code *)
FOR i := 2 TO boxProp.numberOfSons DO

  VisBoxL.GetProperties(boxPtr^.box,sonBoxProp);

  CASE relAlign OF
    VisBoxL.HorUp: sonBoxProp.xPos := prevX + prevWidth;
           sonBoxProp.yPos := prevY;
  | VisBoxL.HorDown: sonBoxProp.xPos := prevX + prevWidth;
             sonBoxProp.yPos := prevY + prevHeight - sonBoxProp.height;
  | VisBoxL.HorCent: sonBoxProp.xPos := prevX + prevWidth;
             sonBoxProp.yPos := (2*prevY + prevHeight - sonBoxProp.height) DIV 2;
  | VisBoxL.VerLeft: sonBoxProp.yPos := prevY + prevHeight;
              sonBoxProp.xPos := prevX;
  | VisBoxL.VerRight: sonBoxProp.yPos := prevY + prevHeight;
               sonBoxProp.xPos := prevX + prevWidth - sonBoxProp.width;
  | VisBoxL.VerCent: sonBoxProp.yPos := prevY + prevHeight;
              sonBoxProp.xPos := (2*prevX + prevWidth - sonBoxProp.width) DIV 2;
  END; (* CASE *)

  VisBoxL.SetProperties(boxPtr^.box,sonBoxProp);

  prevX := sonBoxProp.xPos;
  prevY := sonBoxProp.yPos;
```

```
      prevHeight := sonBoxProp.height;
      prevWidth := sonBoxProp.width;

      IF prevX + prevWidth > maxX THEN
         maxX := prevX + prevWidth;
      END;

      IF prevY + prevHeight > maxY THEN
         maxY := prevY + prevHeight;
      END;

      IF prevX < minX THEN
         minX := prevX;
      END;

      IF prevY < minY THEN
         minY := prevY;
      END;

      relAlign := boxPtr^.alignCode;

      boxPtr := boxPtr^.next;

    END; (* FOR *)

    (* Now, review the list of boxes if minX or MinY are negative (i.e.
       some boxes have wrong positions *)

    IF minX < 0 THEN
       boxPtr := boxProp.boxList.top;

       WHILE boxPtr <> NIL DO

         VisBoxL.GetProperties(boxPtr^.box,sonBoxProp);
         sonBoxProp.xPos := sonBoxProp.xPos + ABS(minX);
         VisBoxL.SetProperties(boxPtr^.box,sonBoxProp);

         boxPtr := boxPtr^.next;
       END;
       maxX := maxX + ABS(minX);
    END;

    IF minY < 0 THEN
       boxPtr := boxProp.boxList.top;

       WHILE boxPtr <> NIL DO

         VisBoxL.GetProperties(boxPtr^.box,sonBoxProp);
         sonBoxProp.yPos := sonBoxProp.yPos + ABS(minY);
         VisBoxL.SetProperties(boxPtr^.box,sonBoxProp);

         boxPtr := boxPtr^.next;
       END;
```

```
      maxY := maxY + ABS(minY);
    END;

    (* Finally, determine the width and height of this box *)

    boxProp.height := maxY;
    boxProp.width := maxX;

    VisBoxL.SetProperties(box,boxProp);

  END; (* IF *)

END AlignHierarchy;

(*============================================================================*
*                           DEBUGGING FUNCTIONS                               *
*----------------------------------------------------------------------------*)

PROCEDURE DumpBox(
     (* In *) box: VisBoxL.TypeBoxId
);
(* Effect: Dumps to screen the box hierarchy *)

VAR boxPtr: VisBoxL.TBoxListP;
    boxProp: VisBoxL.TypeBox;
BEGIN

  VisBoxL.GetProperties(box,boxProp);
  LibIO.WriteString("Id: ");
  LibIO.WriteInt(box,1);
  LibIO.WriteString(" Index: ");
  LibIO.WriteInt(boxProp.index,1);
  LibIO.WriteString(" RefObj: ");
  LibIO.WriteInt(boxProp.objectRef,1);
  LibIO.WriteString(" RefFather: ");
  LibIO.WriteInt(boxProp.fatherRef,1);
  LibIO.WriteString(" Ancho: ");
  LibIO.WriteInt(boxProp.width,1);
  LibIO.WriteString(" Alto: ");
  LibIO.WriteInt(boxProp.height,1);
  LibIO.WriteString(" PosX: ");
  LibIO.WriteInt(boxProp.xPos,1);
  LibIO.WriteString(" PosY: ");
  LibIO.WriteInt(boxProp.yPos,1);
  LibIO.WriteLn;

  IF boxProp.isTerminal THEN
    LibIO.WriteString(" es caja simple: ");
    LibIO.WriteString(boxProp.text);
    LibIO.WriteLn;
    LibIO.WriteLn;
  ELSE
    LibIO.WriteString(" es caja compuesta: ");
```

```
    LibIO.WriteCard(boxProp.numberOfSons,1);
    LibIO.WriteLn;
    LibIO.WriteLn;

    boxPtr := boxProp.boxList.top;

    WHILE (boxPtr <> NIL) DO
      DumpBox(boxPtr^.box);
      boxPtr := boxPtr^.next;
    END;
  END;
END DumpBox;

PROCEDURE ReportSymbol(
     (* In *) symbol: ObjSymb.Symbol
);
VAR cad: ARRAY [0..63] OF CHAR;
BEGIN
  LibIO.WriteString("symbol: ");
  ObjSymb.Name(symbol,cad);
  LibIO.WriteString(cad);
  LibIO.WriteLn;
END ReportSymbol;

PROCEDURE ReportObject(
     (* In *) object: Object.ObjectId
);
VAR cad: ARRAY[0..63] OF CHAR;
BEGIN
  LibIO.WriteString("object type name: ");
  ObjSymb.Name(Object.Type(object),cad);
  LibIO.WriteString(cad);
  LibIO.WriteLn;
  LibIO.WriteString("object type code: ");
  LibIO.WriteInt(Object.Type(object),1);
  LibIO.WriteLn;
END ReportObject;

PROCEDURE Report(
     (* In *) cad: ARRAY OF CHAR
);
BEGIN
  LibIO.WriteString(cad);
  LibIO.WriteLn;
END Report;

(*===========================================================================*
*                          PUBLIC FUNCTIONS                                 *
*----------------------------------------------------------------------------*)

(*===========================================================================*
*                  PUBLIC FUNCTION: Align
*---------------------------------------------------------------------------*)
```

```
PROCEDURE Align(
     (* InOut *) VAR box: VisBoxL.TypeBoxId
);
VAR boxProp: VisBoxL.TypeBox;
BEGIN
  (* Relative position of root box is alwais (0,0) *)
  VisBoxL.GetProperties(box,boxProp);
  boxProp.xPos := 0;
  boxProp.yPos := 0;
  VisBoxL.SetProperties(box,boxProp);
  AlignHierarchy(box);
END Align;

(*============================================================================*
*                  PUBLIC FUNCTION: ReAlign
*----------------------------------------------------------------------------*)

PROCEDURE ReAlign(
     (* InOut *) VAR box: VisBoxL.TypeBoxId
);

VAR father: VisBoxL.TypeBoxId;
    boxPtr: VisBoxL.TBoxListP;
    boxProp, fatherProp, sonBoxProp: VisBoxL.TypeBox;
    i:CARDINAL;
    prevX,prevY,prevWidth,prevHeight: INTEGER;
    minX,minY,maxX,maxY: INTEGER;
    relAlign: VisBoxL.TypeAlign;

BEGIN

  VisBoxL.GetProperties(box,boxProp);

  IF (boxProp.fatherRef <> VisBoxL.NullBoxId) THEN

    (* Align the father box (without realign son boxes) *)

    (* This code is similar to PROCEDURE Align, without recursive calls *)

    father := boxProp.fatherRef;
    VisBoxL.GetProperties(father,fatherProp);

    boxPtr := fatherProp.boxList.top;

    VisBoxL.GetProperties(boxPtr^.box,sonBoxProp);

    sonBoxProp.xPos := 0;
    sonBoxProp.yPos := 0;

    VisBoxL.SetProperties(boxPtr^.box,sonBoxProp);

    prevX := 0;
    prevWidth := sonBoxProp.width;
```

```
prevY := 0;
prevHeight := sonBoxProp.height;

maxX := prevWidth;
maxY := prevHeight;

minX := 0;
minY := 0;

relAlign := boxPtr^.alignCode;

boxPtr := boxPtr^.next;

FOR i := 2 TO fatherProp.numberOfSons DO

  VisBoxL.GetProperties(boxPtr^.box,sonBoxProp);
  CASE relAlign OF
    VisBoxL.HorUp: sonBoxProp.xPos := prevX + prevWidth;
           sonBoxProp.yPos := prevY;
  | VisBoxL.HorDown: sonBoxProp.xPos := prevX + prevWidth;
             sonBoxProp.yPos := prevY + prevHeight - sonBoxProp.height;
  | VisBoxL.HorCent: sonBoxProp.xPos := prevX + prevWidth;
             sonBoxProp.yPos := (2*prevY + prevHeight - sonBoxProp.height) DIV 2;
  | VisBoxL.VerLeft: sonBoxProp.yPos := prevY + prevHeight;
              sonBoxProp.xPos := prevX;
  | VisBoxL.VerRight: sonBoxProp.yPos := prevY + prevHeight;
               sonBoxProp.xPos := prevX + prevWidth - sonBoxProp.width;
  | VisBoxL.VerCent: sonBoxProp.yPos := prevY + prevHeight;
              sonBoxProp.xPos := (2*prevX + prevWidth - sonBoxProp.width) DIV 2;
  END; (* CASE *)
  VisBoxL.SetProperties(boxPtr^.box,sonBoxProp);

  prevX := sonBoxProp.xPos;
  prevY := sonBoxProp.yPos;
  prevHeight := sonBoxProp.height;
  prevWidth := sonBoxProp.width;

  IF prevX + prevWidth > maxX THEN
     maxX := prevX + prevWidth;
  END;

  IF prevY + prevHeight > maxY THEN
     maxY := prevY + prevHeight;
  END;

  IF prevX < minX THEN
     minX := prevX;
  END;

  IF prevY < minY THEN
     minY := prevY;
  END;
```

```
      relAlign := boxPtr^.alignCode;

      boxPtr := boxPtr^.next;

    END; (* FOR *)

    IF minX < 0 THEN
       boxPtr := fatherProp.boxList.top;

       WHILE boxPtr <> NIL DO

         VisBoxL.GetProperties(boxPtr^.box,sonBoxProp);
         sonBoxProp.xPos := sonBoxProp.xPos + ABS(minX);
         VisBoxL.SetProperties(boxPtr^.box,sonBoxProp);

         boxPtr := boxPtr^.next;
       END;
       maxX := maxX + ABS(minX);
    END;

    IF minY < 0 THEN
       boxPtr := fatherProp.boxList.top;

       WHILE boxPtr <> NIL DO

         VisBoxL.GetProperties(boxPtr^.box,sonBoxProp);
         sonBoxProp.yPos := sonBoxProp.yPos + ABS(minY);
         VisBoxL.SetProperties(boxPtr^.box,sonBoxProp);

         boxPtr := boxPtr^.next;
       END;
       maxY := maxY + ABS(minY);
    END;

    fatherProp.height := maxY;
    fatherProp.width := maxX;

    VisBoxL.SetProperties(father,fatherProp);

    (* The son boxes of this box are now on their correct positions, and
       we know the new dimensions of father *)

    (* Now, propagate the realignation to the top *)

    ReAlign(father);

  END; (* IF *)

END ReAlign;

(*=========================================================================*
*                   PUBLIC FUNCTION: Compose
*-------------------------------------------------------------------------*)
```

```
PROCEDURE Compose(
     (* In *) object: Object.ObjectId;
     (* In *) format: Object.ObjectId;
     (* In *) view: Object.ObjectId;
     (* In *) mode: VT.TypeMode;
     (* Out *) VAR box: VisBoxL.TypeBoxId
);

VAR boxProp: VisBoxL.TypeBox;

BEGIN
     VisBoxL.Create(box);
     VisBoxL.GetProperties(box,boxProp);
     (* The root box does not have father box... *)
     boxProp.fatherRef := VisBoxL.NullBoxId;
     boxProp.viewRef := view;
     boxProp.formatRef := format;
     VisBoxL.SetProperties(box,boxProp);
     BuildHierarchy(object,format,mode,1,1,box);
END Compose;

(*===========================================================================*
*                  PUBLIC FUNCTION: Build
*---------------------------------------------------------------------------*)
PROCEDURE Build(
     (* In *) object: Object.ObjectId;
     (* In *) format: Object.ObjectId;
     (* In *) view: Object.ObjectId;
     (* In *) mode: VT.TypeMode;
     (* Out *) VAR box: VisBoxL.TypeBoxId
);

BEGIN
     Compose(object,format,view,mode,box);
     Align(box);
(*   DumpBox(box); *)
(*   HALT; *)
END Build;

(*===========================================================================*
*                  PUBLIC FUNCTION: Destroy
*---------------------------------------------------------------------------*)
PROCEDURE Destroy(
      (* InOut *) VAR box: VisBoxL.TypeBoxId
);

VAR sonBox: VisBoxL.TypeBoxId;
    boxProp: VisBoxL.TypeBox;
    alignCode: VisBoxL.TypeAlign;
BEGIN
  VisBoxL.GetProperties(box,boxProp);
  IF NOT boxProp.isTerminal THEN
     WHILE boxProp.boxList.top <> NIL DO
```

```
      BoxListTop(boxProp.boxList, sonBox, alignCode);
      BoxListDelete(boxProp.boxList);
      (* Destroy the son box *)
      Destroy(sonBox);
    END;
  END;
  (* Delete from list of objects *)
  VisView.DeleteBoxFromList(box);
  (* Free allocated memory *)
  VisBoxL.Delete(box);
END Destroy;


(*=============================================================================*
*                  LOCAL FUNCTION: BuildDisplay
*-----------------------------------------------------------------------------*)

PROCEDURE BuildDisplay(
     (* InOut *) VAR box: VisBoxL.TypeBoxId;
     (* In *) mode: VT.TypeMode
);

(**
*
* DESCRIPTION:
*   Propagate a new mode to a box hierarchy
*
****************************************************************************)

VAR ptr:VisBoxL.TBoxListP;
    boxProp: VisBoxL.TypeBox;
BEGIN
  VisBoxL.GetProperties(box,boxProp);
  boxProp.mode := mode;
  VisBoxL.SetProperties(box,boxProp);
  IF NOT boxProp.isTerminal THEN
    ptr := boxProp.boxList.top;
    WHILE (ptr <> NIL) DO
      BuildDisplay(ptr^.box,mode);
      ptr := ptr^.next;
    END;
  END;
END BuildDisplay;

(*=============================================================================*
*                  LOCAL FUNCTION: ObtainAbsXY
*-----------------------------------------------------------------------------*)

PROCEDURE ObtainAbsXY(
     (* In *) VAR box: VisBoxL.TypeBoxId;
     (* Out *) VAR x,y:INTEGER
);
```

```
(**
*
* DESCRIPTION:
*   Obtain absolute position of a box, into the root box
*
****************************************************************************)

VAR boxProp: VisBoxL.TypeBox;

BEGIN

  VisBoxL.GetProperties(box,boxProp);

  IF boxProp.fatherRef <> VisBoxL.NullBoxId THEN

    x := x+boxProp.xPos;
    y := y+boxProp.yPos;

    ObtainAbsXY(boxProp.fatherRef,x,y);

  END;

END ObtainAbsXY;

(*===========================================================================*
*                  PUBLIC FUNCTION: Display
*---------------------------------------------------------------------------*)
PROCEDURE Display(
     (* InOut *) VAR box: VisBoxL.TypeBoxId;
     (* In *) mode: VT.TypeMode
);
VAR boxProp: VisBoxL.TypeBox;
    viewProp: VisViewL.TypeView;
    x,y:INTEGER;
    absX,absY: INTEGER;
BEGIN

  (* Propagate the mode to the box hierarchy *)
  BuildDisplay(box,mode);

  (* To display the new mode, we need a VT.TextPSetMode service! *)
  VisBoxL.GetProperties(box,boxProp);

  VisViewL.GetProperties(boxProp.viewRef,viewProp);

  absX := 0;
  absY := 0;
  ObtainAbsXY(box,absX,absY);

  FOR y := absY TO absY+boxProp.height DO
    FOR x := absX TO absX+boxProp.width DO
      VT.TextPSetMode(viewProp.textPanel,mode,y,x);
    END;
```

```
  END;

END Display;

(*==========================================================================*
*                   PUBLIC FUNCTION: InsertNthBox
*--------------------------------------------------------------------------*)
PROCEDURE InsertNthBox(
     (* In *) newBox: VisBoxL.TypeBoxId;
     (* In *) index: CARDINAL;
     (* In *) alignCode: VisBoxL.TypeAlign;
     (* InOut *) VAR father: VisBoxL.TypeBoxId
);
VAR i:CARDINAL;
    done: BOOLEAN;
    newElement,ptr:VisBoxL.TBoxListP;
    fatherProp, boxProp, auxProp: VisBoxL.TypeBox;
BEGIN
(* When there is N boxes in the list, valid positions are 1..N.
   To Insert at N+1 position is insert at bottom of list of boxes.
*)

  VisBoxL.GetProperties(father,fatherProp);

  IF fatherProp.isTerminal THEN
    (* The box MUST be composed *)
    LibErr.StoreError("VisBox",13,"InsertNthBox: Can't insert boxes in terminal box");
  ELSIF index > fatherProp.numberOfSons + 1 THEN
    LibErr.StoreError("VisBox",14,"InsertNthBox: index must be less or equal than numberOfS
  ELSE
    VisBoxL.GetProperties(newBox,boxProp);
    boxProp.index := index;
    VisBoxL.SetProperties(newBox,boxProp);

    IF index = fatherProp.numberOfSons + 1 THEN
      BoxListInsertBottom(newBox,alignCode,fatherProp.boxList);

    ELSIF index = 1 THEN

      BoxListInsert(newBox,alignCode,fatherProp.boxList);
      ptr := fatherProp.boxList.top^.next;

      WHILE (ptr <> NIL) DO
        VisBoxL.GetProperties(ptr^.box,auxProp);
        INC(auxProp.index);
        VisBoxL.SetProperties(ptr^.box,auxProp);
        ptr := ptr^.next;
      END;

    ELSE
      LibSys.MemoryGet(newElement,SIZE(VisBoxL.TBoxList),done);
      IF NOT done THEN
         LibErr.StoreError("VisBox.InsertNthBox",1,"Not enough memory");
```

```
      ELSE
        newElement^.box := newBox;
        newElement^.alignCode := alignCode;
        ptr := fatherProp.boxList.top;
        IF (ptr <> NIL) THEN VisBoxL.GetProperties(ptr^.box,auxProp); END;
        WHILE (ptr <> NIL) AND (auxProp.index < index) DO
          ptr := ptr^.next;
          IF (ptr <> NIL) THEN VisBoxL.GetProperties(ptr^.box,auxProp); END;
        END;

        (* Adjust new element pointers *)
        newElement^.next := ptr;
        IF (ptr <> NIL) THEN
          newElement^.prev := ptr^.prev;
          (* Adjust next element pointers *)
          newElement^.next^.prev := newElement;
        ELSE
          newElement^.prev := NIL;
        END;

        (* Adjust previous element pointers *)
        IF newElement^.prev <> NIL THEN
           newElement^.prev^.next := newElement;
        END;

        (* Adjust index of subsequent elements *)
        ptr := newElement^.next;

        WHILE (ptr <> NIL) DO
          VisBoxL.GetProperties(ptr^.box,auxProp);
          INC(auxProp.index);
          VisBoxL.SetProperties(ptr^.box,auxProp);
          ptr := ptr^.next;
        END;
      END;
    END;
    INC(fatherProp.numberOfSons);
    VisBoxL.SetProperties(father,fatherProp);
  END;

END InsertNthBox;

(*============================================================================*
*                  PUBLIC FUNCTION: ReplaceNthBox
*----------------------------------------------------------------------------*)

PROCEDURE ReplaceNthBox(
     (* In *) newBox: VisBoxL.TypeBoxId;
     (* In *) index: CARDINAL;
     (* InOut *) VAR father: VisBoxL.TypeBoxId
);
VAR ptr: VisBoxL.TBoxListP;
    oldBox: VisBoxL.TypeBoxId;
```

```
    boxProp, fatherProp: VisBoxL.TypeBox;
BEGIN

  VisBoxL.GetProperties(father,fatherProp);

  IF fatherProp.isTerminal THEN
    LibErr.StoreError("VisBox",17,"ReplaceNthBox: father must be composed");
  ELSE
    VisBoxL.GetProperties(newBox,boxProp);
    boxProp.index := index;
    VisBoxL.SetProperties(newBox,boxProp);

    ptr := fatherProp.boxList.top;
    IF (ptr <> NIL) THEN VisBoxL.GetProperties(ptr^.box,boxProp); END;

    WHILE (ptr <> NIL) AND (boxProp.index <> index) DO
      ptr := ptr^.next;
      IF (ptr <> NIL) THEN VisBoxL.GetProperties(ptr^.box,boxProp); END;
    END;

    oldBox := ptr^.box;

    (* Replace the oldBox with new *)
    ptr^.box := newBox;

    (* Recursively destroy the old box *)
    Destroy(oldBox);

  END;

END ReplaceNthBox;

(*============================================================================*
*                  PUBLIC FUNCTION: ReplaceAlignCode
*----------------------------------------------------------------------------*)

PROCEDURE ReplaceAlignCode(
   (* In *) alignCode: VisBoxL.TypeAlign;
   (* In *) index: CARDINAL;
   (* InOut *) VAR father: VisBoxL.TypeBoxId
);


VAR fatherProp, boxProp: VisBoxL.TypeBox;
    ptr: VisBoxL.TBoxListP;
BEGIN

  VisBoxL.GetProperties(father,fatherProp);

  IF fatherProp.isTerminal THEN
     LibErr.StoreError("VisBox",18,"ReplaceAlignCode: box must be composed");
  ELSE
```

```
    ptr := fatherProp.boxList.top;

    IF ptr <> NIL THEN VisBoxL.GetProperties(ptr^.box,boxProp); END;

    WHILE ( ptr <> NIL ) AND ( boxProp.index <> index ) DO
      ptr := ptr^.next;

      IF ptr <> NIL THEN VisBoxL.GetProperties(ptr^.box,boxProp); END;
    END;

    ptr^.alignCode := alignCode;

  END;

END ReplaceAlignCode;

(*=============================================================================*
*                   PUBLIC FUNCTION: DeleteNthBox
*-----------------------------------------------------------------------------*)

PROCEDURE DeleteNthBox(
     (* In *) index: CARDINAL;
     (* InOut *) VAR father: VisBoxL.TypeBoxId
);
VAR i:CARDINAL;
    ptr,old:VisBoxL.TBoxListP;
    oldBox: VisBoxL.TypeBoxId;
    fatherProp,boxProp: VisBoxL.TypeBox;
BEGIN

  VisBoxL.GetProperties(father,fatherProp);

  IF fatherProp.isTerminal THEN
    (* The box MUST be composed *)
    LibErr.StoreError("VisBox",15,"DeleteNthBox: Can't delete boxes in terminal box");
  ELSIF fatherProp.numberOfSons < index THEN
    LibErr.StoreError("VisBox",16,"DeleteNthBox: Deleted box must be less or equal than numberOfSons

  ELSE
    ptr := fatherProp.boxList.top;

    IF (ptr <> NIL) THEN

       VisBoxL.GetProperties(ptr^.box,boxProp);

       IF boxProp.index = index THEN
       (* We are deleting the first box *)
          old := ptr;
          fatherProp.boxList.top := ptr^.next;
          IF ptr^.next <> NIL THEN
             ptr^.next^.prev := NIL
          END;
          ptr := ptr^.next;
```

```
      ELSE

        WHILE (ptr <> NIL) AND (boxProp.index < index - 1) DO
          ptr := ptr^.next;
          IF (ptr <> NIL) THEN VisBoxL.GetProperties(ptr^.box,boxProp); END;
        END;

        IF (ptr <> NIL) THEN
          old := ptr^.next;
          IF old <> NIL THEN
            ptr^.next := old^.next;
            IF (ptr^.next <> NIL) THEN
              ptr^.next^.prev := ptr;
            END;
          END;
        END;

      END;

      (* Recursively destroy all son boxes of selected box *)
      IF (old <> NIL) THEN
        Destroy(old^.box);

        LibSys.MemoryFree(old,SIZE(VisBoxL.TBoxList));

        DEC(fatherProp.numberOfSons);
        VisBoxL.SetProperties(father,fatherProp);

        (* Readjust index of subsequent boxes *)
        IF (ptr <> NIL) THEN VisBoxL.GetProperties(ptr^.box,boxProp); END;
        WHILE (ptr <> NIL) DO
          DEC(boxProp.index);
          VisBoxL.SetProperties(ptr^.box,boxProp);
          ptr := ptr^.next;
          IF (ptr <> NIL) THEN VisBoxL.GetProperties(ptr^.box,boxProp); END;
        END;


      END;

    END;

  END;

END DeleteNthBox;

(*=========================================================================*
*                   PUBLIC FUNCTION: ChangeText
*-------------------------------------------------------------------------*)
PROCEDURE ChangeText(
     (* In *) newText: VisBoxL.TypeString;
     (* InOut *) VAR box: VisBoxL.TypeBoxId
```

```
);

VAR boxProp: VisBoxL.TypeBox;

BEGIN
  VisBoxL.GetProperties(box,boxProp);
  IF NOT boxProp.isTerminal THEN
    LibErr.StoreError("VisBox",17,"ChangeText: box must be terminal");
  ELSE
    LibStr.Assign(newText,boxProp.text);
    boxProp.width := LibStr.Length(newText);
    VisBoxL.SetProperties(box,boxProp);
  END;
END ChangeText;

END VisBox.
```

Bibliografía

[1] J. M. Arana: *Gestión Dinámica de Objetos Estructurados para un Entorno de Desarrollo de Software*. Proyecto Fin de Carrera. F.I.M. 1993

[2] M. Chen, R.J. Norman: *A Framework for Integrated CASE*. IEEE Software, Marzo 1992.

[3] M. Collado: *Implementación del Entorno* LOPE. Documento Interno F.I.M. (1993)

[4] M. Collado: *Nuevo Terminal Virtual para el Prototipo 2 de* LOPE. Documento Interno F.I.M. (1996)

[5] M. Collado y D. Villén: *El Lenguaje* LOPE. Documento Interno F.I.M. (1995)

[6] S.A. Dart, R.J. Ellison, P.H. Feiler, A.N. Habermann: *Software Development Environments*. IEEE Computer, Vol 20, NO. 11, Noviembre, 1987.

[7] P. Durbán: *Terminal Virtual para un Entorno Avanzado de Desarrollo de Software*. Proyecto Fin de Carrera. F.I.M. 1993

[8] A.N. Habermann, D. Notkin: *Gandalf: Software Development Environments.* IEEE Transactions on Software Engineering. Vol SE-12, NO. 12. Diciembre, 1986.

[9] A. Fugetta: *A Classification of CASE Technology.* IEEE Computer, Vol 26, NO. 12, Diciembre, 1993.

[10] Luis E. Hernández Lorenzana: *Gestión Dinámica de Almacenamiento de Objetos para un Entorno Integrado de Desarrollo de Software.* Proyecto Fin de Carrera. F.I.M. 1992

[11] C. Luque: LOPE *Browser-Editor.* Proyecto Fin de Carrera. F.I.M. 1996

[12] Sun Microsystems Inc: *Sun Modula-2 Programmer's Guide.* Sun Microsystems Inc. 1990

[13] I. Thomas, B.A. Nejmeh: *Definition of Tool Integration for Environments.* IEEE Software, Marzo 1992.

[14] J. Welsh, J. Elder: *Introducción a Modula-2.* Prentice-Hall, 1990

[15] N. Wirth: *Programming in Modula-2.* Springter-Verlag, 1985

Índice de Materias

árbol de sintaxis abstracta, 7

abstract syntax tree, 7
Ada, 2
alineación
  códigos, 38
alineación condicional, 44–46
alineación lista, 40–41
alineación tupla, 38–39
alternativa, 36–37
AST, 7
atributo, 35

base de datos, 18–24
  integridad, 18
  jerárquica, 18
  tabla de símbolos, 21
BASIC, 6

caja, 28, 43–44
  atributos, 102
  compuesta, 43
  implementación, 99
  jerarquía, 34, 41
  operaciones, 100
  terminal, 43
cargador, 1
CASE, 4
  Westmount, 10
CDIF, 11
ciclo de vida, 3–4
  análisis, 3
  cascada, 4
  codificación, 6
  tilde no, 3
  espiral, 4
  mantenimiento, 6
compartición de datos, 11
compilador, 1
composición, 37

control de versiones, 3, 6, 8
conversión, 35–36
Cornell Program Synthesizer, 8

DFD, 46
directorio, 21

editor
  dirigido por la sintaxis, 7
ejemplo, 143–157
ejemplos, 51–95
entorno, 1
  de desarrollo de software, 3
  asociado a la metodología, 9
  basado en herramientas, 8
  cuarta generación, 5
  integración, 2
  integrado, 16
  orientado a la estructura, 7
  orientado al lenguaje, 5
  para desarrollo de sistemas, 2
  para lenguajes compilados, 2
  para lenguajes interpretados, 2
  programación, 1
entorno Lope, 15–31
entornos de desarrollo de sistemas, 2
entornos de desarrollo de software, 3
especificaciones, 33–49
esquema, 19
  codificación, 22
  de formatos, 41

format, 21
formato, 28, 33
  alineación lista, 40
  alineación tupla, 38
  alternativa, 36
  composición, 37
  conversión, 35
  ejemplos, 51
  esquemas, 41
  gráfico, 46
  nombrado, 37
  por defecto, 37, 41
  universal, 41, 62
formato documento, 74–95
formato fecha, 57–60
formato universal, 62–74
formatos de visualización, 34

gestión de configuración, 8
guarda, 27

herramienta
  integración, 10
historias de módulos, 3

implementación, 97–142
  alineación lista, 114
  alineación tupla, 112
  alternativa, 111
  composición, 110
  conversión, 107

ejemplo, 143
módulos, 97
nombrado, 110
integración, 10
del proceso, 13
en el nivel de presentación, 12
por compartición de datos, 11
por control, 12
integridad, 18

jerarquía de cajas
alineación, 115
generación, 106

lenguaje
Ada, 2
BASIC, 6
Lisp, 6
modula-2, 39
Pascal, 7
PL/1, 8
lenguaje Lope, 51
Lisp, 6
Lope, 15
base de datos, 18
metaentorno, 15
servicios, 24
subsistema de diálogo, 28
terminal virtual, 29
visualización, 28, 33
módulo
VisBox.def, 171
VisBox.mod, 214
VisBoxL.def, 163
VisBoxL.mod, 185
VisView.def, 179
VisView.mod, 193
VisViewL.def, 167
VisViewL.mod, 189
módulos de definición, 163
módulos de implementación, 185
mínima
actualización de pantalla, 120
alineación, 118
metaentorno, 11
reutilización, 5
metaesquema, 23
metodología, 9
modula-2, 39
montador, 1
Motif, 13
MS-Windows, 29

niveles de integración, 13
nombrado, 37
notación polaca inversa, 24

objeto, 18
compuesto, 19
simple, 19
objeto dinámico, 101

ObjXPort, 22
operación
  BuildHierarchy, 106
  BuildScreen, 127
  ChangeText, 124
  Create, 47, 101, 126
  Delete, 48, 101
  DeleteBoxFromList, 136
  DeleteNthBox, 121
  Destroy, 124
  Display, 47, 122
  GetBoxAtXY, 48, 135
  GetBoxList, 48
  GetProperties, 102
  InsertBoxInList, 135
  InsertNthBox, 120
  ReAlign, 120
  RefreshViews, 129
  ReplaceAlignCode, 124
  ReplaceNthBox, 122
  SetProperties, 102

panel de texto, 30, 43
  refresco, 133
Pascal, 7
PCTE, 8
pila, 24
PL/1, 8
Portable Common Tool Environment, 8
procesos, 31
programación
  entorno, 1
programming in the large, 3
programming in the many, 9
programming in the small, 1, 2
prototipado, 5

refresco de vistas, 128
repositorio, 11
reutilización, 6
RPN, 24

schema, 21
service, 21
servicios, 24–27
shell script, 9
Smalltalk, 2, 6

terminal virtual, 29–31
tipos, 18
toolkit, 8
transacción, 27

Unix, 29
Unix PWB, 8

VisBox, 104–124
  operaciones, 104
vista, 28, 34, 41–43
  atributos, 103
  operaciones, 100

refresco, 128
visualización, 28
VisView, 124–137
operaciones, 125

Westmount, 10

X-Window, 29