

Webber v2.0

*Proyecto Fin de Carrera
Ingeniería Informática*

*ETS de Ingeniería Informática
Universidad de Sevilla*

Manuel Pérez Coca

Tutores: Manuel Valencia Barrero y Diego R. López
Departamento de Tecnología Electrónica

18 de septiembre de 2001

Índice General

1	Introducción	7
2	Tecnologías utilizadas	11
2.1	HTML(Hypertext Markup Language)	11
2.1.1	Estructura de un documento	13
2.1.2	Marcas de formato	14
2.1.3	Tablas	16
2.1.4	Enlaces	17
2.1.5	Inclusión de objetos (imágenes, documentos, etc.) . . .	17
2.1.6	Hojas de estilo	19
2.1.7	Marcos	20
2.1.8	Formularios	21
2.2	XML(Extensible Markup Language)	23
2.2.1	Los metadatos	25
2.2.2	Comentarios	27
2.2.3	Estructura de un documento XML	28
2.2.4	Elementos	29
2.2.5	Atributos	29
2.2.6	Entidades de Carácter	30
2.2.7	Instrucciones de Procesamiento	30
2.2.8	Secciones CDATA	31
2.2.9	Documentos bien formados	32
2.3	DOM (Document Object Model)	32
2.4	Perl	37
2.4.1	Los tipos de datos	38
2.4.2	Las expresiones regulares	40
2.4.3	Los operadores split y join	41
2.4.4	Entrada / Salida	42
2.4.5	La definición de funciones	42
2.4.6	Paquetes, módulos y orientación a objetos	43
2.4.7	El módulo XML::DOM	44

2.4.8	Perl/Tk	46
3	El modelo Webber	47
3.1	Introducción	47
3.2	Modelo Webber	50
3.2.1	El núcleo	53
3.2.2	Variables reservadas	54
3.2.3	Procesadores	56
3.2.4	Procesadores estándar	57
4	El núcleo de Webber. Diseño	59
4.1	Introducción	59
4.2	Formato de las plantillas y elementos fuente	61
4.2.1	Instrucciones de procesamiento	69
4.2.2	Fichero de configuración de Webber	71
4.2.3	DTD para los ficheros fuente Webber	72
4.3	Modos de ejecución de Webber	74
4.4	Componentes	76
4.4.1	El cuerpo principal o Main	76
4.4.2	El Parser	80
4.4.3	El Gestor de herencia	82
4.4.4	El Planificador	85
4.4.5	Funciones que realizan el interface con los árboles DOM	87
4.4.6	El módulo DomWrapper	90
5	Interface gráfico de Webber. Diseño	93
5.1	Introducción	93
5.2	Estructura general	94
5.3	Diseño funcional	96
5.3.1	El cuerpo principal o Main	96
5.3.2	La función Parser_conf	97
5.3.3	La función CargarFichero	97
5.3.4	La función InsertaComando	98
5.3.5	La función EliminaComando	98
5.3.6	La función EditarFichero	99
5.3.7	La función CerrarSalvar	99
5.3.8	La función Cerrar	99
5.3.9	La función SalvarFichero	99
5.3.10	La función MostrarAyuda	100
5.3.11	La función About	100
5.3.12	La función EjecutaWebber	100

6	Código fuente	101
6.1	Código fuente del núcleo de Webber	101
6.1.1	Módulo Webber	101
6.1.2	Módulo DomWrapper	135
6.2	Código fuente del interface gráfica de Webber	136
7	Conclusiones	157
8	Bibliografía	159
9	Apéndice A. Manual de usuario	161
9.1	Instalación de módulos	161
9.2	Manual de usuario de Webber	162
9.2.1	Ejecutando Webber	162
9.2.2	Argumentos de Webber	163
9.3	Manual de usuario de Xwebber	166
9.3.1	Ejecución de Xwebber	166
9.3.2	Manejo del interface gráfico	167
9.3.3	Información de depuración	171

Capítulo 1

Introducción

Es de sobra conocida la importancia que está adquiriendo la influencia de Internet en nuestras vidas. En muchos hogares existe un ordenador y gran parte de ellos tiene una conexión a Internet. Se ha convertido en la enciclopedia que todo el mundo desea tener, ya que en ella podemos encontrar toda aquella información que necesitemos. Esto se debe a la facilidad con la que se distribuye la información en la web, no nos encontramos con ninguna barrera porque sabemos que aquello que pongamos en ella, estará a disposición de todo el mundo, ¿Hay algún medio de información más eficaz que éste?

Esto se ha conseguido gracias a cada una de las personas que aporta su granito de arena trabajando día a día en Internet para publicar información accesible para todo el mundo. Cualquiera que necesite obtener información instantánea en un momento determinado sólo necesita un ordenador conectado a Internet para acceder a ella. Este proyecto fin de carrera pretende estar relacionado con este tema, aportando un "granito de arena"; pero no suministrando una extensa documentación sobre un problema concreto, sino que se ha optado por desarrollar una herramienta que nos ayude a la hora de publicar información en un servidor web.

El trabajo necesario para mantener un servidor de información en Internet se ha ido haciendo más complejo a medida que la cantidad de información que debía ser ofrecida y el número de fuentes para la misma han ido creciendo. Una de las aproximaciones comunes para afrontar esta situación ha sido el uso de métodos de *deconstrucción*, cuya característica común es mantener separados (total o parcialmente) los datos que deben ser presentados de las especificaciones acerca de su representación. A la hora de generar la versión final del contenido se emplean procedimientos para transformar los datos de acuerdo a sus especificaciones. Aunque la mayor parte de estas soluciones

han sido desarrolladas de manera completamente específica para un entorno, o incluso un servidor determinados, este tipo de aproximación deconstructiva es la idea que subyace en soluciones que se están proponiendo recientemente, como es el uso de hojas de estilo en documentos HTML[1], o el de XML[3] en conjunción con XSLT[3].

Por otra parte, el empleo de tecnologías basadas en componentes es una alternativa que cada vez gana más aceptación para construir sistemas distribuidos y/o complejos. Un buen ejemplo de ello son modelos de desarrollo como DCOM[10] o JavaBeans[11]. Cuando se trata de trabajar con los contenidos de un servidor de información, el uso de programas simples y pequeños, cada uno de los cuales con una *visión* distinta e independiente de la información, que colaboran para producir un resultado final común se hace más atractivo a medida que crece tanto la heterogeneidad de las fuentes de información como la complejidad en la estructura de la misma.

Aquí presentamos un entorno (al que llamamos Webber) que utiliza ambos métodos. Está basado en componentes y es capaz de emplear diferentes fuentes de información para armonizar y dotar de una estructura común a los contenidos que ofrece un servidor de información. En paralelo, el modelo facilita el uso de componentes para gestionar la metainformación acerca de estos contenidos.

Webber fue concebido originalmente como un pre-procesador de HTML para dar un aspecto homogéneo a las páginas web de un servidor. A partir de ahí, ha evolucionado hasta un entorno completo capaz de emplear diferentes entidades de proceso (los componentes), haciéndolos cooperar de acuerdo con las especificaciones del usuario, para transformar un conjunto de elementos fuente en la versión final de los contenidos que ofrece el servidor, junto con la metainformación relativa a los mismos. De esta manera, las personas encargadas de escribir los contenidos pueden concentrarse en la información en sí, sin preocuparse acerca de la localización o aspecto final de la misma. Por su lado, los administradores del servidor disponen de una herramienta potente y simple de utilizar para asegurarse de que se ofrecen unos contenidos coherentes. El uso de componentes simplifica los mecanismos de procesado de los contenidos, dado que la cooperación de elementos pequeños que se comunican entre sí facilita la introducción de funcionalidades nuevas, el desarrollo de prestaciones específicas, o la reutilización de elementos ya existentes. Es más, el uso de componentes hace que el entorno sea altamente *extensible*, lo que permite la introducción de nuevas tecnologías para la producción de

contenidos o la descripción de metadatos a medida que estén disponibles.

Dado el propósito de este proyecto, también se ha pretendido que la entrada que requiere posea una sintaxis intuitiva, estructurada y fácil de entender, y emplear un lenguaje estándar para describir la información. Al final nos hemos decantado por un lenguaje que cada vez se utiliza más para estructurar la información, para describir recursos y para intercambiar datos entre aplicaciones.

En definitiva, esperamos que Webber sea de gran utilidad para todos aquellos que publican en la web páginas y páginas de información y que están llegando a convertir a Internet en la mayor y diversa base de datos existente.

En el capítulo siguiente se explica y justifica el uso de los diferentes lenguajes de programación y tecnologías utilizadas para la implementación de este proyecto. Veremos en qué ámbito del proyecto han sido utilizados e intentaremos justificar su uso. Después se explicará el modelo Webber, la filosofía en que está fundada esta herramienta, esencialmente la utilización de componentes para el desarrollo de aplicaciones. A continuación se detalla extensamente el diseño que se ha seguido para implementar este modelo. Veremos los grandes bloques que componen Webber, los cuales desglosaremos poco a poco viendo su estructura a nivel funcional. A continuación, y al máximo nivel de detalle, se adjunta el código fuente en lenguaje Perl debidamente comentado. Para terminar, como en cualquier trabajo de estas dimensiones y características se expondrán las conclusiones que se han sacado del trabajo y haremos un pequeño resumen del mismo.

Capítulo 2

Tecnologías utilizadas

A continuación veremos las tecnologías utilizadas para llevar a cabo este proyecto. Como es lógico, los ficheros de salida que genera Webber están basados en **HTML**, ya que es el language utilizado para la creación de páginas Web. **XML** ha sido el language utilizado para los ficheros plantilla y fuente de entrada a Webber debido a su gran potencia para representar información estructurada. **DOM** suministra un interface orientado a objetos para ayudar a la manipulación simple de los ficheros XML. Y por último **Perl**, que ha sido el language utilizado para la implementación del proyecto, dada la gran flexibilidad que soporta y su gran potencia para manipular cadenas de texto.

2.1 HTML(Hypertext Markup Language)

El lenguaje HTML[1], del inglés *Hypertext Markup Language* (Lenguaje de Marcación General de Hipertexto), fue desarrollado por Tim Barners-Lee junto a un grupo de científicos del Centro Europeo de Investigación Nuclear (CERN) allá por el año 1992.

Los desarrolladores se apoyaron en el Standard Generalized Markup Language (SGML)[7] que ya estaba en uso en parte del software de edición y publicación del momento. SGML fue la primera tecnología de la información estandarizada y estructurada de cierta importancia, que procedía de IBM y lo utilizaba para mantener y dar formato a sus documentos legales. Posteriormente se expandió y adaptó para ser usado en un amplio abanico de sectores como estándar ISO. Aunque SGML es extremadamente potente, es muy complejo y requiere software de gran complejidad. Debido a esta complejidad y a su elevado coste, SGML no supuso una alternativa clara al

hipertexto en los primeros días de Internet [7].

Usando como plataforma al SGML, el grupo del CERN creó lo que hoy conocemos como HTML, un lenguaje capaz de conectar documentos mediante enlaces (conocidos como *links* en inglés), de muy fácil manejo, publicación y creación de documentos; además de unificar elementos como imágenes, sonido y texto que anteriormente se tenían que distribuir por separado. Las especificaciones del lenguaje HTML original creadas por el grupo del CERN constituyen el núcleo de las especificaciones y extensiones del HTML que han ido apareciendo a lo largo del tiempo.

Como ya hemos dicho, HTML es un lenguaje estructurado pensado para crear hipertextos que serán presentados en la web. No hay que confundirlo con otros lenguajes también utilizados en la red como los de descripción de páginas (ficheros Postscript o pdf...) o los de programación (Java, Perl, C++...). HTML no es un lenguaje de presentación ni de programación, sino que está orientado a definir la estructura y la semántica del documento.

Desde su nacimiento ha sufrido cambios, dando lugar a diversas versiones: 1.0, (en 1993), 2.0 (en 1995), 3.0(en 1995), 3.2 (en 1997), 4.0 (en 1997 y revisada en 1998). La última versión está disponible en el sitio web del W3 Consortium[13].

Como aspecto novedoso de la última versión respecto a las anteriores hay que destacar que se da una reorientación del HTML hacia su función primitiva de marcado de la estructura y semántica del documento, dejando lo relacionado con el aspecto físico a las hojas de estilo[1] y desaconsejando el uso de ciertos elementos o atributos relacionados con la presentación del documento. Además, esta última versión es capaz de representar diversos alfabetos y de utilizar dispositivos diferentes a la pantalla del ordenador (por ejemplo la televisión, la impresora, un dispositivo Braille...).

El formato de un documento HTML es ASCII, por ser el más sencillo de transportar de una plataforma a otra sin problemas de lectura. Los clientes web son capaces de dotar a la información que contiene el documento del aspecto que dictan las etiquetas HTML.

El elemento básico del HTML es la marca (*tag* en inglés), que permite controlar las características de la información contenida en la página. La mayor parte de las marcas constan de un indicador de principio y de un

indicador de fin, salvo algunas muy específicas que no lo necesitan y que constan sólo de un indicador de principio. Estos indicadores van siempre incluidos entre los caracteres `<` y `>`. Todo el texto (incluidas otras marcas) entre el indicador de principio y el de fin de la marca se ve afectado por ella. El indicador de principio contiene el identificador de la marca y, en muchos casos, lleva parámetros que controlan la forma en que debe aplicarse la marca.

Una característica fundamental del HTML es su **extensibilidad**. Si un navegador recibe un documento que contiene una marca que no es capaz de procesar o que no reconoce, debe ignorarla, tratándola como si fuera texto normal. De esta manera, el lenguaje puede evolucionar con nuevas marcas sin que sea necesario cambiar todos los navegadores, dado que las páginas que usen las nuevas marcas seguirán siendo legibles.

Dado que HTML utiliza marcas con un determinado formato, esto implica que existen algunas limitaciones para escribir el texto. La más inmediata es que, dado que las marcas se delimitan mediante los símbolos `<`, `>` estos símbolos no pueden escribirse directamente. Para resolver este problema, HTML define unos códigos para poder escribir estos caracteres y otros relacionados con las marcas:

- `<` para `<`.
- `>` para `>`.
- `&` para `&`.
- `"` para `'`.

De una manera similar, existen códigos para escribir las letras específicas de distintos idiomas (los "caracteres nacionales" que salen fuera del código ASCII usado por todos los servicios Internet).

2.1.1 Estructura de un documento

Cualquier documento HTML debe estar entre las marcas `<HTML>` y `</HTML>`, que delimitan el contenido de lo que el navegador analizará y presentará.

El documento en sí está dividido en dos zonas principales: El **encabezamiento**, comprendido entre las marcas `<HEAD>` y `</HEAD>`. Y el **cuerpo**, comprendido entre las marcas `<BODY>` y `</BODY>`.

Dentro del encabezamiento hay información del documento, como su título, comprendido entre las marcas `<TITLE>` y `</TITLE>`. El título no aparece en la pantalla principal del navegador, aunque sí se emplea, por ejemplo, para identificar al documento entre los marcadores. Dentro del cuerpo está todo el contenido del documento en sí (texto, imágenes, etc.). Otra marca importante en el encabezamiento es la marca `<META>`, que permite introducir **metainformación** (información acerca de la información) relativa al documento HTML. Elementos típicos de metainformación son el autor del documento, una corta descripción de su contenido o un conjunto de palabras clave relacionados con él.

Para definir el aspecto de la presentación del documento, desde HTML 4.0 se utilizan **hojas de estilo**[1], incluidas o referenciadas dentro de las marcas `<STYLE>` y `</STYLE>`.

También puede aparecer la marca `<BASE>`, que permite definir un URL de base para toda la página. De esta manera, en el resto del documento los enlaces pueden ser parciales, es decir, relativos a este URL de base. Por ejemplo, si en la cabecera de nuestro documento usamos:

```
<BASE HREF=''http://www.dom.inio''>
```

Y dentro del mismo hacemos referencia al URL `"/pagina.html"` se entenderá que estamos refiriéndonos al URL completo `"http://www.dom.inio/pagina.html"`.

Por tanto, la estructura general de un documento HTML es:

```
<HTML>
<HEAD>
[Título, estilos, metainformacion, etc.]
</HEAD>
<BODY>
[Marcas, texto, imágenes, etc.]
</BODY>
</HTML>
```

2.1.2 Marcas de formato

En HTML los espacios, tabuladores, saltos de línea, etc. que aparecen en el texto no tienen importancia. El navegador representa el contenido de la

página en función de las marcas que contiene. Así, para separar el texto en distintos párrafos se emplea la marca <P>, y para realizar un salto de línea sin dejar la línea en blanco que va asociada a una separación entre párrafos se utiliza la marca
.

El texto puede contener cabeceras, comprendidas entre las marcas <Hx> y </Hx>, donde x indica el nivel de la cabecera y puede tomar valores entre 1 y 6. Cuanto mas pequeño sea el número más importante se asume que es la cabecera y el navegador la representará con letras de mayor tamaño y más resaltada. Otras marcas que intervienen en el formato de párrafos son:

- <PRE> y </PRE>: El texto que se encuentre dentro de ella se supone preformateado y aparece con una fuente de espaciado fijo (tipo Courier). Además dentro de esta marca sí se respetan los espacios en blanco, tabuladores y saltos de línea.
- <BLOCKQUOTE> y </BLOCKQUOTE>: Se utiliza para destacar una cita textual dentro del texto general. El resultado práctico es que los márgenes a ambos lados se ensanchan.

Los elementos básicos para destacar alguna parte del texto son las marcas:

- y : para poner el texto que encierren en negrita.
- <I> y </I>: para poner el texto que encierren en cursiva.

Para presentar el texto en forma de lista, HTML ofrece tres tipos distintos:

Las **listas desordenadas** para presentar elementos que no necesitan ir precedidos por un número. Se emplea la marca para indicar el comienzo de la lista, la marca para cada uno de los elementos de la lista y la marca para indicar el final de la lista.

Las **listas ordenadas** para presentar elementos en un orden determinado. Cada párrafo va precedido de un número correlativo. Se emplea la marca para indicar el comienzo de la lista, la marca para cada uno de los elementos de la lista y la marca para indicar el final de la lista.

Las **listas de definición** son apropiadas para glosarios o definiciones de términos. Toda la lista debe ir englobada entre las marcas `<DL>` y `</DL>`. A diferencia de las otras dos cada renglón de la lista tiene dos partes:

- El nombre del término que se define, con la marca `<DT>`.
- Y la definición, con la marca `<DD>`.

Es posible también escribir comentarios en el documento HTML, que son de utilidad para el autor pero que no son mostrados por el navegador al presentar la página. El texto de estos comentarios debe ir entre los símbolos `<!--` y `-->`.

2.1.3 Tablas

Las marcas que se emplean para confeccionar tablas en HTML son las siguientes:

- La marca general, que engloba a todas las demás es `<TABLE>`, `</TABLE>`. Admite atributos como los siguientes:

BORDER, para especificar el número de pixels que definen los bordes de los elementos de la tabla.

WIDTH y **HEIGHT**, para especificar la anchura y la altura que debe tener la tabla. Admiten como valores un número de pixels y, en el caso de **WIDTH**, una expresión que indica el tanto por ciento de la anchura de la ventana del navegador que debe ocupar la tabla (por ejemplo, **60%**).

CELLPADDING y **CELLSPACING**, para especificar el número de pixels que deben separar el contenido de las celdas de los bordes y las celdas mismas entre sí, respectivamente. Una marca opcional, para proporcionar una descripción de la tabla, que debe aparecer a continuación de la marca principal. La descripción debe aparecer entre `<CAPTION>` y `</CAPTION>`.

- En el siguiente nivel, dentro de la marca anterior, se encuentran las marcas para formar cada fila de la tabla, `<TR>` y `</TR>`. Aparecen tantas veces como filas tenga la tabla.
- En el último nivel se encuentran las marcas de cada celda, `<TD>` y `</TD>`, que engloban el contenido de cada celda concreta. Aparecen tantas veces como celdas tenga la fila en que se encuentren. Si se quiere notar que cierta celda marca una cabecera de la tabla (un título para una fila o columna), en vez de usar `<TD>` se utiliza `<TH>`.

Las tres marcas que definen el contenido de una tabla admiten los atributos **ALIGN** y **VALIGN** para definir como se alinea, dentro de las celdas, su contenido. Valores aceptables son **CENTER**, **LEFT** y **RIGHT** para **ALIGN**, y **MIDDLE**, **TOP** y **BOTTOM** para **VALIGN**.

2.1.4 Enlaces

Un enlace hipertexto en una página HTML tiene la siguiente estructura general: `<A HREF="Dst"> Cnt </A>`. Donde **Dst** es el URL de destino del enlace y **Cnt** es el contenido (texto y/o imágenes) que define el área activa del mismo, es decir, la zona de la página que, seleccionada, permitirá acceder al destino del enlace.

El destino de un enlace de hipertexto HTML es un URL[1], pero también puede usarse para hacer referencia a partes de la misma página, utilizando un destino de la forma **#Nmb**, donde **Nmb** es el nombre de un punto del documento que se ha marcado con la siguiente construcción: `<A NAME=' 'Nmb' ' ></A>`.

2.1.5 Inclusión de objetos (imágenes, documentos, etc.)

Una de las características fundamentales del HTML es su capacidad multimedia: cualquier clase de objeto puede, en principio, ser incluido en ellas: imágenes, applets(programas que son automáticamente cargados y ejecutados), sonidos, e incluso otros documentos HTML.

Existe una marca genérica para la inclusión de cualquier tipo de objeto en un documento HTML. Esta marca es `<OBJECT>`. Sus atributos permiten indicar:

- La localización de la información que contiene el objeto.
- El tipo de información que contiene, por medio de un tipo de contenido MIME.
- Cuestiones relativas a la presentación de la información, como la altura y la anchura de la misma en la página y su alineación con otros elementos del documento.

Para incluir imágenes en un documento HTML se puede (y se suele) utilizar también la marca específica `<IMG>`. Esta marca usa un atributo obligatorio

para indicar la localización de los datos de la imagen. Su estructura general es: `<IMG SRC=URLDeLaImagen>`. Además del atributo `SRC` la marca de imagen puede tener otros atributos, que son opcionales, como:

- `ALT="Descripción"`, que introduce una descripción (un texto alternativo) que describe la imagen. Si el navegador no puede mostrar la imagen, muestra el valor del atributo `ALT`, con lo que, por lo menos, la persona que está accediendo a la página puede hacerse una idea del contenido o propósito de la imagen.
- `ALIGN`, que acepta valores como `TOP`, `MIDDLE` o `BOTTOM`, permite definir la posición de la imagen con respecto al texto que la acompaña.
- `HSPACE` y `VSPACE`, que aceptan también como parámetros un número de pixels, permiten definir a qué distancia mínima deben situarse otros elementos de la página (texto u otras imágenes), bien sea en sentido horizontal (`HSPACE`) o vertical (`VSPACE`).

También es posible asociar determinadas áreas de una imagen con diferentes acciones, de forma que al activar el usuario una de estas áreas, la acción se ejecute. Esto permite, por ejemplo, la realización de mapas sensibles o de barras de navegación. Para ello se utiliza las marcas `<MAP>` y `</MAP>`. Esta marca tiene un atributo obligatorio, `NAME`, que permite asociarle un nombre. Si usamos este nombre en el atributo `USEMAP` de una marca de imagen o una marca de objeto, la imagen en cuestión se utiliza para realizar las acciones especificadas en la marca `<MAP>`.

Las acciones correspondientes a una marca `<MAP>` se especifican por medio de la marcas de área que contienen. Esta marca tiene el identificador `<AREA>` y define:

- La acción que debe realizarse cuando el usuario active el área, normalmente por medio del parámetro `HREF` (como en la marca `<A>` para enlaces).
- La forma del área, definida por el atributo `SHAPE`, que puede tomar los valores `RECT`, `CIRCLE`, `POLY` o `DEFAULT`.
- Las coordenadas y dimensiones del área, definidas por el atributo `COORDS`. El valor de este atributo es una lista de números separados por comas, y su significado y cantidad dependen de la forma del área.

2.1.6 Hojas de estilo

Hasta la versión 4.0, los elementos de lo que se llama estilo (los que controlan la presentación de la información por parte del navegador) se encontraban diseminados en la especificación de HTML y cada diseñador de páginas recurría a ciertos "trucos personales" para obtener efectos atractivos. Con esta versión, aparece un lenguaje específico para definir el estilo (llamado CSS, *Cascading Style Sheets*) y elementos, tanto marcas como atributos, para incorporarlo en un documento.

Por un lado, CSS es un lenguaje completo para la definición de la apariencia que deben tener los elementos de un documento HTML. Está basado, al igual que en el caso de las marcas HTML, en un conjunto de atributos y valores que definen las propiedades de presentación. Una definición en CSS consta de una lista de pares atributo-valor separados por punto y coma (el carácter ;). El atributo se separa del valor por medio de dos puntos (el carácter :):

```
atributo1: valor1; atributo2: valor2; ...; atributoN: valorN
```

CSS ofrece atributos para prácticamente cualquier elemento de presentación del documento, como pueden ser:

- Fuentes, tanto en cuanto a su tipo como a su tamaño.
- Colores, de los caracteres, del fondo de la página, de los enlaces, etc.
- Alineación entre los diferentes elementos del documento: texto, imágenes, tablas, etc.
- Bordes y marcos para la representación de objetos.
- Sangrados e interlineados de párrafos.
- Tipos de presentación alternativos, de manera que puedan escogerse diferentes estilos según el equipo usado para acceder la información, e incluso opciones de accesibilidad para personas discapacitadas.

Cualquier marca HTML puede ir acompañada de una especificación de estilo usando CSS por medio del atributo `STYLE`. Por ejemplo, si queremos que el tamaño de la fuente de un párrafo sea de 12 puntos y que el color del texto sea rojo podemos usar la siguiente marca:

```
<P STYLE="font-size: 12pt; color: red">
```

Como vemos **STYLE** admite como parámetro una lista de especificaciones CSS.

La verdadera potencia de las especificaciones de estilo aparece cuando se usa la marca de estilo en la cabecera del documento. Entre `<STYLE>` y `</STYLE>` pueden incluirse:

- Especificaciones CSS que sean aplicables a determinados tipos de marcas. Por ejemplo, para utilizar el color verde en todas las cabeceras de nivel 1 de un documento incluiríamos en la cabecera del mismo las siguientes líneas:

```
<STYLE TYPE="text/css">
H1{ color: green }
</STYLE>
```

- Referencias a especificaciones de estilo que se encuentren disponibles en la red. De esta forma, es posible definir un estilo común para un conjunto de páginas, sin necesidad de modificarlas todas si en cualquier momento es necesario cambiar el estilo.

2.1.7 Marcos

Los marcos HTML permiten presentar los documentos en diferentes subventanas independientes entre sí. De esta manera es posible mantener cierta información siempre visible, mientras otra va cambiando. Por ejemplo, así podemos disponer de un menú de navegación siempre fijo que permita seleccionar los diferentes contenidos, que van apareciendo en una parte separada de la ventana del navegador.

Un documento HTML que consta de varios marcos usa para definir su cuerpo las marcas `<FRAMESET>` y `</FRAMESET>` en lugar de `<BODY>` y `</BODY>`.

Existen atributos para definir el esquema de distribución del espacio en la ventana del navegador. Estos atributos son **ROWS** (que define la división horizontal) y **COLS** (para la división vertical). Como valores del atributo se emplea una lista de valores numéricos (en pixels), de tantos por ciento (para hacer una distribución proporcional) o el carácter especial ***** (para indicar el resto de la ventana del navegador). Por ejemplo, la siguiente definición

utiliza tres subventanas horizontales, una con la mitad de la ventana, otra con la tercera parte y la tercera con el resto:

```
<FRAMESET ROWS="50%,30%,*">
```

```
...
```

```
</FRAMESET>
```

Entre las marcas `<FRAMESET>` y `</FRAMESET>` pueden aparecer otras marcas de la misma clase (permitiendo sucesivas subdivisiones del espacio) o marcas `<FRAME>` que definen el documento que aparecerá inicialmente en la subventana. Esta marca ofrece atributos para:

- Asignar un nombre al marco, de manera que pueda ser referenciado por los otros marcos, con `NAME`.
- Especificar el URL del documento que debe aparecer dentro del marco, con `SRC`.
- Definir características visuales como barras de desplazamiento (`SCROLLING`), bordes (`FRAMEBORDER`) o márgenes (`MARGINWIDTH` y `MARGINHEIGHT`).

Para que un enlace haga referencia a un marco concreto en una página con múltiples marcos se utiliza el atributo `TARGET` de la marca de enlaces `<A>`. Utilizando como valor de `TARGET` el nombre de un marco, el navegador cargará el documento identificado por el enlace en el marco en cuestión, dejando el contenido de los demás intacto.

2.1.8 Formularios

Los formularios ofrecen mecanismos para facilitar que la persona que está viendo una página envíe información al servidor. Una vez el servidor la reciba, se empleará algún programa para procesar los datos y enviar una respuesta adecuada.

La estructura general de un formulario es:

- Etiqueta de inicio: `<FORM ACTION="Dst"METHOD="Mtd">`. Donde: `Dst` contiene el URL destino al que se accederá cuando se seleccione la opción de enviar los datos del formulario. `Mtd` es el método HTTP que se desea invocar. El método define el mecanismo que el navegador

utilizará para enviar los datos al servidor. Si se usa el método GET, el valor de los datos irá incluido en el URL de destino. Si se usa el método POST, el navegador entregará los datos cuando se lleve a cabo la conexión HTTP con el servidor.

- Cuerpo del formulario, con los distintos elementos para poder introducir los datos, utilizando la marca, que tiene el atributo obligatorio **TYPE**. Además, tiene asociado el atributo **VALUE**, cuyo contenido es el que será enviado al servidor y que cambia de acuerdo con la entrada que proporcione el usuario. El atributo **TYPE** define el tipo de introducción de datos, que puede ser:
 - Un campo de texto, para **TYPE="TEXT"**.
 - Un campo de texto del que no se muestra el contenido (típico para proporcionar contraseñas), para **TYPE="PASSWORD"**.
 - Un área de texto, de varias filas y columnas, para **TYPE="TEXTAREA"**.
 - Un botón, para **TYPE="BUTTON"**.
 - Un botón asociado a una imagen. para **TYPE="IMAGE"**.
 - Un botón de confirmación, para **TYPE="CHECKBOX"**.
 - Un conjunto de botones del que se puede escoger uno, para **TYPE="RADIO"**.
 - El contenido de un fichero en el ordenador del usuario, que el navegador ayudará a seleccionar, para **TYPE="FILE"**.
 - Existe otra marca que permite añadir un menú desplegable de opciones del que habrá que escoger una. Cada una de las opciones se identifica con la marca **<OPTION>** dentro del ámbito delimitado por **<SELECT>** y **</SELECT>**.
 - Botones de envío y borrado. Existen dos tipos particulares de tipos de entrada que se emplean para que el navegador envíe los datos al servidor (**TYPE="SUBMIT"**) y para que se borren todos los datos del formulario, volviendo los campos a sus valores originales (**TYPE="RESET"**). Al menos el primero debe aparecer siempre en un formulario.
- Por último, el final del formulario es identificado (obviamente) por la marca de fin **</FORM>**.

2.2 XML(Extensible Markup Language)

Este ha sido el lenguaje elegido para los ficheros plantilla y fuente de Webber. En el capítulo que explica el diseño de la aplicación se detallará exhaustivamente el formato y la sintaxis de tales ficheros.

Como hemos dicho en el apartado anterior, una de las tecnologías de cierta importancia para la estandarización y normalización de la información estructurada fue SGML, que procedía de IBM. A partir de él se creó el primer formato estándar en la web, HTML. HTML tuvo mucho éxito pero pasado algún tiempo se hizo patente la necesidad de algo más que imágenes estáticas y texto. Lo que siguió fue una infinita sucesión de tecnologías tendentes a añadir interactividad a la Web. Algunas de estas tecnologías, como Java, han permanecido y florecido, mientras que otras se han quedado en la estacada. Aparte de las tecnologías interactivas como Java, el propio HTML atravesó una rápida serie de mejoras para ensancharse y adaptarse a hacer cosas para las que en un principio no estaba destinado.

Agregar interactividad a HTML, esperando que representara información dinámica, supuso un cambio en su propia naturaleza. Aún más importante que el cambio de información estática a dinámica fue el hecho de que se esperaba que HTML sirviera como forma de almacenar tipos específicos de datos, como transacciones financieras, catálogos de productos y resúmenes. Los desarrolladores web pronto se dieron cuenta de las ventajas de conectar páginas a bases de datos de servidores, en cuyo caso se utilizaba HTML como interface a bases de datos. HTML estaba cambiando de lenguaje de información basado en presentaciones a una tecnología de software. Ahora hay aplicaciones enteras que se implementan como HTML, junto con una serie de scripts, applets de Java, plug-ins y otras tecnologías relacionadas.

A pesar de todo esto los miembros del Consorcio de la World Wide Web (W3C) se dieron cuenta de que sería necesaria una alternativa mucho más amplia a HTML para satisfacer las necesidades futuras de la Web. Quizás la limitación más importante de HTML sea su conjunto finito de etiquetas. Es imposible que los desarrolladores Web añadan etiquetas personalizadas con HTML, lo cual sería infinitamente más útil al tratar con datos pertenecientes a una aplicación o sector específicos. Por ejemplo, un editor de libros utilizaría mucho más etiquetas como `<título>` y `<autor>` que una etiqueta de párrafo genérica `<p>`.

En 1996, el W3C, se propuso introducir el poder y la flexibilidad de SGML en el dominio de la Web. SGML ofrecía tres ventajas importantes que faltaban en HTML: extensibilidad, estructura y validación. Técnicamente, hay estructura en HTML, pero la mayoría de los navegadores obvian si se adhiere a ella o no. SGML está mucho más estructurado que HTML, mientras que el software SGML es mucho menos compasivo que los navegadores web a la hora de aventurarse fuera de la estructura permitida. Así, por razones puramente prácticas, HTML es un lenguaje poco estructurado.

El W3C creó un equipo de expertos en SGML cuyo objetivo era el de crear una nueva tecnología de marcado con las ventajas nucleares de SGML y con la relativa simplicidad de HTML. Por lo que en febrero de 1998 se lanzó la especificación XML 1.0, que significa "*eXtensible Markup Language*" (Lenguaje de Marcado eXtensible)[2].

XML es un subconjunto simplificado de SGML que incorpora muchas de sus características, entre las que se incluyen las tres más importantes: extensibilidad, estructura y validación. En cierta forma, XML representa una nueva época para la Web en la medida que establece una forma de transmitir datos estructurados. XML, en realidad, no es más que un formato de texto estandarizado que sirve para representar información estructurada en la Web. XML está intencionalmente diseñado para que parezca algo sencillo, ya que algo tan aparentemente sencillo e insignificante como un formato de texto puede tener repercusiones vitales en el mundo del software.

XML no sólo es una versión extensible de HTML. De hecho XML no es en absoluto una versión de HTML. Mientras que HTML es un lenguaje de marcado específico que se utiliza para codificar información para presentarla en navegadores web, XML es un **metalenguaje**. Esto implica que HTML, que fue escrito como aplicación específica de SGML, podría rediseñarse como aplicación de XML. De hecho ya está siendo diseñado como aplicación XML, llamado XHTML[1].

Para los diseñadores web, que ven la web como un escaparate para el diseño gráfico, inicialmente XML podría no resultar muy útil. Aunque la presentación juega un papel muy importante en cualquier sitio web, las aplicaciones web modernas son conducidas por datos de tipos muy específicos. Como es fácil deducir, HTML es un lenguaje de marcado muy pobre para representar tales datos. Con su soporte para vocabularios de marcado personalizados, XML abre la puerta a la descripción cuidadosa de los datos y

las relaciones que hay entre las distintas piezas de datos. De hecho, uno de los principales objetivos de XML consiste en separar el contenido (los datos) de la presentación (cómo se ven los datos) en los documentos web.

2.2.1 Los metadatos

Al centrarse en el contenido, XML proporciona una forma de incluir metadatos en documentos web. A continuación vamos a intentar esclarecer este concepto y descubrir su utilidad en Internet.

Teniendo en cuenta tanto su crecimiento (que no ha alcanzado ni remotamente su potencial) como su modelo completamente descentralizado e informal de desarrollo, uno de los problemas fundamentales de la información ofrecida por Internet es su accesibilidad. Es evidente que no nos referimos aquí a una cuestión de disponibilidad física ni de anchos de banda, sino a la posibilidad de que la información en cuestión sea encontrada por aquellas personas que, en un momento determinado, pudieran necesitarla.

Una de las respuestas más evidentes a este problema es la creación de índices que, de un modo u otro, permitan dirigir a un usuario a las piezas de información que necesita. Sin embargo, la enorme cantidad de datos disponible hace que estos índices no puedan ser contruidos a partir de la información en crudo, tal como está disponible en Internet, so pena de ofrecer respuestas incoherentes o, en el mejor de los casos, irrelevantes a las consultas de los usuarios. Conseguir una lista de cientos, incluso miles, de referencias sin interés como respuesta a una consulta es una experiencia, no menos frustrante por lo común, que han sufrido y sufren muchos usuarios de Internet. Obviar este problema y ser capaces de ofrecer a los usuarios respuestas con sentido a sus búsquedas requiere el empleo de índices basados en información sobre la información, o metainformación. Esta metainformación contiene datos como la fecha en que la información fue producida, su autor, o un conjunto de palabras clave que describen su contenido.

Ahora bien, conviene tener en cuenta que la mayor parte de la información contenida en los servidores Internet no ofrece esta metainformación junto con los datos y que, aún en el caso de que así fuera, su adaptación a medida que la información cambia supone una carga adicional que ni los operadores de los servidores ni los autores de la información suelen asumir. Es más, es importante también considerar que la creación y gestión de la metainformación es un trabajo delicado (por ejemplo, la metainformación suele ser empleada

para clasificar el recurso al que hace referencia) y en el que deben emplearse conocimientos específicos.

A la hora de extraer la metainformación acerca de un recurso disponible a través de Internet, hay que tener en cuenta que la información se encuentra fuertemente estructurada dentro de cada uno de los recursos concretos. Se emplean lenguajes con más o menos formalismo, pero con una estructura determinada que facilita la tarea de detectar los datos que resultan relevantes, usando las regularidades y frecuencias detectadas dentro de esa estructura. Por otro lado, es deseable que el sistema sea lo suficientemente flexible como para poder ser configurado de manera simple e intuitiva, sin necesidad de conocimientos específicos.

Como ya se ha comentado anteriormente, un fichero XML describe la estructura y el significado que posee pero no el formato que va tener, con lo que la extracción o asignación de metadatos a su contenido se simplifica enormemente. Las etiquetas que contienen el documento dicen qué información hay en éste, pero no como debe ser presentado. Mientras en HTML se mezcla en el mismo documento el formato, la estructura y la semántica de la página. Por ejemplo, la etiqueta `<B>` simboliza formato (pone en negrita parte del texto), la etiqueta `<STRONG>` simboliza semántica porque está enfatizando la parte de texto que contiene y la etiqueta `<TD>`, que simboliza la celda de una tabla, es un elemento claramente estructural. Veamos un ejemplo para ver las diferencias entre una página HTML y otra XML, en ambos casos describiremos una canción. En HTML tendría el siguiente aspecto:

```
<dt> Hot Cop
<dd> Jacques Morali, Henri Belolo, and Victor Willis
<ul>
<li> Productor: Jacques Morali
<li> Discográfica: PolyGram Records
<li> Duración: 6:20
<li> Escrita en: 1978
<li> Artista: Village People
</ul>
```

Y en XML tendríamos:

```
<CANCIÓN>
<TÍTULO> Hot Cop </TÍTULO>
<COMPOSITOR> Jacques Morali </COMPOSITOR>
```

```
<COMPOSITOR> Henri Belolo </COMPOSITOR>
<COMPOSITOR> Victor Willis </COMPOSITOR>
<PRODUCTOR> Jacques Morali </PRODUCTOR>
<DISCOGRAFICA> PolyGram Records </DISCOGRAFICA>
<DURACION> 6:20 </DURACION>
<AÑO> 1978 </AÑO>
<ARTISTA> Village People </ARTISTA>
</CANCIÓN>
```

Como se puede ver las etiquetas utilizadas con XML pueden hacerse autodescriptivas, por lo que la lectura de dicho fichero resulta más fácil. O mejor aún, si en un fichero existiesen multitud de canciones siguiendo este formato, tanto para una máquina como para un humano detectar las canciones en un fichero escrito en XML será mucho menos tedioso que en uno escrito en HTML. Además cualquier persona que no conozca XML, si lee el trozo de código anterior sabría distinguir perfectamente que se trata de una canción y extraería sin problemas la información almacenada sobre ella como el título, la fecha, los compositores, etc.

XML es ideal para grandes y complejos documentos porque los datos se encuentran muy estructurados. Y no sólo nos permite especificar un vocabulario que define los elementos del documento, sino que además especifica las relaciones entre elementos.

2.2.2 Comentarios

Los comentarios de XML, como los comentarios de la mayoría de los lenguajes de programación, están pensados para anotaciones que no se quiere que tengan influencia sobre los subsiguientes procesamientos. Muchos procesadores descartan los comentarios al analizar los documentos, en cambio DOM, como veremos más adelante, lo considera como un nodo más del árbol que representa al fichero XML. Un comentario es de la siguiente forma:

```
<!-- Estos es un comentario -->
```

Los comentarios en XML no se pueden anidar.

2.2.3 Estructura de un documento XML

La mayoría de los documentos XML comienzan con un prólogo. A veces el prólogo puede ser bastante extenso, aunque en general suele componerse de sólo dos cosas: la Declaración XML y la Declaración de Tipo de Documento (DTD).

La Declaración XML, que debe ser lo primero que aparezca en el archivo XML tiene tres propósitos:

- Identifica la versión de XML a la que se ajusta el documento. De momento, la versión debe ser la 1.0. El número de versión es obligatorio.
- Puede identificar la página de códigos utilizada en el documento. Aunque XML utiliza Unicode, hay muchas maneras distintas de representar los caracteres de un documento (ISO Latin 1, KOI8-R, Big5, etc.). Si no hay declaración, los sistemas XML intentan averiguar el tipo de codificación (a partir de las cabeceras MIME o el sistema de ficheros). Los procesadores de XML sólo están obligados a soportar las codificaciones UTF-8 y UTF-16, pero en la práctica, la mayoría soportan muchas más.
- Puede identificar al documento como independiente (*standalone*). Los documentos independientes informan de que no incluyen declaraciones externas que afecten a la información que pasan a la aplicación de proceso.

El formato de una Declaración XML es:

```
<?xml version='1.0' encoding='US-ASCII' standalone='no'?>
```

El otro elemento común en el prólogo es la de Declaración de Tipo de Documento:

```
<!DOCTYPE nombre-elemento  
PUBLIC "-//Ejemplo//Ejemplo de identificador público de DTD  
//EN//XML" "http://www.ejemplo.com/dtds/ejemplo.dtd">
```

Esta declaración tiene dos propósitos: identifica el elemento raíz del documento (`<nombre-elemento>`) y asocia al documento un conjunto de declaraciones externas, la DTD (Declaración de Tipo de Documento). La DTD se utiliza para ayudar al analizador (parser) a validar el documento XML.

Una de las funciones más importantes que incorporan los analizadores de XML es la capacidad de validar un documento. Una DTD contiene un conjunto de declaraciones que identifican las reglas adicionales y las restricciones que debe satisfacer un documento para que sea considerado una instancia válida de un tipo de documentos asociados a una DTD.

Por ejemplo, una DTD podría afirmar que un elemento `<dirección>` debe tener un elemento `<nombre>`, otro `<calle>` y otro `<telefono>`. Puede llegar incluso a especificar los tipos de los datos que pueden contener esos elementos. Un documento es válido si y sólo si satisface todas las reglas de la DTD a la que está asociado.

2.2.4 Elementos

Los elementos están delimitados por los signos mayor y menor. Los elementos con contenido aparecen con el formato `<nombre-elemento>`, seguido del contenido opcional, seguido de `</nombre-elemento>`. La primera aparición es la etiqueta de inicio, la segunda es la etiqueta de final. En XML, todo elemento no vacío debe tener explícitamente etiquetas de inicio y de final, y deben estar anidadas apropiadamente. A diferencia de HTML, no se puede omitir etiquetas finales nunca.

Los elementos vacíos tienen el formato `<nombre-elemento/>`. La barra final antes del símbolo mayor indica que el elemento no tiene contenido y en consecuencia no se permite una etiqueta de final.

Los nombres de los elementos deben comenzar con un carácter alfabético y pueden contener cualquier número de letras, dígitos, guiones, subrayados y puntos.

2.2.5 Atributos

Las etiquetas de inicio de los elementos pueden tener además atributos (las del final no). En el siguiente ejemplo, el tipo de teléfono de Juan está identificado con un atributo:

```
<direccion>
<nombre> Juan </nombre>
<ciudad> Córdoba </ciudad>
<telefono tipo="trabajo"> 957-334411 </telefono>
</direccion>
```

Los atributos son parejas de nombre-valor. Como los nombres de los elementos, los nombres de los atributos deben ajustarse a la notación de nombres de XML. Todos los valores de los atributos deben ir entrecomillados, con comillas dobles o simples.

2.2.6 Entidades de Carácter

XML se reserva una pequeña cantidad de caracteres, como `<`, para identificar el etiquetado del documento. Para poder utilizar estos caracteres literalmente en un documento se utilizan varios mecanismos. Uno de los más generales es la referencia numérica del carácter. Cualquier carácter Unicode[12] puede ser referenciado en el documento utilizando una referencia numérica al carácter, independientemente de la página de códigos del propio documento. Dicha referencia tiene el formato `&#nnnn;`, donde `nnnn` es el número del carácter Unicode en decimal. Si se quiere utilizar hexadecimal, el formato a seguir es `&#xhhhh;`.

Los caracteres de etiquetado comunes (`<`, `>`, `&`, `'`, y `"`) son tan utilizados literalmente en los documentos XML que se han definido referencias mnemotécnicas para ellos, llamadas entidades y que se resumen a continuación:

- `<`; para `<`.
- `>`; para `>`.
- `&`; para `&`.
- `'`; para `'`.
- `"`; para `"`.

2.2.7 Instrucciones de Procesamiento

Algunas veces se le quiere dar información adicional a una aplicación de procesamiento específica. Por ejemplo, se le puede decir a la aplicación qué nombre de fichero utilizar para enviar de información de salida.

Las instrucciones de procesamiento son la forma de pasar esta información a la aplicación de procesamiento y tienen el siguiente formato:

```
<?destino la-información-que-se-quiere-suministrar?>
```

Todas las instrucciones de procesamiento deben comenzar con un destino. El destino es simplemente un nombre. Todo lo que vaya después del nombre, hasta el `?>` de cierre, es parte de la instrucción de procesamiento. Aunque no es obligatorio hacerlo, se ha convertido en tradición utilizar una sintaxis similar a la de los atributos en las instrucciones de procesamiento, como la siguiente:

```
<?destino nombre-dato="valor"?>
```

Los nombres de destinos que empiecen con `xml` (en cualquiera de las posibles combinaciones de mayúsculas y minúsculas) están reservados.

2.2.8 Secciones CDATA

En las secciones `CDATA` todo el texto es considerado como caracteres de información. Lo que en él pueda parecer una etiqueta o una instrucción de procesamiento, sólo es el texto de la etiqueta o de la instrucción de procesamiento, y el procesador de XML no intentará interpretarlo de ninguna forma.

Las secciones `CDATA` son utilizadas cuando se quiere que el texto sea considerado como puros caracteres de información y no como etiquetas. Estas son muy útiles cuando tenemos un gran bloque de texto que contiene un montón de caracteres `<`, `>`, `&` o `"`, pero no etiquetas. También resultan muy útiles cuando escribimos algo sobre XML en XML.

Por ejemplo, los dos siguientes bloques de código XML serían considerados como información por el procesador de XML, y no serían ni interpretados ni procesados por éste.

```
&lt;?xml version="1.0" standalone="yes"?&gt;
```

```
&lt;SALUDO&gt;
```

```
Hola XML!
```

```
&lt;/SALUDO&gt;
```

Ahora utilizando las secciones `CDATA`:

```
<![CDATA[  
<?xml version="1.0" standalone="yes"?>  
<SALUDO>  
Hola XML!  
</SALUDO>  
]]>
```

El único texto que no está permitido dentro de una sección CDATA es el delimitador de cierre `]]>`.

2.2.9 Documentos bien formados

Un documento XML bien formado es aquel que cumple los siguientes requerimientos:

- Las etiquetas de inicio y fin de todos los elementos están anidadas adecuadamente.
- Los valores de los atributos no contienen caracteres `<` (la entidad `<` sí se puede utilizar).
- Todas las referencias numéricas a caracteres se refieren a caracteres Unicode que están permitidos en documentos XML.
- Las entidades con nombre se utilizan de forma correcta (están declaradas, están estructuradas y no son recursivas).

2.3 DOM (Document Object Model)

Los programadores y diseñadores de sitios web persiguen acceder y manipular los contenidos de los documentos web de diferentes formas. Muchas veces se utilizan scripts de una forma u otra para interactuar con el usuario; o bien para validar formularios, o bien para crear efectos dinámicos visuales. Pero en definitiva, el script accede y manipula las partes relevantes del documento web y el usuario ve los resultados. La gran barrera con la que se encontraban los desarrolladores de estos scripts era que no había un interface estándar para manipular el documento. Esto significa, por ejemplo, que los diseñadores del sitio web deben aprender las diferentes formas en la que los navegadores manipulan la parte relevante del documento para conseguir que el efecto deseado se aprecie para cualquier usuario.

Este interface para el documento es denominado *Document Object Model* (DOM), y se basan en una aproximación en la que el documento web está modelado por medio de objetos. Las partes del documento (elementos, atributos, comentarios ..) son tratados como objetos, los cuales pueden ser accedidos y manipulados. El documento puede ser procesado al completo o por partes.

El W3C ha desarrollado un interface para documentos HTML y XML. El grupo de trabajo del DOM de W3C ha completado la primera especificación. El DOM Level 1 define un API neutral para acceder, manipular y navegar documentos HTML y XML. Un API se compone de una serie de funciones de las cuales sólo conocemos su interface, la cual utilizamos para realizar las llamadas, pero no conocemos cual es su implementación. Define funciones primitivas, y otras funciones que pueden ser combinación de las primitivas y que se utilizan para tareas muy comunes pero que generalmente no forman parte del interface del nivel 1 de DOM. Existen implementaciones de DOM que ofrecen, además de las funciones básicas del DOM, funciones aún más complejas para otras tareas no directamente especificadas en el DOM.

Uno de los objetivos del DOM es evitar implementaciones forzadas, por lo que sólo se especifica el **interface** que debe tener el modelo. Los interfaces son un concepto a menudo utilizado en Java[8] y otros lenguajes de programación orientados a objetos. El que una clase implemente ciertos interfaces, significa que la clase contendrá las funciones, métodos y propiedades especificadas en el interface. La clase puede implementar los interfaces como el implementador lo desee. Los interfaces pueden ser heredadas, y la herencia múltiple es posible. Incluso con la herencia no hay intención de compartir código u otros detalles de implementación. La clase puede además implementar otras funciones, métodos o propiedades que no estén especificadas en el interface.

El DOM Level 1 define cómo manipular objetos encontrados en documentos HTML o XML. Por lo tanto, esto es importante para saber cuales son las interfaces de esos objetos. La figura siguiente muestra el árbol de herencia del núcleo de las interfaces del DOM Level1, que son derivadas desde la clase principal:

Todas las partes de un documento heredan de *Node*. Los objetos más comúnmente utilizados en el DOM son *Node*, *Element*, *Attr*, *Document*

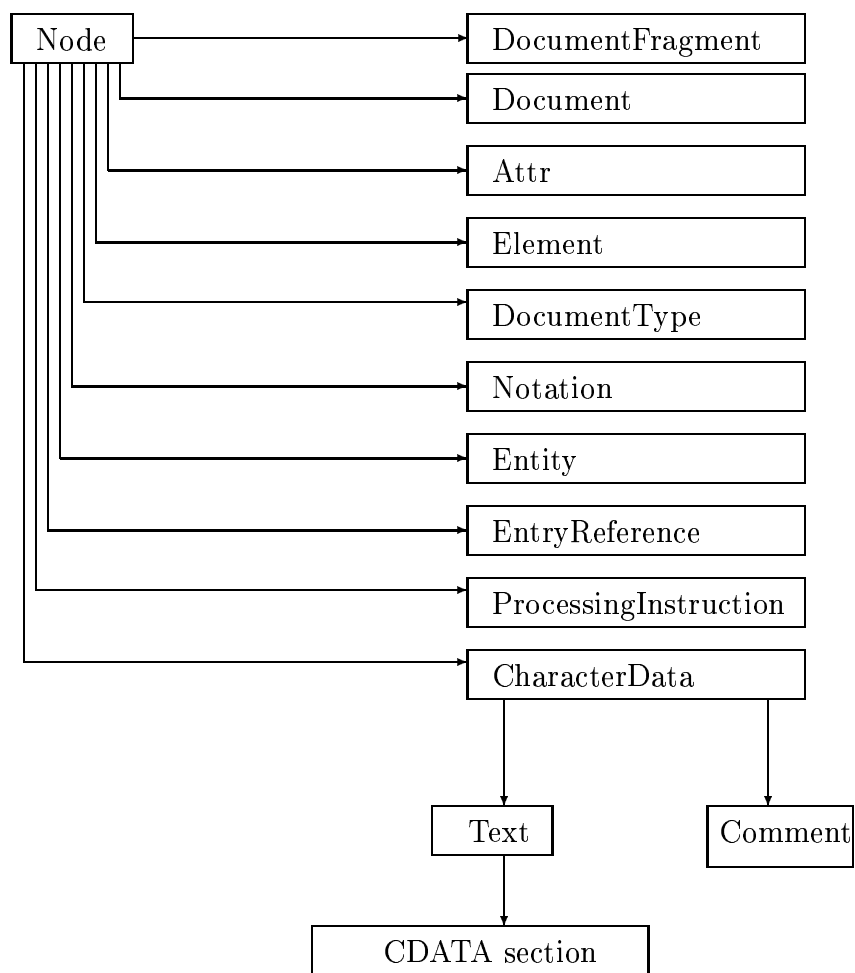


Figura 2.1: Árbol de herencia de DOM Level 1

y **Text**.

El interface básico es **Node**. Tiene métodos para borrar, sustituir e insertar, los cuales están disponibles para todas las interfaces que lo heredan. Los métodos básicos para navegar están definidos en **Node**, tales como las funciones básicas para recorrer un árbol: **ParentNode** (obtiene el padre de un nodo), **firstChild** (mediante este método obtenemos el primer hijo de un nodo padre), **nextSibling** (con este método podemos recorrer los nodos hijos de un nodo, ya que con él se obtiene el siguiente nodo hijo con respecto en el que estamos procesando actualmente) y **previousSibling** (este otro método también nos permite recorrer los nodos hijos pero en sentido contrario a la anterior, ya que nos devuelve el nodo hermano anterior al que estamos procesando actualmente). Algunos usuarios del DOM prefieren una especie de matriz para navegar por el documento, por este motivo, la interface **Node** contiene una forma de obtener una lista de nodos de todos los hijos de una determinada instancia de **Node**. Este nodo puede ser el documento completo o un elemento en particular.

Element representa los elementos del documento. Es derivado de **Node** por lo que tendrá todos los métodos y propiedades de él. La mayoría de los documentos tiene más **Element** y **Text** que de cualquier otro tipo de objeto.

Attr representa un atributo asociado con un elemento. Los nodos **Attr** no son nodos hijo de **Element**, por lo que los atributos de **Node**, **ParentNode**, **PreviousSibling** o **NextSibling** tienen un valor nulo para los objetos atributo.

Text representa parte del contenido textual en un documento. Cuando un documento es parseado al principio dentro del modelo de la estructura DOM, hay un solo nodo de texto por cada bloque de texto existente en el documento. Como el usuario de DOM manipula el documento se pueden ir creando múltiples nodos de texto adyacentes. Puesto que no hay manera de especificar los límites de estos nodos de texto en HML o XML, el método **normalize()** sobre un **Element** nos va permitir unir todos los nodos de texto adyacentes en uno único; cosa que es aconsejable realizar antes de aplicar algunas operaciones que dependan de una estructura en particular del documento.

Document representa al documento completo. Es conceptualmente la raíz del árbol del documento, y suministra acceso a todos los datos que hay dentro

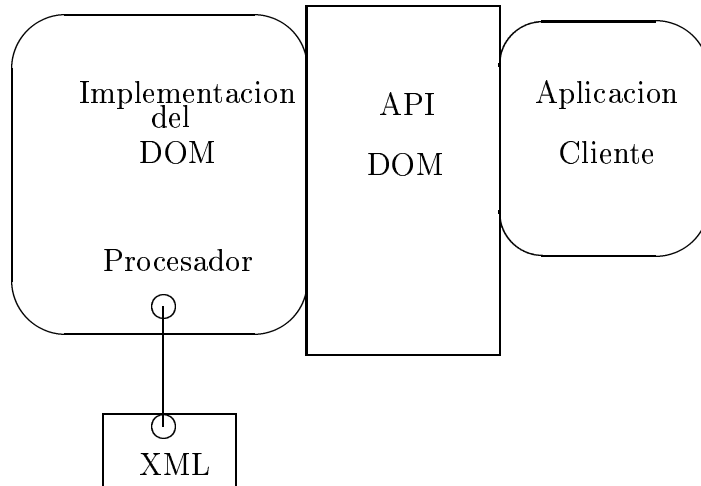


Figura 2.2: Arquitectura de un procesador HTML/XML que implementa el interface DOM

del documento. Todos los nodos de un documento en particular poseen exactamente el mismo valor para la propiedad `ownerDocument`, la cual apunta al documento con el que está asociado.

El programa que incorpora la implementación DOM, por ejemplo un navegador o un editor, tiene que leer el documento y cambiarlo a una representación interna que permita acceder por medio de las interfaces DOM. Este proceso de lectura depende de la implementación y se muestra en la figura.

El procesador HTML/XML lee el documento o cadena de datos en una representación interna. En la especificación de DOM, a esto se le conoce como representación interna del modelo, y a menudo es implementada como un árbol. La aplicación del cliente (a menudo un script o applet) accede al documento parseado por medio de las funciones y métodos definidos en el API DOM.

La especificación de DOM Level 1 se organiza en tres partes:

- Interfaces fundamentales, que son las interfaces a estos objetos que son comunes a los documentos HTML y XML, tales como atributos, elementos, comentarios, texto.
- Interfaces extendidas, que son para objetos XML, tales como `Entity` y `CDATA Sections`.

- Los interfaces HTML, que suministran interfaces especiales para el conjunto de marcas de HTML.

2.4 Perl

Por fin llegamos al lenguaje en el que se encuentra codificada la aplicación: Perl. Perl es un lenguaje de programación muy fácil de aprender y de utilizar, y en el cual no hay que decir mucho más de lo que realmente se quiere decir. En muchos lenguajes de programación tenemos que declarar los tipos, variables y subrutinas que vamos a utilizar antes de escribir ni una sola línea de código ejecutable. En Perl escribiendo simplemente `print "Hello World!"`; ya tenemos un programa que imprime en pantalla la tan conocida frase *"Hello World!"*.

Se trata de un lenguaje que facilita mucho la manipulación de números y texto, de ficheros y directorios, de computadoras y redes y como no de programas. Es capaz de ejecutar programas externos y leer su salida, además ofrece un intuitivo depurador para sus programas; los cuales van a resultar muy fácil de compilar y ejecutar. Y lo que es mejor aún, son portables a cualquier máquina: Linux, Windows, Macintosh, OS/2, VMS, Plan9. Perl es especialmente popular en programadores de sistemas y desarrolladores Web, pero cada vez está abarcando otros muchos campos.

Perl es un lenguaje de libre distribución y hay muchos lugares en Internet donde podemos encontrar módulos para él que nos permiten hacer todo aquello que se nos pase por la cabeza. Sin ir más lejos, en este proyecto se utilizan varios de esos módulos, entre ellos `XML::DOM` que nos va a suministrar un interface muy simple para manipular ficheros XML que son accedidos de acuerdo con el DOM Level 1.

Perl es un lenguaje simple en muchos aspectos: no hay que saber indicaciones especiales para compilar un programa; los tipos y estructuras de datos son fáciles de utilizar y comprender; y además Perl no impone limitaciones arbitrarias sobre los datos que almacenamos, por ejemplo, las cadenas y arrays pueden crecer tanto como se desee (siempre que tengamos memoria suficiente). Y algo que es muy importante, no hay que conocer todo lo referente a Perl para poder empezar a programar, sino que es un lenguaje que se puede aprender poco a poco.

En un principio estaba pensado sólo para navegar por ficheros, escanear y manipular grandes cantidades de texto, crear y obtener dinámicamente datos e imprimir fácilmente dichos datos almacenados. Pero a lo largo del tiempo, Perl ha empezado a crecer, y ahora lo podemos utilizar para administrar bases de datos, para la programación funcional y orientada a objetos, para la programación cliente-servidor. etc... Perl está diseñado para ser modularmente extensible, nos permite rápidamente diseñar, programar y depurar aplicaciones; pero además es muy fácil de extender la funcionalidad de éstas tanto como sea necesario. Podemos embeber Perl en otros lenguajes, al igual que podemos embeber otros lenguajes en Perl. A través del mecanismo de importación de módulos, podemos utilizar definiciones externas como si se trataran de características propias de Perl.

A diferencia de un lenguaje estrictamente interpretado tal como el shell, Perl primero compila el programa completo rápidamente en un formato intermedio. Como cualquier otro compilador, realiza varias optimizaciones y nos da información acerca de los posibles errores sintácticos y semánticos. Una vez que el compilador no ha detectado ninguno error de este tipo, le pasa el código intermedio al intérprete para que ejecute el código. Esto suena complicado, pero el compilador y el intérprete son bastantes eficientes, y podemos apreciar que estas tareas se realizan en muy poco tiempo.

El language Perl se percibe habitualmente como un language intermedio entre los shell scripts y la programación en C. En efecto, los programas Perl son una sucesión de instrucciones y son similares a los shell scripts porque no existe un procedimiento principal como la rutina "main" en C. Sin embargo, se parece al language C en su sintaxis y en el número importante de funciones que permiten la manipulación de cadenas de caracteres y archivos.

2.4.1 Los tipos de datos

El language Perl posee tres tipos de representaciones de datos, las cuales permiten asociar a cada variable utilizada un tipo. Hay que tener en cuenta que las variables Perl no tienen que declararse antes de su uso, sino que se asignan de manera dinámica y se les asigna un valor predeterminado en función del contexto.

- **Los escalares:** El escalar representa el tipo básico. Permite representar enteros, reales y cadenas de caracteres. Las variables de tipo escalar van precedidas por el símbolo \$.

Los valores numéricos especificados literalmente se presentan en forma de valores reales codificados en doble precisión. Este formato interno se utiliza para todas las operaciones aritméticas. Los valores enteros no pueden empezar por 0 porque ésto los distingue de la codificación octal (empieza por 0) o hexadecimal (empieza por 0x):

```
$x = 0.75      # Un real.

$y = 1234      # Un entero.

$x = 0377      # Equivale a $x = 255.

$y = 0xff      # Equivale a $y = 255.
```

Un tipo escalar también permite representar las cadenas de caracteres, las cuales se especifican literalmente por medio de una sucesión de caracteres delimitada por comillas "...", por ejemplo:

```
$str = "Hola mundo";
```

Los operadores escalares definidos en Perl contienen todos los operadores estándar a los que se añaden operadores específicos para la manipulación de cadenas de caracteres: `.` (concatenación de dos cadenas), `.=` (concatenación y asignación) y `x` (repetición de escalares).

- **Las matrices de escalares:** Una matriz es una lista ordenada de datos de tipo escalar. Cada elemento de la lista es una variable escalar a la que se le asocia un valor. Las variables de tipo matriz se identifican por el prefijo arroba `@`. Por ejemplo:

```
@numeros = (3, 1, 23, 45);

@letras = ("esto", "es", "ün", "ejemplo");
```

Los elementos de una matriz se referencian mediante índices. El primer elemento se referencia por el índice 0. Por ejemplo:

```
$numero[2]; # Vale 23

$letras[3]; # Vale "ejemplo"
```

También se puede acceder a una parte de la matriz utilizando el operador `..` que define un intervalo de índices.

Los operadores definidos sobre el tipo matriz son: `$#` (permite determinar el último índice válido) y `$[` (indica el primer índice válido del array).

Las matrices se asignan de manera dinámica y su tamaño se modifica en función de la asignación de nuevos elementos. Los operadores **push** y **pop** permiten respectivamente insertar y suprimir el último elemento de una matriz. Y los operadores **shift** y **unshift** son equivalentes a los anteriores pero se aplican al primer elemento de la lista. Con **reverse** y **sort** podemos invertir el orden de los elementos y clasificar sus elementos por orden.

- **Las listas asociativas:** Las listas asociativas son idénticas a las listas de escalares para las que algunos elementos se consideran como índices. Estos no son valores enteros sino escalares de cualquier tipo. Representan las claves que permiten acceder a los valores correspondientes. Las variables de este tipo se identifican mediante el prefijo `%`. El acceso a los elementos de este tipo de variables se hace mediante una clave representada por una cadena de caracteres. Por ejemplo:

```
# Crea la clave "aux" y le asocia el valor "bbb"
$es{"aux"} = "bbb"
# Crea la clave "3.14" y le asocia el valor "pi"
$math{"3.14"} = "pi"
```

El operador **keys** proporciona la lista de claves o índices de una lista asociativa. El operador **values** devuelve la lista de valores. El operador **each** nos permite recorrer la lista iterativamente y por último **delete** elimina elementos de una lista asociativa.

2.4.2 Las expresiones regulares

Una expresión regular es un modelo o una forma de comparar con una cadena de caracteres. Esta comparación conocida con el nombre de *pattern matching* o reconocimiento de patrones, permite identificar las ocurrencias del modelo en los datos tratados. La utilización principal de las expresiones regulares en

Perl consiste en la identificación de cadenas de caracteres para la búsqueda, modificación y extracción de palabras clave.

Las expresiones regulares se tratan mediante el operador `/ /`. Este operador puede declinarse de varias formas y representa una herramienta muy potente para la manipulación de cadenas de caracteres.

Por ejemplo, si se desea identificar todas las líneas de un archivo de texto que contienen el modelo *html*, basta con considerar la comprobación siguiente:

```
if (/html/) {  
    print $_;  
}
```

El argumento predeterminado del operador `/ /` es la variable `$_`. Esta variable contiene la cadena de caracteres o la línea actual. Esta variable puede sustituirse por una variable cualquiera mediante el operador `=~`. Por ejemplo:

```
if ($linea) =~ /html/ {  
    print $linea;  
}
```

En este ejemplo el modelo es muy simple porque se compone de una cadena de caracteres. El operador `/ /` permite la representación de modelos mucho más complejos. Estos pueden integrar información relativa al contexto, tal como la búsqueda de líneas que empiecen por la cadena de caracteres, la extracción de palabras que tengan sufijos o prefijos particulares...

2.4.3 Los operadores `split` y `join`

Los operadores **`split`** y **`join`** permiten transformar una cadena en una lista de valores y viceversa. Esta manipulación de cadena es muy útil cuando los datos a tratar se concatenan en una cadena de caracteres. Por ejemplo, cuando un usuario envía un formulario al servidor WWW usando el método GET, su contenido se transmite tras concatenar pares nombre-valor separados por el símbolo `&`. Para acceder a la información transmitida, es necesario descomponer esta cadena de caracteres.

La función **`split`** admite dos argumentos, una expresión regular y una cadena de caracteres. La expresión regular define el modelo que permitirá descomponer la cadena de caracteres. La sintaxis es la siguiente:

```
@lista = split(/expresión/,cadena);
```

El operador **join** realiza la tarea inversa. Así, a partir de un separador y de una lista de valores, concatena los elementos de la lista y la cadena de caracteres que representa el separador. La sintaxis del operador join es:

```
$cadena = join(separador,@lista)
```

2.4.4 Entrada / Salida

El script Perl interactúa con el usuario o el sistema operativo por medio de entradas/salidas que permiten el intercambio de información. Este intercambio de datos se gestiona por medio de operadores específicos que configuran un interface entre el script y su entorno.

Perl accede a los archivos de tipo ASCII por medio de punteros a archivos. Estos se referencian mediante variables y constituyen el enlace entre el script y el archivo consultado. Los archivos se consultan mediante la rutina **open** que admite dos argumentos, un puntero de archivo y un nombre de archivo:

```
open (PUNTERO, "nombre_archivo");
```

El argumento de open representa el nombre del archivo, y puede tomar uno de los prefijos: < (abre el archivo en modo lectura), > (abre el archivo en modo escritura, si el archivo existe vaciará su contenido), >> (abre al archivo en modo concatenación, y | (el resultado del script se envía a un programa). El lenguaje Perl posee tres punteros de archivos predefinidos **STDIN**, **STDOUT** y **STDERR**, que son inicializados automáticamente para cada script, y permiten comunicar con el proceso que haya inicializado el script.

Una vez inicializado el puntero mediante la rutina open, este mismo puntero permite acceder al contenido del archivo por medio del operador < >. El contenido del archivo se recorre línea a línea mediante la estructura <PUNTERO>. Al final de la utilización, el archivo se cierra mediante el operador **close**.

2.4.5 La definición de funciones

La definición de funciones permite asociar un nombre a un conjunto de instrucciones Perl. La sintaxis para la definición de función es:

```
sub funcname {  
    instrucciones;  
}
```

donde **funcname** representa el nombre de la función. A cada llamada a la función, el conjunto de instrucciones delimitadas por llaves se ejecuta. La definición de una función es global y en consecuencia puede insertarse al principio o al final del archivo.

El valor devuelto por una llamada a una función corresponde al valor precisado por el operador **return**. Cuando éste está ausente, el resultado de la función corresponde al resultado de la última instrucción ejecutada en el cuerpo de la función.

Al llamar a una función, cuando ésta va seguida de una lista de variables, éstas últimas se consideran como sus argumentos. El language Perl duplica estas variables en la variable especial `@_` durante la ejecución de la función. La variable `@_` permite así acceder a los parámetros de la función. Éstos se colocan en la matriz `$_[0]`, `$_[1]`, ..., `$_[n]` referenciando los diferentes argumentos de la función.

2.4.6 Paquetes, módulos y orientación a objetos

Los paquetes suministran el bloque de construcción fundamental a partir del cual se basan conceptos de más alto nivel como los módulos y las clases.

Los paquetes son elementos independientes, donde podemos tener varios paquetes en un mismo fichero o un sólo paquete distribuido por varios ficheros; aunque lo usual es tener un paquete por cada fichero. El nombre de dicho fichero deberá ser el mismo que el del paquete añadiéndole la extensión `.pm`. El comienzo de la declaración de un paquete se hace con la palabra reservada **package** seguido del nombre del paquete. Todos los símbolos (variables y funciones) que sigan a esa línea de código pertenecerán a la tabla de símbolos de dicho paquete. Un paquete puede ser incluido dentro de un fichero utilizando **require** o **use**.

Para referirnos a identificadores de otros paquetes distintos del que nos encontramos actualmente utilizamos la siguiente notación:

Paquete::Identificador

donde **Paquete** es el nombre del paquete que contiene la variable o procedimiento denominada **Identificador**.

Un módulo no es más que un paquete reutilizable que está definido en un fichero de librería cuyo nombre es el mismo que el nombre del paquete (con la extensión `.pm`). Un módulo suministra un mecanismo para exportar algunos de sus símbolos en la tabla de símbolos de cualquier otro paquete o fichero que lo utilice. Para incluir un módulo en un programa hay que utilizar la siguiente instrucción:

```
use Modulo;
```

donde `Modulo` es el nombre del módulo que queremos exportar.

Un *objeto* es una estructura de datos que soporta una colección de comportamientos. Un objeto obtiene su comportamiento cuando realiza la instancia de una *clase*, la cual define los métodos que será aplicados sobre ese objeto. A los métodos que crean nuevos objetos de una clase se les llama *métodos constructores*.

En Perl, un objeto es una referencia a la clase a la que pertenece; una clase es un paquete que suministra los métodos que definen el comportamiento de los objetos que se instancien a partir de ella, y un método es una subrutina que espera como primer argumento una referencia del objeto sobre el que va a actuar.

2.4.7 El módulo XML::DOM

El módulo `XML::DOM` de Perl va a ser el encargado de realizar el interface para trabajar cómodamente con los ficheros XML de entrada a Webber. Este módulo nos va a facilitar en gran medida manipular, interpretar y modificar estos ficheros, y lo que es aún más importante, el hecho de poder trabajar con varios ficheros simultáneamente para obtener un documento resultante como mezcla de todos ellos siguiendo las reglas que en el apartado de diseño se detallarán.

Este módulo está basado en el nivel 1 de DOM, y para su funcionamiento requiere la instalación previa de otros dos módulos más:

- El módulo **XML-Parser**: Este módulo le suministra el interface para parsear documentos XML.
- El módulo **expat**: Este módulo le suministra al anterior una librería de aún más bajo nivel para manipular y parsear documentos XML. Cuando llamamos a un método de la librería anterior para parsear un documento, inmediatamente es creado un objeto `XML::Parser::Expat`.

Estos módulos pueden ser adquiridos en **CPAN** (Comprehensive Perl Archive Network), en el URL siguiente: <http://www.cpan.org>. En este sitio web podemos encontrar todo tipo de recursos para el lenguaje de programación Perl, podríamos decir que es la más grande y actualizada distribución de Perl.

Cuando creamos un objeto `XML::DOM::Parser`, los métodos `parse` -para parsear cadenas- y `parsefile` - para parsear ficheros - crean un objeto `XML::DOM::Document` a partir de la entrada. Este objeto documento puede ser examinado, modificado, escrito en un fichero o convertido a una cadena. Un documento tiene una estructura de árbol compuesta de objetos `Node`. Un nodo puede contener otros nodos dependiendo de su tipo. Un documento puede tener nodos elemento, de texto, comentarios, o secciones `CDATA`. Los nodos elemento pueden tener atributos, elementos, texto, comentarios y secciones `CDATA`. El resto de nodos no puede tener nodos hijos.

Algunos de los procedimientos que este módulo nos aporta para manipular ficheros XML son los siguientes:

- **getNodeType**: Nos devuelve el tipo de un nodo (sección `CDATA`, elemento, comentario, texto...)
- **get/setNodeValue**: Para obtener/establecer el valor de un nodo.
- **get/setParentNode**: Para obtener/establecer el padre de un nodo.
- **getChildNodes**: Nos devuelve la lista completa de todos los hijos que posee un nodo. Esta lista es dinámica y se modifica automáticamente conforme el árbol este siendo actualizado.
- **getFirstChild**, **getLastChild**, **getPreviousSibling** y **getNextSibling**: Estas funciones se utilizan para recorrer una lista de hijos de un nodo.
- **getElementsByTagName**: Nos devuelve la lista de hijos elemento que coinciden con el nombre que se le pasa como parámetro. Si lo pasamos como parámetro el carácter '*' nos devuelve todos los elementos hijo del nodo.
- **getAttribute**: Nos devuelve el valor del atributo cuyo nombre le pasamos como parámetro del elemento actual.
- **createTextNode**: Nos permite crear un nodo de texto que luego insertaremos en el árbol DOM.

2.4.8 Perl/Tk

Como ya hemos dicho anteriormente, Perl es un buen lenguaje para el procesamiento de ficheros, conexión a bases de datos, etc... Pero, por muchos años, los programas Perl estaban limitados al interface por línea de comandos. El módulo Tk cambió todo esto.

La extensión Tk para Perl permite crear interfaces gráficas para nuestros programas. Utilizando los módulos incluidos con la distribución de Tk se pueden crear ventanas, botones, texto y otros tipos de widgets para obtener un interface gráfica adecuada a nuestra aplicación.

Al igual que Perl, este módulo es también portable a una máquina Windows, es más, existen compiladores para estos sistemas operativos que a partir de un fichero fuente Perl como el de esta aplicación son capaces de generarnos un fichero ejecutable en cualquier máquina Windows sin necesita de tener instalado nada más: ni Perl, ni el módulo Perl/Tk.

La motivación principal de usar Perl/Tk ha sido el hecho de haber realizado la codificación de Webber en Perl, por lo que se sigue manteniendo este mismo lenguaje para implementar su interface gráfica.

Perl/Tk nos suministra las funciones necesarias para crear los objetos gráficos o **widgets** más utilizados para dotar de funcionalidad a las ventanas del interface gráfico de la aplicación, como por ejemplo: botones, barras de scroll, campos de texto, etc.. A continuación podemos observar la única línea de código que necesitaríamos para añadir a nuestra ventana principal \$mw un botón etiquetado con el texto "Algo".

```
mw->Button(-text=>"Algo")->pack();
```

Hay multitud de widgets al igual que el botón -y mucho más complejos- disponibles para ser usados. Es más, si no se encuentra el widget que se desea podemos implementarlo mediante una combinación de los existentes. Con sólo unas pocas líneas de código se podría mostrar un botón y otros widgets más que en su conjunto completan la funcionalidad de una aplicación.

Capítulo 3

El modelo Webber

3.1 Introducción

El desarrollo de aplicaciones Web se realiza mayoritariamente ad hoc, no hay una propuesta sistemática para desarrollar este tipo de aplicaciones, por lo que se está llevando a cabo basándose principalmente en el conocimiento y experiencia de desarrolladores individuales. Una razón a la carencia de una propuesta estructurada puede estar en considerar a la Web como un medio de información más que como una plataforma de aplicaciones. El desarrollo Web no está siendo visto actualmente como un problema de desarrollo de software al cual habría que aplicarle los principios de la ingeniería del software ya establecidos y consolidados.

El modelo de implementación Web no se relaciona con el estado del arte de los modelos de desarrollo de software. La implementación Web está basada en tecnologías de bajo nivel que no suministran abstracciones de alto nivel para compartir y reutilizar. La carencia de abstracciones apropiadas hace difícil la construcción de esquemas que capturen decisiones de diseño arquitectural para la reutilización de diferentes partes en una misma aplicación o incluso en otra diferente. La carencia de abstracción hace además difícil mantener y desarrollar aplicaciones basadas en la Web. Las decisiones de diseño son difíciles de tomar en implementaciones de tan bajo nivel, y cualquier cambio en la aplicación puede llevarnos a la inconsistencia. El problema del mantenimiento es crítico en este tipo de aplicaciones ya que éstas tienden a evolucionar rápidamente, con frecuentes actualizaciones y rediseños.

Un modelo de desarrollo de aplicaciones Web basado en el uso de componentes permitiría aplicar técnicas de ingeniería del software al desarrollo de

tales aplicaciones. En este modelo se deben reflejar los principios de la orientación a objetos tales como modularidad, abstracción y encapsulación. Esto facilita la descripción de conceptos de alto nivel y esquemas para reutilizar que pueden hacer de puente entre el diseño a alto nivel y la implementación a bajo nivel.

El proceso de desarrollo usual de software está compuesto de cuatro fases: Análisis, diseño, implementación y mantenimiento.

Durante el **análisis**, los desarrolladores construyen un modelo de la aplicación en términos de dominio. Idealmente, este análisis se concentra en el sistema de espacio de la aplicación. En esta fase no se tiene en cuenta para nada si la aplicación depende o no de la Web para su correcto funcionamiento.

Basándonos en ese análisis, un modelo de la solución software es definido durante la fase de **diseño**. Obviamente, el proceso de desarrollo de la aplicación Web -en contraste con el proceso para el software en general- asume una solución para un software basado en la Web y debe, por tanto, acomodarse a esto.

La **implementación** transforma el diseño en el software que va a ejecutarse. Para aplicaciones Web, esta fase debe tener en cuenta la actual tecnología para implementar este tipo de aplicaciones dependiente de la red.

Finalmente, en la fase de **mantenimiento** se realizan las modificaciones que el software vaya necesitando, y dichas modificaciones pueden afectar a cualquier nivel (análisis, diseño o implementación).

Los procedimientos generales del software son difíciles de relacionar con el modelo de desarrollo de aplicaciones Web. El modelo de implementación Web está basado en la *decomposición* de aplicaciones en *recursos*. Los recursos colaboran entre sí para conseguir un objetivo común a todos ellos; están residentes en el servidor Web a la espera de la petición de un cliente para desarrollar su tarea. Pueden ser estáticos o dinámicamente generados por un script, pero deben estar especificados inherentemente, es decir, no deben capturar abstracciones.

La World Wide Web fue originalmente diseñada como un medio de información para distribuir equipos de recursos. Un objetivo clave era facilitar lo más posible a los autores la tarea de desarrollar documentos, y la noción del

desarrollo de aplicaciones Web se reducía al desarrollo de un documento por el propio autor o pequeño grupo de autores. Para este tipo de desarrollo, el ciclo de vida comprende un análisis informal de lo que va a ser presentado, un diseño informal de cómo se estructura toda la información mediante hiperenlaces, y la implementación se lleva a cabo mediante un lenguaje de marcado. El mantenimiento de los documentos se realizaba por los propios autores.

El modelo de implementación Web estaba diseñado para conocer los requisitos de estos ciclos de vida. Estaba basado en la noción de recursos que la mayoría de la veces autocontenían trozos de información, y mediante los enlaces eran combinados dentro de un conjunto coherente de documentos, como por ejemplo un sitio Web. El concepto de desarrollo mediante recursos se ajusta perfectamente al concepto de ciclo de vida del desarrollo de un documento, además soportan el principio de modularidad en este contexto.

De cualquier forma, el uso de la Web ha sido llevado más allá de su ámbito original, incluso en el desarrollo de documentos. El ciclo de vida de un sitio Web está basado típicamente en un análisis intensivo de requisitos en términos de contenido, estructura, acceso y comprobación de identidad. El diseño debe descomponer estos requisitos en recursos e hiperenlaces; además de crear un plan para las direcciones. El mantenimiento debe enfocarse sobre la evolución de contenidos y en la integridad del sitio Web.

La noción de recurso está resaltando que el modelo de implementación Web no conoce los requisitos de este tipo de ciclos de vida. Es decir, el modelo de implementación Web no suministra abstracciones para capturar diseño estructural que se puedan reutilizar, o lo que es más, el plan y las estructuras de navegación son comúnmente reutilizadas en diferentes partes del sitio.

En resumen, el desarrollo de aplicaciones en el entorno Web es radicalmente diferente de la forma usual de desarrollo de software e impone una propuesta diferente para el desarrollo de la aplicación. Obviamente, el diseño y mantenimiento de tales aplicaciones debería centrarse sobre todo en el interface de usuario, lógica de la aplicación y acceso a bases de datos, más que en los recursos. La actual tecnología de implementación Web es a demasiado bajo nivel como para imponer un propio proceso de desarrollo.

Este modelo utiliza los componentes como un concepto uniforme para modelar entidades Web en niveles arbitrarios de granularidad y abstracción. En contraste con estos recursos, los componentes no están fijados para un cierto tamaño de grano, pero si están diseñados para capturar diseños a su granularidad natural. Por ejemplo, los componentes pueden capturar una unidad de contenido como diseño independiente de una página web, la cual estará separada del diseño. Soportar la granularidad arbitraria significa que los componentes pueden modelar entidades web tan pequeñas como enlaces individuales. Además pueden ser asociados con un recurso completo, por ejemplo, un documento HTML o script generador de un documento Web. En contraste a otras propuestas de modelos Web orientadas a objetos, este modelo no requiere componentes-objeto para ser compuesto o derivado desde un conjunto de primitivas ya dado; algunos conjuntos de primitivas pueden ser modelados como los bloques de construcción de una aplicación, por ejemplo, las primitivas del interface de usuario, o las primitivas para un método de diseño específico.

Los componentes puede referenciar a otros componentes en el modelo de agregación o en el de especialización. Por ejemplo, una página modelada por componentes puede referenciar componentes que modelen la cabecera, el cuerpo u otras partes. Como ejemplo, un componente que modele una estructura de navegación puede referenciar los componentes modeladores de enlaces y anclas.

Por medio de un especial tipo de referencia, los componentes pueden hacer referencia a los llamados componentes plantilla, desde los cuales heredan su estado y comportamiento. Cualquier componente puede funcionar como un plantilla. Así, este modelo está basado en un paradigma de instaciamiento de plantillas, el cual elimina la distinción entre clases e instancias tan conocidas en los modelos de objeto. Este paradigma es naturalmente apropiado para modelar aplicaciones Web, primero porque algunas entidades Web son únicas, y segundo porque el prototipado refleja el estilo de "copiar y modificar" utilizado en la reutilización de aplicaciones Web. Este modelo de composición Web se asemeja conceptualmente a los modelos de diseño orientados a objetos porque utilizándolo es fácil emular una clase viéndola como un prototipo.

3.2 Modelo Webber

La figura ilustra el modelo de ejecución en el que se basa Webber:

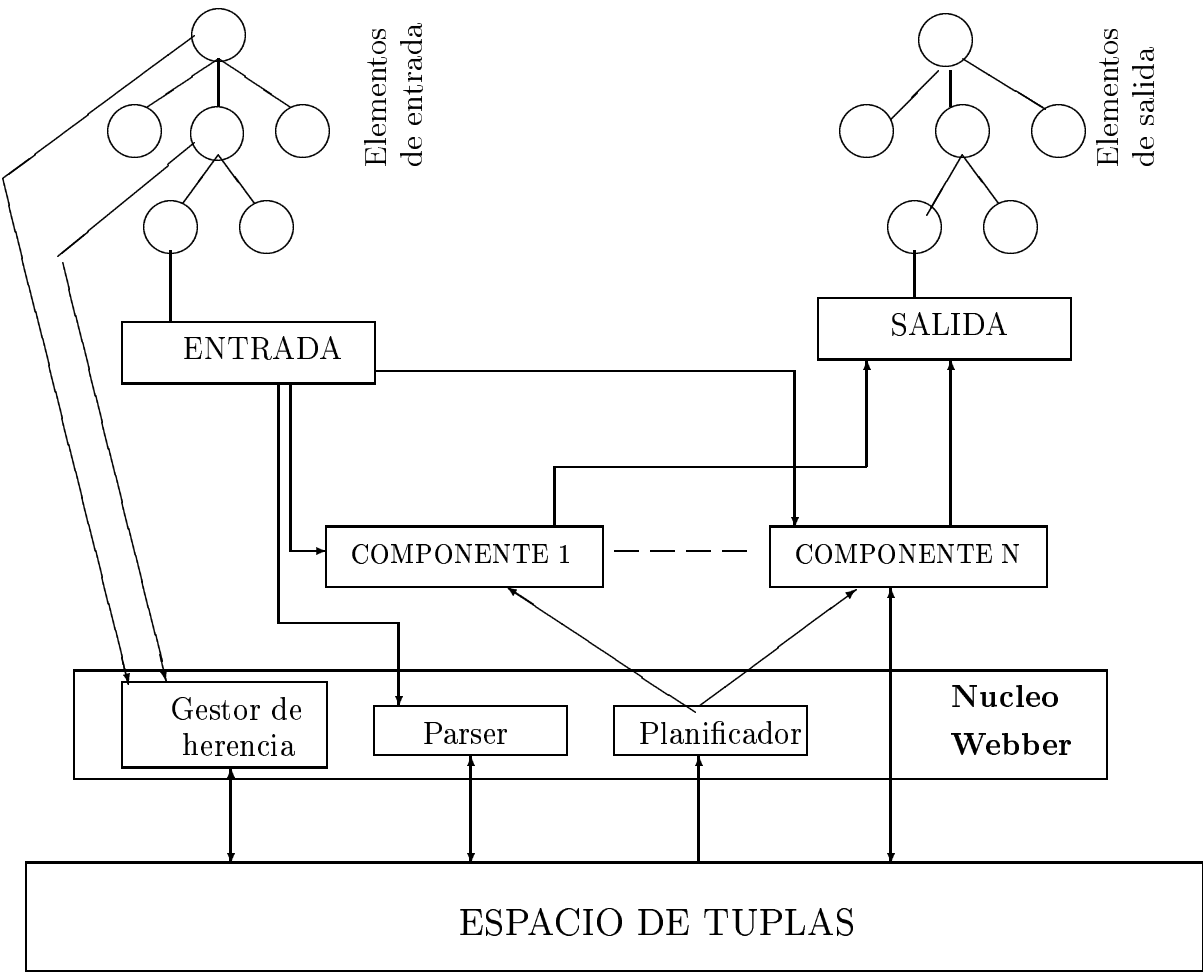


Figura 3.1: Modelo de ejecución de Webber

El entorno se basa en un espacio de datos compartidos (un espacio de *tuplas*) que permite la comunicación entre los componentes. El espacio de tuplas se organiza en variables, cada una de las cuales puede ser accedida de manera asociativa, y leída o escrita por los componentes en ejecución, de manera que éstos puedan dejar información para que sea usada por otros componentes. La ejecución del propio núcleo de Webber se controla por medio de alguna de estas variables. Como es lógico, las fuentes de información con las que trabajan los componentes se consideran asimismo variables en el espacio de tuplas.

Los valores iniciales de las variables del espacio de tuplas son asignados dentro de los elementos fuente o bien dentro de plantillas. A la hora de hablar de estos elementos fuente, conviene tener en cuenta que no son necesariamente las fuentes que se utilizan para construir los contenidos del servidor: un elemento fuente Webber puede contener en sus variables referencias a otras fuentes de datos, que son los que emplearán los componentes para producir la versión final de la información ofrecida por el servidor. Las plantillas ofrecen un mecanismo para establecer valores comunes para un conjunto de elementos fuente determinado. Puede definirse uno de estos plantillas para cada una de las localizaciones (directorios) que contenga el servidor.

Una de las características más importantes de Webber es que ofrece mecanismos de *herencia*, tanto a lo largo de los URL como por medio de declaraciones explícitas. En el primer caso, los valores de las variables que han sido fijados en una plantilla en un nivel superior, se mantienen en los niveles hijos si no son reescritos. En el segundo, es posible utilizar los valores definidos en algún otro elemento fuente mediante una referencia al mismo. De esta manera, se pueden aplicar valores por defecto a todos los contenidos del servidor (o un subconjunto determinado), así como a *clases* específicas de contenidos.

Estos mecanismos de herencia no sólo tienen su aplicación en las cuestiones de representación de la información, como puede ser el caso de enlaces para facilitar la navegación dentro del servidor o el uso de formularios comunes. Es importante señalar también que buena parte de la metainformación puede ser derivada jerárquicamente a través del URL (datos como el autor o el área de conocimiento suelen ser comunes a una determinada rama del servidor) o corresponder con una determinada clase de elementos. En definitiva, una vez se dispone de un conjunto de componentes adecuados, Weber ofrece una interface de programación completamente *declarativo*, de manera que los gestores de la información pueden concentrarse en los modelos de datos

y metadatos del servidor.

El núcleo de Webber está constituido por tres componentes *privilegiados* cuyas tareas comprenden el cargar los valores iniciales de las variables en el espacio de tuplas, ofrecer acceso a las variables, mantener la herencia a través de los diferentes URLs y clases, e instanciar y ejecutar los diferentes componentes encargados de procesar la información. Los únicos requisitos que Webber impone sobre estos componentes de proceso (procesadores) son relativos a su interface. Esta interface debe estar escrito en Perl (como lo está el núcleo de Webber) y los procesadores deben acceder a las variables del espacio de tuplas a través de las funciones que ofrece el núcleo de Webber, de manera que se preserven las propiedades que Webber mantiene para ellas. Por tanto, un procesador Webber puede ser desde un procedimiento muy simple (básicamente un conjunto de llamadas a la función **print** de Perl que escriben el valor de una variable), a un interace específico que conecte con otros programas tan complejos como sea necesario.

3.2.1 El núcleo

Los tres componentes que conforman el núcleo de Webber son los encargados de dirigir la ejecución del programa y de guiar el resto de componentes para que colaboren en la producción de los contenidos. De acuerdo con el orden en el que toman control de la ejecución de Webber, estos componentes son:

- El **parser** lee las plantillas y elementos fuente para extraer los valores de las variables y asignarlos en el espacio de tuplas, además de ejecutar las llamadas al sistema o instrucciones Perl que pueden incluir determinados elementos de los ficheros fuente o plantilla. Este es el primer componente invocado y es el encargado de generar una lista de árboles (uno por cada fichero plantilla o fuente) a partir de los ficheros que procesa a partir del fuente. Esta es la lista que después pasará al gestor de herencia para que éste construya el espacio de tuplas. Si, por algún motivo, los elementos que se están empleando como fuentes Webber no se corresponden con la sintaxis Webber (por ejemplo, se están empleando ficheros escritos en HTML para su transformación) es posible evitar el uso del parser, permitiendo así que otro componente se encargue de analizar el elemento fuente e incorporar los valores adecuados al espacio de tuplas.
- El **gestor de herencia**. Este componente se encarga de instanciar las variables dentro del espacio de tuplas definidos por la localización

del elemento (herencia a través del URL) y por la(s) clase(s) a la que pertenece.

- **El planificador** se encarga de identificar, cargar y ejecutar los componentes que deben ser aplicados a un determinado elemento fuente. Este componente se encarga de pasar el control a los procesadores y de ofrecerles el interface de acceso a las variables incluidas en el espacio de tuplas, garantizando el cumplimiento de las restricciones que se hayan impuesto sobre las mismas.

3.2.2 Variables reservadas

Como ya hemos dicho, el comportamiento de los componentes del núcleo, como el de todos los componentes ejecutados por Webber, es controlado por medio de variables en el espacio de tuplas. Las variables que definen el comportamiento del núcleo constituyen, como es lógico, un conjunto de variables *reservadas* y sus identificadores comparten el prefijo común **wbb**. Estas variables son las siguientes:

- **wbbTemplateName** es el nombre de la plantilla aplicable a cada localización.
- **wbbSourceRegExp** es una expresión regular que define qué elementos van a ser considerados como fuentes por Webber para una determinada localización. Esto permite el que los árboles fuente y destino sean el mismo y evita el procesamiento de elementos que no sean fuentes Webber, como puedan ser imágenes o datos en formato binario.
- **wbbTarget** y **wbbTargetName** definen los nombres relativo (a la localización actual) y absoluto del elemento destino que se está generando.
- **wbbLang** se utiliza para almacenar un identificador de dos letras del lenguaje del elemento destino. Aparte de su uso para asignación de metainformación, puede ser empleado también con propósitos de organizar la información, como puede ser el incluir enlaces a versiones del documento en otros lenguajes, o incluir diferentes versiones de elementos de navegación.
- **wbbLangs** Almacena pares clave-valor, donde cada clave identifica un idioma conocido por Webber y el valor es una descripción más extensa que suministra a los procesadores.

- **wbbDate** contiene la fecha de creación del elemento destino en el formato *AÑO MES DÍA*
- **wbbProcLib** define las localizaciones (directorios) donde pueden encontrarse los procesadores.
- **wbbClassLib** define las localizaciones (directorios) donde pueden encontrarse definiciones de clases para los elementos fuente.
- **wbbLib** define los procesadores que deben aplicarse para generar el elemento destino. Las características de estos procesadores son el objetivo del siguiente apartado.
- **wbbSourceRoot** define el directorio raíz del árbol de directorios fuente. Su valor determina el punto de comienzo de la herencia de variables. Por ejemplo, si establecemos su valor a `/wbbsrc` y procesamos un fichero llamado `/wbbsrc/docs/tech/report1`, se le aplicará la definición de variables hecha en las plantillas de `/wbbsrc/docs` y `/wbbsrc/docs/tech`.
- **wbbTargetRoot** define el directorio raíz del árbol de directorios destino. Se usa para determinar el nombre del fichero en el cual los procesadores escribirán su salida. Cuando se llama al Webber con la opción `-i` los procesadores deberán escribir en la salida estándar.
- **wbbVersion** contiene la versión de Webber.
- **wbbFileNameRegExp** es una expresión regular utilizada por Webber para formar el nombre del fichero destino. Esta expresión es aplicada al nombre del fichero fuente y el resultado de la coincidencia del primer paréntesis será concatenado con la actual extensión de los ficheros destino.
- **wbbFileLangRegExp** es una expresión regular utilizada por Webber para extraer a partir del nombre del fichero fuente el identificador del idioma que se va aplicar, utilizando el mismo mecanismo que en la variable anterior.
- **wbbExtension** define la extensión que será utilizada cuando se crea el nombre del fichero destino, se concatenará al nombre extraído mediante la variable **wbbFileNameRegExp**.
- **wbbTargetFileMode** define el modo en el que serán creados los ficheros por Webber. Debe ser una cadena numérica con el formato aceptado por la función Perl *chmod*. Si esta variable no se establece, su valor por defecto será `'0444'` (permiso de sólo lectura para todos los usuarios).

- **wbbExtParser** le dice a Webber que no parsee los ficheros fuente. Si su valor es distinto de 0, Webber no leerá las variables de los ficheros fuente. Un parser externo (normalmente uno o varios procesadores) se encargará de leer dichas variables.

3.2.3 Procesadores

Ya ha quedado dicho que los componentes de proceso (los procesadores) constituyen la esencia de las capacidades de tratamiento de la información del entorno Webber. Los únicos requisitos que un procesador debe cumplir son:

- El planificador debe poder invocarlo a través de métodos Perl accesibles a través de un módulo Perl. Es importante notar aquí que esto no implica restricción alguna sobre el lenguaje que se ha usado para codificar el procesador, que puede ser cualquiera. Únicamente el interface de invocación por parte del planificador debe estar escrito en Perl.
- Las variables en el espacio de tuplas deben ser accedidas usando uno de los dos métodos que ofrece Webber: un árbol DOM o un array asociativo Perl enlazado (de manera transparente) a ese árbol. En general, la mayor parte de los procesadores utilizarán el array asociativo, dado que el interface que ofrece es mucho más sencillo que el del árbol DOM. Sin embargo, dado que Webber almacena el espacio de tuplas en forma de una árbol DOM, se ha optado por ofrecer también un acceso directo a dicho formato.

El planificador carga los módulos Perl que contienen los procesadores por medio de sentencias **require** de Perl, por lo que las referencias a los procesadores dentro de un elemento fuente deben hacerse en la forma:

`NombreDeModulo::NombreDeMetodo`

De la misma manera, la extensión estándar para los módulos Perl, ".pm", debe ser empleada para nombrar los ficheros que contienen interfaces de procesadores Webber. Los procesadores son invocados secuencialmente por el planificador, de acuerdo con el orden en el que aparecen incluidos en la variable del núcleo **wbbLib**. Cualquier cosa que un procesador escriba en su salida estándar es automáticamente redirigida hacia el elemento fuente, por lo que los procesadores más sencillos pueden consistir en una serie de sentencias **print**, que tienen en cuenta el valor de algunas variables en el espacio de tuplas.

3.2.4 Procesadores estándar

Uno de los requisitos básicos para que un entorno de este tipo, basado en componentes, sea aplicable es proporcionar junto con sus elementos básicos un conjunto suficiente de componentes. De esta manera, se cubren dos objetivos fundamentales. Por un lado, se facilita el uso del entorno "tal como es", sin requerir que sus potenciales nuevos usuarios comiencen a escribir componentes personalizados desde el principio. Por otro, se proporcionan ejemplos prácticos de cómo deben codificarse los componentes. La distribución actual de Webber incluye un conjunto de procesadores que se detallan a continuación:

- **PrintIn:** Este es el procesador más simple. Lo único que hace es pasar el contenido de la variable `wbbIn` a la variable `wbbOut`.
- **BodyFaq:** Este procesador compone una página HTML con una tabla de contenidos al comienzo de dicha página. El formato de la plantilla puede estar definida en un fichero plantilla. La tabla de contenidos puede ser multinivel. Cualquiera de la entradas de la tabla de contenidos es un hiperenlace referenciando al comienzo del texto correspondiente en la página.
- **DumpVars:** Este procesador almacena en la variable `wbbOut` los actuales contenidos de las variables que se seleccionen. Puede ser de gran ayuda a la hora de la depuración.
- **Exec:** Asigna como valores de variables Webber el resultado de la ejecución de programas externos.
- **PgpSign:** Este procesador firma la página utilizando PGP[13]. La firma está oculta en un comentario HTML de manera que los usuarios no verán nada especial en la página, pero pueden descargar y chequear la firma. Éste debe ser el último procesador que modifique los contenidos de la página.
- **Slide:** Construye y enlaza un conjunto de página como si una serie de transparencias se tratasen. Los nombres de los ficheros destino tienen la siguiente estructura:

<PREFIX>N<POSTFIX>

Donde <PREFIX> y <POSTFIX> son cadenas arbitrarias, sin números dentro de ellas, y N es el número de transparencia. Las cadenas

<PREFIX> y <POSTFIX> deben ser las mismas para todas las transparencias. Se asume que todas las transparencias están situadas en el mismo directorio.

- **Webbo:** Sustituye en la variable `wbbOut` las etiquetas de la forma `<var name="WEBBER-VARIABLE"/>` por el valor de la variable identificada por la etiqueta. Si `wbbOut` está vacía, primero copia el valor de la variable `wbbIn` en ella. **Webbo** debe ser usado como uno de los primeros procesadores.
- **Caparse:** Parsea ficheros fuente en HTML/XML y extrae de ellos los valores de las variables de Webber.
- **Maketoc:** Antepone una tabla de contenidos en una página HTML.
- **Capaweb:** Extrae variables Webber de datos etiquetados (HTML, XML, ...) usando expresiones regulares.

Aparte de estos procesadores de carácter general, la distribución de Webber incluye también otros componentes especialmente diseñados para servidores concretos, de manera que cualquier administrador de un servidor pueda tomarlos como ejemplo para diseñar sus procesadores de propósito específico.

Capítulo 4

El núcleo de Webber. Diseño

4.1 Introducción

El núcleo de Webber se puede dividir en tres componentes claramente diferenciadas en cuanto a su funcionalidad y en cuanto al momento de su participación en la ejecución de esta herramienta. Estos componentes, por orden de ejecución son: el **Parser**, el **Gestor de herencia** y el **Planificador**. Además de los procedimientos y funciones que los componen, Webber también posee otras funciones para realizar tareas específicas comunes a estos tres elementos. Por citar algún ejemplo de este tipo de actividades podemos nombrar la función `LeerTemplates`, que es la encargada de procesar en el orden adecuado todos los ficheros plantilla heredados a partir del directorio donde se está procesando un fichero fuente, hasta llegar al directorio fuente o la raíz de la estructura de directorios de la máquina.

El **Parser** recibe como entrada un fichero o una cadena -en el caso de que se haya utilizado la entrada estándar para suministrar datos-, y se encarga de construir el árbol DOM correspondiente a éste y a todos los posibles ficheros de los que pueda heredar explícitamente. Al parsear la entrada verifica que no hay errores sintácticos, y además en aquellos nodos del árbol donde encuentra una instrucción de procesamiento, la ejecuta y sitúa en su lugar el resultado de su correspondiente ejecución. En el caso de que encontrase una instrucción de procesamiento no válida, es decir, no perteneciente a ninguno de los tres tipos soportados se encargará de eliminarla del documento para evitar posteriores procesamientos inútiles de ese nodo.

Como hemos dicho el parser procesa todas las clases a que pertenece la entrada debido a la herencia explícita, por lo tanto, después de ejecutar este

componente de Webber no obtendremos un árbol DOM, sino que obtendremos una array compuesto por todos los árboles DOM correspondientes a cada una de las clases. El mecanismo de herencia y el orden de los árboles del array resultante se aplican mediante las siguientes reglas:

Supongamos que A hereda explícitamente a B y a su vez B hereda a C:

- El Parser después de realizar su tarea con el fichero A nos devolverá un array de árboles DOM.
- Si A', B' y C' son los árboles DOM contruidos por el Parser correspondientes respectivamente a A, B y C, el array resultante tendrá el siguiente orden: C', B' y en último lugar A'.
- Esto significa que cuando el gestor de herencia procese esta lista ordenada de árboles, el último árbol en ser tratado es el de A, por lo que éste tendrá prioridad sobre el resto, que han sido heredados por él.

El **Gestor de Herencia** recibe a su entrada una lista ordenada de árboles compuesta por las estructuras DOM resultantes de parsear todas las clases a las que pertenece el elemento de entrada, tanto de manera explícita como por medio de la herencia a través de la estructura de directorios. El orden que sigue esta lista es el siguiente: Cuanto más arriba de la estructura de directorios (más próximo al directorio fuente establecido en la variable *wbbSourceRoot*) se encuentre el elemento fuente más al principio de la lista estará, y si éste hereda explícitamente otros ficheros, éstos irán en el orden que se ha indicado en el apartado anterior del parser con respecto a él.

La función del gestor de herencia es obtener a partir de esa lista de árboles, donde en último lugar y con la mayor preferencia se encuentra el fichero o cadena de entrada, un sólo árbol sobre el que definitivamente trabajará el resto de componentes de Webber. Éste árbol se calcula como combinación de todo el conjunto de árboles que componen la lista que recibe y cuyo orden de procesado es de dos en dos empezando por el principio de la lista. Supongamos que el gestor de herencia recibe la siguiente lista de árboles DOM: A, B, C y D. El orden de procesamiento es: primero se combinan A y B, el resultado de ambos se procesa con C y el resultado de esta combinación se procesa junto con D. El árbol resultante final es el que devuelve el gestor de herencia y quedará residente en memoria constituyendo el espacio de tuplas hasta que finalice la ejecución de Webber para ese elemento de entrada, ya que todos sus componentes deben acceder a él para su correcto funcionamiento.

La combinación de dos árboles se basa en la combinación de los nodos que los constituyen y que representan a las variables. Los nodos en un documento XML de entrada al Webber pueden tener los siguientes modos, que se indican en su atributo `mode`:

- **Readonly:** Este modo indica que el nodo es de sólo lectura, por lo que su valor no puede ser modificado ni tampoco puede ser eliminado.
- **Delete:** Si nos encontramos con una variable el atributo `mode` a este valor nos está indicando que al combinarse con otro árbol, si éste posee esta misma variable con un determinado valor, y no es de sólo lectura, entonces será eliminada.
- **Overwrite:** El nodo que posee este modo sobrescribirá su valor actual con el que existe en el árbol actualmente. Este es el modo por defecto para Webber.
- **Append:** Se concatenará el valor del nodo al final del valor actual de esa variable en el árbol. En el caso de que no existiese se crea nueva con ese mismo valor.
- **Prepend:** Se concatenará el valor del nodo al principio del valor actual de esa variable en el árbol. En el caso de que no existiese se crea nueva con ese mismo valor.

El **Planificador** aparece una vez que se dispone del árbol de variables definitivo cargado en memoria en formato DOM. Su función es la de localizar y ejecutar en orden los procesadores que se encargarán de producir el elemento de salida. Para ello tendrá que tener en cuenta el valor de la variable `wbbLib`, la cual contiene la lista de tales procesadores, además de posibles instrucciones que pueden aparecer intercaladas en dicha lista y que también deben ser ejecutadas en orden.

4.2 Formato de las plantillas y elementos fuente

El lenguaje elegido para la implementación de los ficheros plantilla y/o fuente de Webber ha sido XML y la estructura que se debe seguir en cada uno de ellos es la siguiente:

```

< webber inherit=''class'' >
  < variable1 >
    [CONTENIDO DE LA VARIABLE1]
  < /variable1 >
  < variable2 >
    [CONTENIDO DE LA VARIABLE2]
  < /variable2 >
  .....
< /webber >

```

El elemento principal y raíz del documento es denominado **webber** y puede poseer opcionalmente un atributo llamado **inherit**. A este atributo se le asigna una cadena que hace referencia a una plantilla situado en cualquier lugar de la estructura de directorios donde se ejecute Webber y debe seguir este mismo formato. Con este atributo estamos utilizando la herencia explícita que soporta Webber, es decir, el elemento hereda, además de las plantillas heredadas debido al recorrido descendente de la estructura de directorios, la plantilla aquí referenciada y todas las que a su vez éste pueda heredar de manera explícita.

Dentro del elemento XML llamado **webber**, aparecen una serie de elementos que hacen referencia a cada una de las variables que hacen posible el correcto funcionamiento de Webber. A continuación describiremos las variables del núcleo y sus correspondientes estructuras, ya que dependiendo de los posibles valores que puedan tomar su estructura será diferente:

- **wbbSourceRoot:** Este directorio es la raíz del árbol de directorios fuente de Webber. Su valor determina el punto de comienzo para la herencia de variables. Las variables del fichero que está siendo procesado en un directorio determinado van heredando los valores que van teniendo en los ficheros plantilla que nos encontremos desde ese directorio al indicado por **wbbSourceRoot**. Por ejemplo, si el directorio raíz de los ficheros fuente es **/wbbsrc** y estamos procesando un fichero denominado **/wbbsrc/docs/tech/report1**, entonces la definición de variables realizada en las plantillas de **/wbbsrc/docs** y **/wbbsrc/docs/tech** serán aplicadas. Si el Webber es llamado con la opción **-i**, no serán utilizados los ficheros plantilla.

Ejemplo:

```
<wbbSourceRoot>
```

```

    /home/manolo/webber
</wbbSourceRoot>

```

- **wbbTargetRoot:** Es el directorio raíz del árbol de directorios destino de Webber. Se usa para determinar cual será el nombre del fichero en el cual los procesadores de Webber escribirán su salida, es decir, el fichero que contendrá la página web resultante de la ejecución. En el directorio destino se crearán los directorios que hubiera en el fuente sólo en el caso de que ejecutemos a Webber con la opción `-m`. El directorio fuente y destino pueden ser el mismo. Cuando utilizamos la opción `-i` la salida de Webber es dirigida a la salida estándar.

Ejemplo:

```

<wbbTargetRoot>
    /home/manolo/webber
</wbbTargetRoot>

```

- **wbbLangs:** Esta variable contiene pares clave-valor, donde la clave identifica a un lenguaje y el valor suministra a los procesadores una descripción más detallada del lenguaje con el que esté asociado.

Ejemplo:

```

<wbbLangs>
    <lang id="es"> Español </lang>
    <lang id="en"> Inglés </lang>
</wbbLangs>

```

- **wbbTemplateName:** Esta variable contiene el nombre de los ficheros plantilla de Webber de cualquier directorio. No es obligatorio tener un fichero plantilla por directorio, pero si existe su nombre debe ser idéntico al valor de esta variable.

Ejemplo:

```

<wbbTemplateName>
    .template
</wbbTemplateName>

```

- **wbbSourceRegExp**: Contiene una expresión regular que define qué elementos van a ser considerados como fuentes por Webber para una determinada localización. Esto permite que los árboles fuente y destino sean el mismo y evita el procesamiento de elementos que no sean fuentes Webber, como puedan ser imágenes o datos en formato binario. Se considerarán ficheros fuente aquellos que cumplen la expresión regular que contiene esta variable y que no son plantilla, es decir, no poseen el nombre indicado en la variable `wbbTemplateName`.

En el ejemplo vamos a considerar ficheros fuente aquellos que empiecen por el carácter ".":

```
<wbbSourceRegExp>
  ^\..*
</wbbSourceRegExp>
```

- **wbbLang** se utiliza para almacenar un identificador de dos letras del lenguaje del elemento destino. Aparte de su uso para asignación de metainformación, puede ser empleado también con propósitos de organizar la información, como puede ser el incluir enlaces a versiones del documento en otros lenguajes, o incluir diferentes versiones de elementos de navegación.

Ejemplo:

```
<wbbLang> es </wbbLang>
```

- **wbbDate** contiene la fecha de creación del elemento destino en el formato *AÑO MES DÍA*.

Ejemplo:

```
<wbbDate> 2001 07 17 </wbbDate>
```

- **wbbProcLib** define las localizaciones (directorios) donde pueden encontrarse los procesadores. Cada directorio que pueda contener procesadores y en el cual Webber deberá buscar cuando realiza una ejecución debe estar situado dentro de un elemento independiente incluido en el elemento `wbbProcLib` que es el que define esta variable. En el ejemplo

siguiente se han elegido como nombres para esos elementos `dir` que es lo que marca el DTD asociado a los ficheros XML de Webber.

Ejemplo:

```
<wbbProcLib>
  <dir>
    /home/manolo/PROYECTO/WEBBER/proc
  </dir>
  <dir>
    /home/manolo/PROYECTO/lib
  </dir>
</wbbProcLib>
```

- **wbbClassLib** define las localizaciones (directorios) donde pueden encontrarse definiciones de clases para los elementos fuente. Es decir, contiene los directorios donde Webber buscará la plantilla heredada cuando se realiza un herencia explícita en algún fichero mediante el atributo `inherit` del elemento raíz `webber`. Cada directorio que pueda contener ficheros de clase y en el cual Webber deberá buscar cuando se realiza tal herencia debe estar situado dentro de un elemento independiente incluido en el elemento `wbbClassLib` que es el que define esta variable. En el ejemplo siguiente se observa que el nombre para esos elementos es `dir`, porque es el que indica el DTD asociado a los ficheros XML de Webber.

Ejemplo:

```
<wbbClassLib>
  <dir>
    /home/manolo
  </dir>
  <dir>
    /home
  </dir>
  <dir>
    /home/manolo/PROYECTO/WEBBER/DIR
  </dir>
</wbbClassLib>
```

- **wbbLib** define los procesadores que se deben ejecutar para generar el elemento destino. Contiene una lista ordenada con los procesadores que el planificador deberá llamar para producir el resultado final, y manteniendo el orden en el que aparecen en esta variable. Además de los procesadores a ejecutar esta variable puede contener elementos denominados **inline** que agrupan instrucciones Perl que serán ejecutadas siguiendo el orden. Cada nombre de procesador irá separado por un retorno de carro. Pueden aparecer tantos elementos **inline** como se quiera y cada uno de ellos puede contener una o más instrucciones Perl. Ejemplo:

```
<wbbLib>
  Webbo::webbo
  PgpSign::pgpsign
  <inline>
    $a = $b + $c;
  </inline>
  PrintIn::printin
</wbbLib>
```

- **wbbVersion** contiene la versión de Webber.

Ejemplo:

```
<wbbVersion> webber v.2.0 </wbbVersion>
```

- **wbbFileNameRegExp** es una expresión regular utilizada por Webber para formar el nombre del fichero destino. Esta expresión es aplicada al nombre del fichero fuente y el resultado de la coincidencia del primer paréntesis, es decir, el valor de la variable Perl **\$1**, será concatenado con la actual extensión de los ficheros destino.

En este ejemplo si el fichero fuente se llama **.index**, el resultado de aplicarle esta expresión regular es **index**.

```
<wbbFileNameRegExp>
  ^\.(.*)$
</wbbFileNameRegExp>
```

- **wbbFileLangRegExp** es una expresión regular utilizada por Webber para extraer a partir del nombre del fichero fuente el identificador del idioma que se va aplicar, utilizando el mismo mecanismo que en la variable anterior.

En este ejemplo si el fichero fuente se llama `.index.fr`, el resultado de aplicarle esta expresión regular es `fr`.

```
<wbbFileLangRegExp>
  .+\.([a-z][a-z])$
</wbbFileLangRegExp>
```

- **wbbExtension** define la extensión que será utilizada cuando se crea el nombre del fichero destino, se concatenará al nombre extraído mediante la variable `wbbFileNameRegExp`.

Ejemplo:

```
<wbbExtension>
  .html
</wbbExtension>
```

- **wbbTarget** y **wbbTargetName** definen los nombres relativos (a la localización actual) y absoluto del elemento destino que se está generando. Son utilizadas para determinar el nombre del fichero destino que se calcula de la siguiente forma:
 - Si la variable `wbbTargetName` tiene valor, ese será el nombre del fichero destino.
 - Si no tiene valor el nombre del fichero destino será `wbbTarget.idioma.wbbExtension`, donde `wbbTarget` y `wbbExtension` son los valores de dichas variables e idioma es el valor del idioma calculado para el fichero fuente.
 - En caso de que ninguna de estas dos variables tenga valor el nombre del fichero destino será el resultado de aplicarle al nombre del fichero fuente la expresión regular que contiene la variable `wbbFileNameRegExp` concatenándole el idioma y el valor de `wbbExtension`.

- **wbbTargetFileMode** define el modo en el que serán creados los ficheros por Webber. Debe ser una cadena numérica con el formato aceptado por la función *chmod*. Si esta variable no se establece, su valor por defecto será '0444' (permiso de sólo lectura para todos los usuarios). Esto sólo tendrá sentido en sistemas operativos que lo soporten (no en Windows, por ejemplo)

Ejemplo:

```
<wbbTargetFileMode> 0444 </wbbTargetFileMode>
```

- **wbbExtParser** le dice a Webber que no parsee los ficheros fuente. Si su valor es distinto de 0, Webber no leerá las variables de los ficheros fuente. Un parser externo (normalmente será uno o varios procesadores) se encargará de leer dichas variables.

Ejemplo:

```
<wbbExtParser> 1 </wbbExtParser>
```

- **wbbIn** almacena el contenido principal de información del fichero fuente. Los procesadores realizan un procesamiento especial con esta variable, ya que en ella esperan la mayoría del contenido de información de la página resultante.
- **wbbOut** define el contenido principal del destino. De nuevo tratamos con un concepto algo difuso, donde lo único que se podría destacar es que esta variable también tiene un tratamiento especial para Webber, ya que el resultado final de la ejecución será almacenado por los procesadores en esta variable, que finalmente, Webber volcará en el fichero de salida.

Si el contenido de alguna variable posee código HTML o XML, para que no sea confundido con la sintaxis del fichero fuente o plantilla que la contenga, debemos encerrar su contenido dentro de un elemento **CDATA** de XML. El parser de XML que utilizamos del módulo `Perl XML::DOM` considera el contenido de un elemento **CDATA** como si de un nodo de texto se tratara, sin interpretar las posibles etiquetas que pueda contener.

Veamos un ejemplo para la estructura de una variable que posee ese contenido especial dentro de un nodo `CDATA`:

```
<VARIABLE>
  <![CDATA[
    <p> Esto es un párrafo cualquiera. </p>
  ]]>
</VARIABLE>
```

4.2.1 Instrucciones de procesamiento

En Webber podemos utilizar las instrucciones de procesamiento de XML para asignarle un valor a una variable. Estas instrucciones son nodos que poseen una sintaxis especial y que representan la llamada a un programa externo a Webber, y el resultado de su ejecución será almacenado como el valor de la variable que la contiene. La estructura que poseen es la siguiente:

```
<variable>

<?TIPO INSTRUCCIÓN?>

</variable>
```

Donde `TIPO` es el tipo de instrucción de procesamiento que queremos construir, y de él dependerá los posibles valores para el parámetro `INSTRUCCIÓN`. La clasificación de los tres tipos que podemos utilizar es:

- **perl:** En este caso el parámetro `INSTRUCCIÓN` debe ser una llamada a una función Perl. Por ejemplo, supongamos que tenemos la función `imprimeCadenaTexto()` que devuelve la cadena `abcdefghijklm`, entonces si en un fichero fuente de Webber tenemos un nodo tal como este:

```
<variable>
  <?perl imprimeCadenaTexto()?>
</variable>
```

cuando sea parseado el nodo pasará a tener la siguiente estructura:

```
<variable>

abcdefghijklm

</variable>
```

- **eval:** En este caso el parámetro INSTRUCCIÓN debe ser un parámetro adecuado para la función `eval` de Perl, la cual debe recibir una instrucción de este language en una cadena de texto. Por ejemplo, supongamos que tenemos las cadenas Perl `$a` con valor "Hola" y `$b` con valor "Mundo". Si en un fichero fuente de Webber tenemos un nodo tal como este:

```
<variable>
  <?eval $a.$b?>
</variable>
```

cuando sea parseado el nodo pasará a tener la siguiente estructura:

```
<variable>
  HolaMundo
</variable>
```

- **system:** En este caso el parámetro INSTRUCCIÓN debe ser una llamada al sistema operativo. El resultado de ejecutar dicha llamada será colocado como valor de la variable que la contenga. Supongamos que ejecutamos a Webber situados en el directorio `/home/manolo/proyecto/webber/`, entonces dado el siguiente nodo:

```
<variable>
  <?system pwd?>
</variable>
```

cuando sea parseado pasará a tener la siguiente estructura:

```
<variable>
  /home/manolo/proyecto/webber/
</variable>
```

Es importante aclarar que todas las instrucciones de procesamiento de tipo llamada al sistema se ejecutarán en el directorio donde Webber fue invocado, independientemente del fichero, perteneciente a la lista de la herencia aplicable para el fichero de entrada, que la contenga.

4.2.2 Fichero de configuración de Webber

Para finalizar este apartado sólo nos queda hablar del fichero de configuración de Webber. Este fichero sigue exactamente la misma estructura que los ficheros fuente y plantilla, y puede contener igualmente instrucciones de procesamiento como los anteriores. La única diferencia y restricción que posee frente al resto es que no puede poseer herencia explícita mediante el uso del atributo `inherit` del elemento principal `webber`.

El nombre por defecto para este fichero es *webber.conf*, pero puede ser cambiado mediante el parámetro `-C` de Webber. Este fichero debe contener las variables que necesita Webber para empezar a trabajar. Un ejemplo de fichero de configuración podría ser el siguiente:

```
<webber>
  <wbbVersion>
Webber 2.0
  </wbbVersion>
  <wbbLangs>
  <lang id="es">Español</lang>
  <lang id="en">English</lang>
  </wbbLangs>
  <wbbFileNameRegExp>
^\.(\.*)$
  </wbbFileNameRegExp>
  <wbbFileLangRegExp>
.+\.([a-z][a-z])$
  </wbbFileLangRegExp>
  <wbbProcLib>
  <dir1>
/home/manolo/PROYECTO/WEBBER/proc
  </dir1>
  </wbbProcLib>
  <wbbClassLib>
  <dir1>
```

```

/home/manolo
</dir1>
</wbbClassLib>
<wbbSourceRegExp>
^\. .*
</wbbSourceRegExp>
<wbbTemplateName>
.template
</wbbTemplateName>
<wbbSourceRoot>
/home/manolo/PROYECTO/
</wbbSourceRoot>
<wbbTargetRoot>
/home/manolo/PROYECTO/WEBBER/
</wbbTargetRoot>
<wbbExtension>
.html
</wbbExtension>
<wbbLib>
Iris2::cab
PrintIn::printIn
Iris::pdp
</wbbLib>
<iconDir>
/home/manolo/PROYECTO/WEBBER/icon/
</iconDir>
</webber>

```

Por último simplemente comentar que el árbol DOM generado al parsear el fichero de configuración siempre va a ser el primer árbol de la lista de árboles que reciba el gestor de herencia, es decir, será el árbol menos prioritario de todos los que herede el fichero de entrada a Webber.

4.2.3 DTD para los ficheros fuente Webber

A continuación se detalla la DTD correspondiente a los ficheros XML fuente y plantilla que utilizan las variables del núcleo de Webber:

```

<!DOCTYPE webber[
  <!ELEMENT webber (wbbSourceRoot | wbbTargetRoot | wbbLangs |
                    wbbTemplateName | wbbSourceRegExp | wbbTarget |

```

```

        wbbTargetName | wbbLang | wbbDate | wbbProcLib |
        wbbClassLib | wbbLib | wbbVersion | wbbOut |
        wbbFileLangRegExp | wbbFileNameRegExp | wbbIn |
        wbbExtension | wbbTargetFileMode | wbbExtParser)>
<|ELEMENT wbbSourceRoot (#PCDATA)>
<|ELEMENT wbbTargetRoot (#PCDATA)>
<|ELEMENT wbbTemplateName (#PCDATA)>
<|ELEMENT wbbSourceRegExp (#PCDATA)>
<|ELEMENT wbbTarget (#PCDATA)>
<|ELEMENT wbbTargetName (#PCDATA)>
<|ELEMENT wbbLang (#PCDATA)>
<|ELEMENT wbbDate (#PCDATA)>
<|ELEMENT wbbVersion (#PCDATA)>
<|ELEMENT wbbFileNameRegExp (#PCDATA)>
<|ELEMENT wbbFileLangRegExp (#PCDATA)>
<|ELEMENT wbbExtension (#PCDATA)>
<|ELEMENT wbbTargetFileMode (#PCDATA)>
<|ELEMENT wbbExtParser (#PCDATA)>
<|ELEMENT wbbIn (#PCDATA)>
<|ELEMENT wbbOut (#PCDATA)>
<|ELEMENT wbbLangs (lang+)>
<|ELEMENT lang (#PCDATA)>
<|ELEMENT wbbProcLib (dir+)>
<|ELEMENT dir (#PCDATA)>
<|ELEMENT wbbClassLib (dir+)>
<|ELEMENT wbbLib (#PCDATA | inline)+>
<|ELEMENT inline (#PCDATA)>
<|ATTLIST webber inherit "">
<|ATTLIST wbbSourceRoot mode (OVERWRITE|APPEND|PREPEND|
                             DELETE|READONLY) "OVERWRITE">
<|ATTLIST wbbTargetRoot mode (OVERWRITE|APPEND|PREPEND|
                             DELETE|READONLY) "OVERWRITE">
<|ATTLIST wbbLangs mode (OVERWRITE|APPEND|PREPEND|
                        DELETE|READONLY) "OVERWRITE">
<|ATTLIST wbbTemplateName mode (OVERWRITE|APPEND|PREPEND|
                                DELETE|READONLY) "OVERWRITE">
<|ATTLIST wbbSourceRegExp mode (OVERWRITE|APPEND|PREPEND|
                                DELETE|READONLY) "OVERWRITE">
<|ATTLIST wbbTarget mode (OVERWRITE|APPEND|PREPEND|
                          DELETE|READONLY) "OVERWRITE">
<|ATTLIST wbbTargetName mode (OVERWRITE|APPEND|PREPEND|

```

```

                                DELETE|READONLY) "OVERWRITE">
<!ATTLIST wbbLang mode (OVERWRITE|APPEND|PREPEND|
                                DELETE|READONLY) "OVERWRITE">
<!ATTLIST wbbDate mode (OVERWRITE|APPEND|PREPEND|
                                DELETE|READONLY) "OVERWRITE">
<!ATTLIST wbbProcLib mode (OVERWRITE|APPEND|PREPEND|
                                DELETE|READONLY) "OVERWRITE">
<!ATTLIST wbbClassLib mode (OVERWRITE|APPEND|PREPEND|
                                DELETE|READONLY) "OVERWRITE">
<!ATTLIST wbbLib mode (OVERWRITE|APPEND|PREPEND|
                                DELETE|READONLY) "OVERWRITE">
<!ATTLIST wbbVersion mode (OVERWRITE|APPEND|PREPEND|
                                DELETE|READONLY) "OVERWRITE">
<!ATTLIST wbbFileNameRegExp mode (OVERWRITE|APPEND|PREPEND|
                                DELETE|READONLY) "OVERWRITE">
<!ATTLIST wbbFileLangRegExp mode (OVERWRITE|APPEND|PREPEND|
                                DELETE|READONLY) "OVERWRITE">
<!ATTLIST wbbExtension mode (OVERWRITE|APPEND|PREPEND|
                                DELETE|READONLY) "OVERWRITE">
<!ATTLIST wbbTargetFileMode mode (OVERWRITE|APPEND|PREPEND|
                                DELETE|READONLY) "OVERWRITE">
<!ATTLIST wbbExtParser mode (OVERWRITE|APPEND|PREPEND|
                                DELETE|READONLY) "OVERWRITE">
<!ATTLIST wbbIn mode (OVERWRITE|APPEND|PREPEND|
                                DELETE|READONLY) "OVERWRITE">
<!ATTLIST wbbOut mode (OVERWRITE|APPEND|PREPEND|
                                DELETE|READONLY) "OVERWRITE">
]>

```

4.3 Modos de ejecución de Webber

Una vez que se ha instalado Webber, éste puede ser invocado desde el shell de dos posibles formas:

```
$ webber Fichero
```

o bien

```
$ webber Directorio
```

En ambos casos, Webber se ejecutará utilizando el fichero de configuración por defecto y en el primer caso procesará el fichero fuente identificado por el parámetro `Fichero` y en el segundo caso procesará todos los ficheros contenidos en el directorio identificado por el parámetro `Directorio`.

También podemos utilizar varios nombres de ficheros y directorios cuando llamamos a Webber, en el siguiente ejemplo:

```
$ webber F1 D1 F2
```

procesará el fichero `F1`, después todos los ficheros contenidos en el directorio `D1` y por último el fichero `F2`.

Webber además acepta una serie de argumentos que controlan la forma en la que debe procesar los ficheros. Estos argumentos son:

- `-C` define un fichero de configuración alternativo al definido por defecto que es `webber.conf`.
- `-d` establece el modo de depuración en Webber. El programa mostrará información sobre su estado y sobre los ficheros fuente que está procesando.
- `-f` fuerza a actualizar los ficheros destino. Webber comprueba si la fecha de modificación del fichero destino es más reciente que la fecha de modificación del fichero fuente. Si esta opción no es utilizada, todos los ficheros fuente más antiguos que sus correspondientes ficheros destino no serán procesados.
- `-h` para mostrar un breve mensaje de ayuda sobre el funcionamiento de Webber y finalizar la ejecución.
- `-H` para mostrar la ayuda disponible sobre los procesadores actualmente conocidos por Webber, que serán todos aquellos situados en el mismo directorio que éste o en los directorios que contenga la variable de configuración llamada `wbbProcLib`.
- `-i` para hacer que Webber se ejecute utilizando la entrada estándar en lugar de procesar un fichero. Hay que tener en cuenta que en este caso ningún fichero plantilla será procesado.
- `-I` para mostrar la lista de los procesadores conocidos por Webber, además de una breve descripción de ellos.

- `-m` para que Webber cree los directorios destino que vaya necesitando a la hora de generar los ficheros de salida. Por defecto, Webber asume que los directorios destino ya han sido creados.
- `-r` para recorrer recursivamente los directorios fuente, empezando por el directorio indicado en la línea de comandos. Por defecto, si a Webber le indicamos que procese un directorio, sólo procesará los ficheros incluidos en él e ignorará los subdirectorios que pueda contener. Con esta opción también procesará todos los subdirectorios que contenga el directorio fuente.

4.4 Componentes

4.4.1 El cuerpo principal o Main

El cuerpo principal de Webber está formado por las primeras instrucciones que se ejecutan antes de llamar a ninguno de los componentes del núcleo. Las tareas que se realizan aquí son las siguientes:

- Carga los módulos que Webber utiliza para su funcionamiento:
 - *Shell*: permite ejecutar comandos del sistema como si de funciones Perl se tratara, siguiendo su misma sintaxis.
 - *XML::DOM*: suministra el interface con los ficheros en XML (fuente y plantilla).
 - *Cwd*: suministra funciones para trabajar facilmente con directorios.
 - *Getopt::Std*: almacena un '1' en una variable de la forma `$opt_x` si el argumento llamado `x` ha sido suministrado a Webber al invocarlo, si además el parámetro puede llevar asociado un valor, (como ocurre con el argumento `-C` que lleva asociado el nombre del fichero de configuración), éste se escribirá en mayúsculas y su valor será almacenado en la variable anteriormente comentada. Esto facilita mucho el trabajo con los argumentos que le son suministrados a Webber por la línea de comandos.
 - *DomWrapper*: que ha sido creado para mantener los procesadores antiguos de Webber con el interface en Perl en lugar de en XML; se describirá detalladamente en un apartado posterior.
- Inicializa las variables `$opt_x`, donde `x` es el nombre de un posible argumento que puede recibir Webber.

- Si han llamado a Webber con el argumento `-h` invocará a la función `showHelp`, que no recibe ningún parámetro y que lo único que hace es mostrar una breve ayuda sobre los argumentos de Webber y finalizar la ejecución.
- Invoca al Parser para que realice su función sobre el fichero de configuración de Webber, de manera que éste ya quede residente en memoria en formato árbol de DOM, para que se puedan acceder a las variables necesarias para comenzar la ejecución.
- Si nos han solicitado ayuda o información sobre los procesadores conocidos por Webber invocará a la función `Procesadores` con el parámetro adecuado para que muestre la información exigida.
- Si se ha ejecutado a Webber con la opción `-i` llamará a la función `ProcesaSTDIN`, que será la encargada de procesar la entrada estándar.
- Por último, si ninguna de las opciones anteriores estaba activa, leerá cada uno de los nombres de fichero y/o directorios que se le suministren, y dependiendo de si se trata de una clase u otra llamará a la función `ProcesaFichero` o `ProcesaDirectorio`.

La función `Procesadores`

Recibe como parámetro una cadena cuyos dos valores posibles son **info** (para suministrar la información que se tenga sobre los procesadores) o **help** (para suministrar la ayuda que se posea sobre los procesadores).

Para realizar su tarea, esta función recorrerá todos los directorios incluidos en la variable `wbbProcLib` del fichero de configuración, y de todos y cada uno de los procesadores que localice mostrará el tipo de información que se le ha indicado por el parámetro que ha recibido, es decir, ayuda o información. Ésto consiste simplemente en ejecutar la función `info` o `help` del procesador, ya que éstos están implementados como módulos Perl que podrán o no incluir estas funciones. En el caso de que estas funciones no esten implementadas se suministrará un mensaje informando de tal hecho.

La función `ProcesaSTDIN`

Esta función va a realizar la misma tarea que las dos funciones que vienen a continuación pero en el caso especial de que Webber se ejecute con la opción `-i`, es decir, cuando la entrada y salida de Webber deben coincidir con la estándar.

Lo primero que hace es almacenar todo lo que llega por la entrada estándar en una cadena para poder suministrársela al Parser. Con la lista de árboles que le devuelve éste invoca al gestor de herencia. El siguiente paso es actualizar la variable `wbbDate` con la fecha adecuada y inicializar la variable `wbbbOut` a cadena vacía. Por último se ejecuta la función `LlamaProcesadores` que es el núcleo del Planificador.

La función `ProcesaFichero`

A continuación enumeramos las operaciones que esta función realiza:

- Lo primero que debe hacer esta función es obtener el nombre del fichero destino resultante de procesar el fuente que recibe como parámetro, para ello hace uso de las variables ya comentadas. Y comprueba que el nombre del fichero fuente cumple la expresión regular de la variable `wbbSourceRegExp` y no se trata de un fichero plantilla. Para averiguar el idioma que debe asignarle al fichero destino utiliza la función `DefinidoIdioma`, que lo único que hace es buscar en la variable `wbbLangs` si existe algún identificador para el idioma extraído del fichero fuente con la expresión regular contenida en la variable `wbbFileLangExpresion`.
- En segundo lugar llama a la función `LeerTemplates` que se encarga de obtener la lista ordenada de árboles heredados a través de la estructura jerárquica de directorios.
- En el caso de que la variable `wbbExtParser` no esté activa, llamará al Parser dándole como argumento el fichero de entrada y con el resultado obtenido y la lista de árboles anteriores confecciona la lista final que le suministrará al gestor de herencia.
- Actualiza los valores de las variables `wbbTarget`, `wbbSource` y `wbbOut`.
- El paso siguiente es comprobar si el fichero destino debe ser actualizado, es decir, si el fichero fuente ha sido modificado después de la fecha de creación del fichero destino o si el usuario ha utilizado el argumento `-f`; en ese caso se invoca al planificador y la salida de los procesadores se envía al fichero destino.
- Por último establece el modo para el fichero destino creado, utilizando para ello el valor de la variable `wbbTargetFileMode`.

La función `ProcesaDirectorio`

Esta función se invoca cuando el argumento de entrada es un directorio en lugar de ser un fichero, en cuyo caso hay que procesar todos los ficheros fuente incluídos en dicho directorio.

Lo primero que se hace es leer todo el contenido del directorio, teniendo en cuenta que no se procesarán enlaces simbólicos, ficheros que representan colas FIFO o sockets, ficheros especiales de bloques, etc. Sólo se tendrán en cuenta los ficheros de texto y los subdirectorios que se irán encolando para su posible tratamiento posterior (que dependerá si se invocó a Webber con la opción `-r`).

Cada vez que encuentra un fichero fuente válido, llama al Parser pasándole como argumento el fichero de configuración (esto se hace cada vez que se encuentra un fichero fuente porque el gestor de herencia modifica los árboles en memoria y así siempre se tiene el fichero de configuración original cargado), y después llama a la función `ProcesaFichero` pasándole como argumento el fichero fuente.

Si se invocó a Webber con la opción `-r`, esta función se llama a sí misma recursivamente por cada uno de los subdirectorios que pudiera contener y que se habían encolado previamente.

La función `LeerTemplates`

Esta función es la encargada de calcular la herencia implícita de ficheros plantilla del fichero que se le pasa como argumento. Con herencia implícita nos estamos refiriendo a la herencia de ficheros que posee un determinado fichero fuente debido a su ubicación en la estructura de directorios del servidor Web.

También va a recibir como argumento la lista donde queremos que nos devuelva en orden los ficheros plantilla que hayan sido heredados. Esta lista está compuesta por árboles con estructura DOM y a la cual se le añadirán el árbol resultante del fichero de configuración (al principio) y del fichero que se está procesando (al final), junto con los posibles ficheros heredados explícitamente con el atributo `inherit`.

Esta función parte del directorio donde se encuentra el fichero que se le pasa como argumento y va ascendiendo en la estructura de directorios del servidor Web hasta llegar al directorio indicado por la variable `wbbSourceRoot` o hasta llegar al directorio raíz `/`. Por cada directorio que se recorre busca si hay algún fichero plantilla, en cuyo caso invoca al Parser para él y lo añade a la lista de árboles DOM antes comentada. Cuanto más cerca esté un fichero plantilla del directorio `wbbSourceRoot` más al principio de la lista y las clases a las que pertenece (porque un fichero plantilla también puede pertenecer a una clase mediante el atributo `inherit`) estarán justo delante de él.

4.4.2 El Parser

Como ya hemos comentado en apartados anteriores, el Parser es el encargado de comprobar la sintaxis de los ficheros fuente y plantilla y de generar un árbol DOM por cada uno de ellos. Para ello utiliza el módulo `XML:DOM` que facilita el interface con ficheros XML y suministra funciones para la manipulación de los árboles que se generan. A continuación vemos las funciones que utiliza y profundizamos en las tareas que debe realizar.

La función Parser

Esta es la primera función del Parser que se invoca cuando queremos parsear un fichero. Lo primero que hace es generar un objeto Parser del módulo `XML:DOM` con el atributo `ProtocolEncoding` con el valor `ISO-8859-1` (para utilizar la codificación de caracteres utilizada en ISO-LATIN-1 de Europa occidental), para comprobar la sintaxis del fichero y generar el árbol DOM en memoria. Este árbol será el que utilice la función para leer o modificar el contenido del fichero.

Una vez construido el árbol, recorre todos y cada uno de los elementos que lo componen en busca de instrucciones de procesamiento que deberá ejecutar y consultar su resultado para asignar al nodo que las contenga. Para realizar esta tarea llama a la función `ParserNode` por cada nodo que se encuentra.

La siguiente tarea fundamental que debe realizar es adquirir el valor del atributo `inherit` para ver si el fichero posee herencia explícita sobre algún fichero plantilla. En el caso de que este atributo tenga valor y sólo se suministren el nombre del fichero heredado, habrá que localizarlo en el directorio actual o en los directorios contenidos en la variable `wbbClassLib`. Una vez localizado se invoca recursivamente esta función para realizar también el parseado de la clase.

Esta función recibe, además del fichero a parsear, un parámetro denominado `stdin` que le indica si ha sido utilizada o no la entrada estándar, ya que la función del módulo `XML::DOM` para parsear una cadena o un fichero es diferente.

También recibe como argumento una lista donde deberá situar el fichero parseado y todas las clases heredadas explícitamente. La clase siempre debe ir delante en la lista del que lo hereda, ya que tiene menor preferencia como se ha indicado en apartados anteriores.

La función `ParserNode`

Esta función recibe como argumentos una referencia al documento (siempre necesaria para añadirle nuevos elementos), una referencia al nodo que se va a examinar y una referencia al padre, ya que esto facilita en gran medida la eliminación y modificación de nodos hijos.

Va a ser la encargada de ejecutar las instrucciones de procesamiento encontradas en los nodos y situar en ellos el resultado de la ejecución; por lo tanto los únicos nodos que le van a interesar son los de tipo 7 según la especificación del DOM, es decir, los correspondientes a instrucciones de procesamiento. Si encuentra un nodo de tipo 1, es decir, un elemento que puede contener otros nodos a su vez, se llama recursivamente pasando como argumento ese nodo para profundizar en el árbol en esa dirección porque podrían aparecer nuevas instrucciones de procesamiento.

Una vez ha sido localizado una instrucción de procesamiento se determina a que tipo pertenece y según eso se llevará a cabo su ejecución de una manera o de otra:

- Si es de tipo `eval` su ejecución consiste en invocar a la función de Perl `eval` con el segundo argumento de la instrucción de procesamiento.
- Si es de tipo `system` su ejecución consiste en invocar a la función `CreaInstrucción` con el segundo argumento de la instrucción de procesamiento, que convertirá el comando del sistema suministrado en una función de Perl que se ejecutará ayudándose de la función `eval` de nuevo. Esto se consigue gracias a la inclusión del módulo `Shell` que nos permite ejecutar comandos del sistema como si de funciones Perl se tratarán.

- Si es de tipo **perl** su ejecución consiste en invocar a la función de Perl **eval** añadiéndole al segundo argumento de la instrucción de procesamiento el carácter **&** delante para indicar que se trata de una función y el carácter **;** al final.

El resultado de las ejecuciones se almacena en una variable de tipo cadena y su valor sustituirá al nodo de tipo 7 en el árbol del fichero que se está parseando. Para ello primero creamos un nuevo nodo de texto con su contenido en el árbol cuya referencia nos han pasado y después sustituimos el que contenía la instrucción de procesamiento por él, utilizando la función del módulo DOM denominada **ReplaceChild** que realiza esta tarea.

La función **CreaInstruccion**

Como hemos comentado antes esta función transforma la sintaxis de un comando del sistema en la sintaxis de una función de Perl, donde el nombre del comando es el nombre de la función, y los argumentos que recibe separados por espacios serán argumentos de dicha función. Veamos un ejemplo:

Si recibe como argumento el siguiente comando del sistema:

```
cp -f webber.pl /home/manolo/
```

tras su ejecución nos devolvería lo siguiente:

```
cp(f","webber.pl","/home/manolo/");
```

Lo primero que hace es dividir la cadena que recibe a su entrada con la función Perl **split** en los trozos separados por espacios en blanco. Cada uno de los elementos obtenidos los sitúa dentro de unas comillas y por último los une siguiendo el formato que arriba se ha comentado: el primero corresponde al nombre de la función y el resto se meten entre los caracteres (y) y separados por comas.

4.4.3 El Gestor de herencia

El gestor de herencia es el encargado de combinar todas las plantillas y clases, convertidos a árboles de formato DOM, heredados por el fichero fuente de entrada, más el fichero de configuración, más el propio fichero para dar como resultado un único árbol que será el utilizado por Webber en su ejecución y

en él leerán y escribirán los valores de las variables los procesadores que el planificador invoque.

La función **GestorHerencia**

Esta la primera función del gestor de herencia que se ejecuta y recibe como entrada la lista de árboles ordenada que hemos comentado anteriormente, la cual va a procesar empezando por el primer elemento -el menos prioritario- hasta llegar al último -el más prioritario-.

Esta función coge los dos primeros árboles que le llegan en la lista, que será los que pase como argumentos a la función **CombinaArboles** para que los procese y devuelva el resultado de combinar esos dos árboles. Este árbol devuelto se vuelve a combinar con el siguiente árbol de la lista y el resultado a su vez con el siguiente árbol. Y así se procede hasta que conseguimos vaciar la lista, de manera que el último árbol devuelto por la función **CombinaArboles** será el árbol global que quede residente en memoria y es el resultado de la ejecución del gestor de herencia.

Después de cada llamada a la función **CombinaArboles**, al árbol que nos devuelve se lo pasamos a la función **EliminaNodos**, cuya única función será eliminar aquellos nodos que poseen el atributo **mode** con el valor **delete**.

La función **CombinaArboles**

Esta función recibe como argumentos dos árboles que debe combinar para devolver uno único como combinación de ambos.

Supongamos que esos dos árboles los denominamos A1 y A2. Cogemos el árbol A2, y vamos recorriendo todos sus nodos hijo y por cada uno de ellos se realiza la siguiente operación: Si ese mismo nodo existe en el otro árbol, A1, se realiza la combinación de ambos nodos utilizando la función **CombinaNodos** y el resultado de la combinación se sustituye por el nodo antiguo en A1. En caso de que el nodo no exista directamente copia el nodo en cuestión en el árbol A1.

Esta función sólo recorre los nodos hijo del primer nivel del árbol, en el caso de que ese nodo exista en los dos árboles ya se encargará la función **CombinaNodos** de profundizar en él.

La función **CombinaNodos**

Esta función recibe como argumento dos nodos y realiza la combinación de ambos la cual devuelve como el nodo resultado. Lo primero que hace es comprobar si los nodos que le llegan son nodos terminales, es decir, si no son nodos elemento que poseen otros nodos dentro, en ese caso se los pasa como argumento a la función **CombinaNodosSegunModo** que realizará la combinación de ambos teniendo en cuenta el valor de sus atributos **mode**.

En el caso de que no sean nodos terminales, por lo que se trata de subárboles de los árboles originales, leemos los elementos hijo de cada uno de los dos nodos y se procede de la misma forma que la función **CombinaArboles** comentada en el apartado anterior.

La función **CombinaNodosSegunModo**

Esta función recibe como argumentos dos nodos terminales de los árboles que está combinando el gestor de herencia, y basándose en los modos que poseen ambos y sabiendo que el segundo nodo tiene preferencia sobre el primero, genera el nodo resultante para el árbol que devuelve como resultado el gestor de herencia. Para ello tiene en cuenta los valores de los atributos **mode** de los dos nodos siguiendo las siguientes reglas:

- Si el primer nodo tiene modo **readonly**, independientemente del modo del segundo nodo se devuelve el primer nodo.
- Si el primer nodo no tiene modo **readonly**:
 - Si el segundo nodo tiene modo **readonly** u **overwrite** el primer nodo será sustituido por éste.
 - Si el segundo nodo tiene modo **append** al primer nodo se le concatenará el contenido íntegro del segundo nodo al final de lo que él contenga.
 - Si el segundo nodo tiene modo **prepend** al primer nodo se le concatenará el contenido íntegro del segundo nodo al principio de lo que él contenga.

La función **EliminaNodos**

Esta función la ejecuta el gestor de herencia después de ejecutar la función **CombinaArboles**, que realiza la combinación de dos árboles, para eliminar aquellos nodos que han permanecido en el árbol y que poseen el modo **delete** que indican que deben ser eliminados del árbol.

Estos nodos pueden permanecer en el árbol porque habíamos comentado que la función `CombinaNodos` si encuentra un nodo en el segundo árbol que no se encuentra en el primero directamente se lo añade a éste último. Si ese nodo añadido que fuera de tipo elemento, es decir, que posea otros nodos hijo, nada impide que alguno de ellos tengan el atributo `mode` con valor `delete`.

Por este motivo se ejecuta esta función que realiza una búsqueda por todos los nodos del árbol resultante de la función `CombinaArboles` para encontrar aquellas con modo `delete` y así eliminarlos del árbol.

Esta función se llama recursivamente para realizar una búsqueda en profundidad en el árbol.

4.4.4 El Planificador

El planificador es el encargado de invocar en el orden adecuado a los procesadores necesarios para llevar a cabo la ejecución de Webber. Estos procesadores se indican en la variable `wbbLib`, la cual puede contener a su vez instrucciones de Perl en nodos denominados `inline` que deberán ser ejecutadas junto con los procesadores y de nuevo siguiendo el orden en el que aparecen.

La función que se encarga de realizar las tareas del planificador es

`LlamaProcesadores`, aunque para su funcionamiento también utilice la función `DOMtoARRAY`.

La función `LlamaProcesadores`

Lo primero que esta función hace es almacenar los directorios contenidos en la variable `wbbProcLib` en la variable de tipo array de Perl `@INC`, que es aquella variable que almacena los directorios en los que el intérprete de Perl busca cada vez que se utiliza un módulo Perl en algún programa ejecutado. Para ello utiliza la función `DOMtoARRAY`, la cual lee un variable DOM que contenga una lista de directorios y devuelve un array donde cada elemento es un directorio que contenía dicha variable.

El segundo paso es leer la variable `wbbLib` que contiene los procesadores a llamar y las instrucciones Perl a ejecutar. Para ello va recorriendo todos los nodos que contiene el nodo que representa a esta variable y nos podemos encontrar dos casos:

- Si el nodo es de texto se trata del nombre del procesador que hay que invocar y el formato que debe seguir es el siguiente: `Módulo::función`. Donde `Módulo` es el nombre del módulo que contiene al procesador a ejecutar y `función` es el nombre de la función contenida en el módulo anterior que queremos ejecutar. El procedimiento a seguir para invocar a un procesador es el siguiente:

```
#con require cargamos el módulo con extensión .pm
require Módulo.pm;
#Invocamos la función del procesador incluida en el módulo.
&función();
```

- Si se trata de un nodo elemento es porque se trata de un nodo `inline` que contiene una lista de instrucciones a ejecutar. En este caso lo que se hace es leer el contenido de texto del nodo `inline` y pasárselo como parámetro a la instrucción `eval` de Perl, que es la encargada de ejecutar una serie de instrucciones de Perl que le llegan en una cadena de texto.

La función `DOMtoARRAY`

Esta función recibe como parámetro el nombre de una variable contenida en el árbol DOM que contiene una lista de nodos elemento donde cada uno de ellos contiene el nombre de un directorio. Veamos un ejemplo de esta variable:

```
<wbbProcLib>
  <dir1>
    /home/manolo/PROYECTO/WEBBER/proc
  </dir1>
  <dir2>
    /home/manolo/PROYECTO/lib
  </dir2>
  ...
  ...
  <dirN>
    /home/procs
  </dirN>
</wbbProcLib>
```

Esta función recorrerá cada uno de los nodos en el ejemplo denominados `dir1,dir2...dirN`, leerá sus contenidos y eliminará los posibles retornos de carro y espacios en blanco que el usuario haya añadido para dar un formato claro al fichero de entrada. Cada uno de los valores leídos los almacena en una posición de una variable de tipo array que será la que devuelva al final de su ejecución.

A partir del ejemplo anterior el resultado que se obtendría sería un array con los siguientes elementos:

```
(/home/manolo/PROYECTO/WEBBER/proc,  
 /home/manolo/PROYECTO/lib,  
 ...,  
 /home/procs/)
```

4.4.5 Funciones que realizan el interface con los árboles DOM

Debido a que la mayoría de las funciones que componen Webber necesitan manipular árboles DOM para trabajar con las variables, ya sea para leer valores de variables o para actualizar valores de variables o incluso crear nodos en el árbol que representen a variables que aún no existían en el árbol, se han creado una serie de funciones que implementan el interface con estos árboles y que serán utilizadas cada vez que queremos acceder a un nodo o variable de dichos árboles.

A continuación se detallan cada una de estas funciones, los parámetros que reciben, la operación que realizan sobre el árbol y el valor que devuelven.

La función `leeTXT`

Esta función se encarga de leer un nodo de un árbol DOM y devuelve su contenido en una cadena de texto.

Recibe como parámetros un árbol y el camino a seguir hasta llegar al nodo que se quiere leer. Este camino es una cadena de texto que sigue el siguiente formato: `nodo1.nodo2.nodo3..nodoN`, donde `nodo1`, `nodo2..nodoN` son cada uno de los nodos por los que se debe pasar, siendo `nodoN` el nodo cuyo contenido se quiere leer (este mismo formato seguirán todos los caminos recibidos como parámetro para el resto de las funciones).

Esta función lo primero que hace es recorrer el árbol siguiendo el camino anteriormente comentado hasta llegar al último nodo, si este nodo es de tipo texto lee su contenido y lo devuelve eliminando antes los posibles espacios en blanco y retornos de carro indeseados que pudiera contener, pero si este nodo es de tipo elemento y contiene a su vez otros nodos hijo, deberá recorrer cada uno de ellos y crear una cadena con la concatenación de las cadenas resultantes de aplicar la función de DOM `toString` (que convierte un nodo de formato DOM a texto) a cada uno de ellos.

En el caso de que no encuentre el nodo indicado devuelve la cadena vacía.

La función `leeDOM`

Esta función realiza la misma tarea que la anterior pero no devuelve el contenido del nodo en formato texto, sino que lo devuelve en formato DOM, es decir tal y como está en el árbol. En el caso de que el nodo no se encuentre en el árbol avisará con un mensaje en pantalla del error y devolverá un valor nulo.

La función `escribeTXT`

Esta función escribe en un nodo del árbol DOM el contenido que reciba en una cadena de texto. Tiene como parámetros el árbol DOM que queremos modificar, el camino que se desea recorrer hasta llegar al nodo a actualizar (puede darse el caso de que el último nodo del camino, el que se corresponde con el nodo a modificar, no exista, entonces quiere decir que se debe crear un nodo nuevo con el nombre de ese último nodo), y la cadena de texto que se desea escribir.

Esta función recorre el árbol siguiendo el camino indicado hasta llegar la nodo buscado:

- Si el nodo existe en el árbol y es un nodo terminal, es decir, no es un nodo que contenga otras variables dentro, su contenido se sustituye por la cadena de texto que recibe como parámetro.
- Si el nodo existe pero no se trata de un nodo terminal, entonces se informa de dicho error y no se modifica el árbol; ya que no podemos modificar un nodo que no se corresponde con una variable.
- Si el nodo no existe en el árbol, entonces creamos un nodo de texto cuyo contenido sea la cadena de texto que recibimos como parámetro

y lo colocamos en el lugar donde se esperaba dicho nodo en el árbol indicado por el camino.

La función `escribeDOM`

Esta función realiza exactamente la misma tarea que la anterior, con la única diferencia que en lugar de recibir una cadena de texto para asignar como valor al nodo, directamente obtenemos el nodo a escribir en formato DOM. Por lo tanto, si el nodo ya existía en el árbol sólo hay que sustituirlo por el que recibimos como parámetro. Y en el caso de que no exista hay que insertarlo en el árbol en el lugar indicado por el camino.

La función `anadeFinalTXT`

Esta función recibe como parámetros el árbol a modificar, el camino al nodo a modificar y una cadena de texto que deberá concatenar al final del contenido de dicho nodo del árbol.

Lo primero que hace es buscar el nodo en el árbol siguiendo el camino que se le pasa como parámetro, en el caso de que el nodo se encuentre y sea un nodo terminal, a su valor se le concatenará al final el valor de la cadena de texto recibida como parámetro. Si el nodo no existe en el árbol se crea en el sitio indicado por el camino y con el contenido de la cadena de texto.

La función `anadeFinalDOM`

Esta función realiza exactamente la misma tarea que la anterior, con la única diferencia que en lugar de recibir una cadena de texto para asignar como valor al nodo, directamente obtenemos el contenido a añadir al nodo en formato DOM. Por lo tanto, si el nodo ya existía en el árbol sólo hay que añadir la estructura DOM recibida al nodo, y si no existía hay que crear un nuevo nodo con el nombre del nodo que estábamos buscando y con el contenido de la estructura DOM recibida, el cual añadimos al árbol en el lugar adecuado.

La función `anadePrincipioTXT`

Esta función recibe como parámetros el árbol a modificar, el camino al nodo a modificar y una cadena de texto que deberá concatenar al final del contenido de dicho nodo del árbol.

Lo primero que hace es buscar el nodo en el árbol siguiendo el camino que se le pasa como parámetro, en el caso de que el nodo se encuentre y sea un nodo terminal, a su valor se le concatenará al principio el valor de la cadena de texto recibida como parámetro. Si el nodo no existe en el árbol se crea en el sitio indicado por el camino y con el contenido de la cadena de texto.

La función `anadePrincipioDOM`

Esta función realiza exactamente la misma tarea que la anterior, con la única diferencia que en lugar de recibir una cadena de texto para asignar como valor al nodo, directamente obtenemos el contenido a añadir al nodo en formato DOM. Por lo tanto, si el nodo ya existía en el árbol sólo hay que añadir la estructura DOM recibida al principio del nodo, y si no existía hay que crear un nuevo nodo con el nombre del nodo que estábamos buscando y con el contenido de la estructura DOM recibida, el cual añadimos al árbol en el lugar adecuado.

La función `NodoTerminal`

Esta función recibe como parámetro un nodo en formato DOM y nos devuelve cierto si es un nodo terminal y falso si no lo es. Un nodo terminal es aquel que se corresponde con una variable.

4.4.6 El módulo `DomWrapper`

La versión anterior de Webber utilizaba un language propio para la descripción de los ficheros fuente y plantilla, que por lo tanto seguían una sintaxis diferente a los actuales. El espacio de tuplas donde se almacenaban las variables estaba implementado como una variable de tipo array asociativo de Perl denominado `var`, donde las claves eran los nombres de las variables y los valores asociados a dichas claves eran los valores que correspondían a esas variables.

Como ya sabemos los procesadores que se ejecutan dentro de Webber acceden a ese espacio que almacena las variables constantemente para poder llevar a cabo su tarea, y poder realizar operaciones de lectura y escritura sobre los valores de las variables. Por lo tanto, en la versión anterior el acceso al espacio de tuplas no se llevaba a cabo mediante las funciones comentadas en el apartado anterior que realizan el interface con el árbol DOM, sino que se realizaba accediendo directamente al array asociativo. Por ejemplo, si

quisiéramos leer el valor de la variable *wbbSourceRegExp* en una variable denominada *\$aux* escribiríamos la siguiente sentencia:

```
$aux = $var{wbbSourceRegExp};
```

Uno de los objetivos buscados era reutilizar los procesadores escritos para la versión anterior de Webber sin modificar su código, y lo que es mejor aún, poder ejecutarlos sin que ellos sepan realmente cual es el contenedor del espacio de tuplas, un array asociativo o un árbol DOM. Para ello se utiliza el módulo **DomWrapper**, el cual transforma los accesos de los procesadores antiguos a un array asociativo de Perl, en accesos a un árbol DOM.

Para ello dentro del código del módulo de Webber se debe introducir la siguiente instrucción con llamada a la función Perl **tie**:

```
tie %var, DomWrapper;
```

La función **tie** asocia el módulo **DomWrapper** a la variable **var** de tipo array asociativo (se aprecia por el carácter **%** que le precede a su nombre), de manera que cuando realizamos una operación sobre dicha variable, automáticamente Perl invoca su función equivalente que está localizada en el módulo asociado, en este caso **DomWrapper**.

En el módulo **DomWrapper** se han implementado sólo tres de todas las posibles funciones asociadas a un array asociativo, y son:

- **STORE:** Esta función se invoca cuando realizamos una escritura sobre el array asociativo. En el módulo **DomWrapper** se convierte en una llamada a la función **escribeTXT** sobre el árbol que representa el espacio de tuplas en Webber y pasándole como parámetro la clave y el valor asociado que queremos que tenga.
- **FETCH:** Esta función se invoca cuando realizamos una lectura sobre el array asociativo. En el módulo **DomWrapper** se convierte en una llamada a la función **leeTXT** sobre el árbol que representa el espacio de tuplas en Webber y pasándole como parámetro la clave cuyo valor asociado queremos leer.
- **EXISTS:** Esta función se invoca cuando realizamos la operación **exists** sobre el array asociativo. Esta operación nos devuelve cierto si existe un valor asociado a la clave que se le pasa como parámetro, y falso si no lo hay. En el módulo **DomWrapper** se convierte en una llamada

a la función `leeTXT` sobre el árbol que representa el espacio de tuplas en Webber y pasándole como parámetro la clave que queremos comprobar si tiene valor asociado. Si la función `leeTXT` nos devuelve un valor distinto de cadena vacía devolvemos cierto porque sabemos que esa clave tiene un valor asociado, en caso contrario devolvemos falso, porque la función `leeTXT` devuelve cadena vacía cuando no encuentra una variable en el árbol.

Capítulo 5

Interface gráfico de Webber. Diseño

5.1 Introducción

La creación de los interfaces de usuario ha sido un área del desarrollo de software que ha evolucionado enormemente a partir de la década de los setenta. El interface de usuario es el vínculo entre el usuario y el programa, y está constituido por un conjunto de comandos o menús a través de los cuales el usuario se comunica con el programa.

La elaboración de un interface de usuario bien diseñado, exige una gran dedicación ya que generalmente los interfaces son grandes, complejos y difíciles de implementar, depurar y modificar. Hoy en día los interfaces de manipulación directa (también llamados interfaces gráficos de usuario, GUI por sus siglas en inglés) son prácticamente universales. Los interfaces que utilizan ventanas, iconos y menús se han convertido en estándar en los materiales computacionales.

El interface representa el punto de encuentro entre el usuario y la computadora. En esta interacción, el usuario juzga la utilidad del interface; el hardware y el software se convierten en simples herramientas sobre las cuales fue construída el interface.

Existen muchos tipos de software para la creación de interfaccs de usuario. El *sistema de ventanas* permite la división de la pantalla en diferentes regiones rectangulares, llamadas *ventanas*. El sistema de ventanas X-Window

para Unix divide la función de la ventana en dos capas: el sistema de ventanas, el cual es el interface funcional, y el administrador de ventanas. El sistema de ventanas provee de procedimientos que permiten a la aplicación dibujar figuras en la pantalla y sirve de medio de entrada de las acciones del usuario. El administrador de ventanas le permite al usuario final mover las ventanas por la pantalla, y es el responsable de desplegar las líneas de título, bordes e iconos alrededor de las ventanas.

La parte central de un sistema de ventanas es el conjunto de herramientas (*toolkit*), el cual contiene los objetos gráficos (*widgets*) más empleados, tales como menús, botones, barras de scroll, y campos para entrada de texto. El *toolkit* generalmente se conecta a los programas de aplicación a través de una serie de procedimientos definidos por el programador. La función de estos procedimientos es el decidir la forma en que se comportarán los objetos gráficos.

5.2 Estructura general

El fichero `Xwebber.pl` se corresponde con el fichero Perl que implementa un interface gráfico para el manejo de Webber. Para ello se utiliza el módulo Perl/Tk como se comentó en el capítulo 2, y tiene la limitación que sólo se puede utilizar para ejecutar Webber sobre un sólo fichero, y no sobre un directorio.

Este interface gráfico está compuesto únicamente por una ventana principal estructurada en seis filas de *widgets*. Los widgets son aquellos objetos que podemos introducir dentro de una ventana, como por ejemplo listas, botones, etiquetas, cuadros de texto, etc...

A continuación podemos ver en la figura un esquema de la ventana principal:

La estructura es la siguiente:

- La primera fila está formada por dos botones de menú, uno para los ficheros y otro para la ayuda.
- La segunda fila está formada por el botón que inicia la ejecución de Webber y por un cuadro de texto que contiene el comando que se va a ejecutar y que sirve de ayuda al usuario.

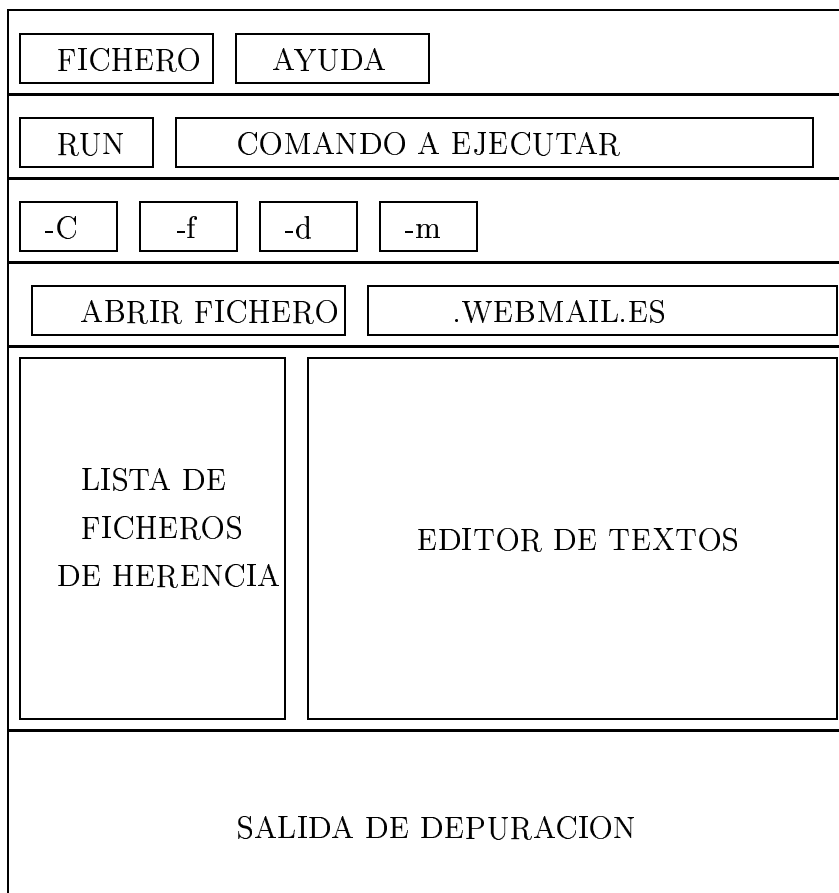


Figura 5.1: Estructura del interfaz gráfico de usuario.

- La tercera fila está compuesta por cuatro *checkboxes*, cada uno de los cuales representa una opción que el usuario tiene disponible. Como se aprecia el número de opciones es menor del que posee Webber, esto es debido a que las relativas al procesamiento de directorios no tendría sentido en esta aplicación que sólo es capaz de procesar un único fichero.
- La cuarta fila está compuesta por una etiqueta que describe al cuadro de texto adjunto y que simboliza el fichero con el que ha sido invocado Xwebber. El nombre de este fichero puede ser modificado en ese cuadro de texto.
- La quinta fila está compuesta por una lista con los ficheros heredados por el fichero de entrada y un editor de textos para poder ver y modificar los ficheros de la lista anterior.
- La sexta y última fila está ocupada por un cuadro de texto que va a capturar la salida de depuración de Webber, para que podamos verla y guardarla en un fichero si queremos.

5.3 Diseño funcional

5.3.1 El cuerpo principal o Main

En el Main del fichero `Xwebber.pl` lo primero que se hace es cargar los módulos que vamos a necesitar. Todos ellos van a coincidir con los que utilizaba el núcleo de Webber, salvo el denominado Tk que es el perteneciente al módulo Perl que antes hemos llamado Perl/Tk y que es el que nos ayuda a crear elementos gráficos para un interface de usuario.

El siguiente paso es leer el nombre del fichero que se ha pasado como argumento (ya que es obligatorio invocar a Xwebber con un fichero de procesamiento). En el caso de que no haya sido facilitado se avisa del error y se para la ejecución.

Una vez leído el fichero, al igual que hacía Webber, el interface gráfica parsea el fichero de configuración para poder empezar la ejecución, para ello utiliza la función `Parser_conf`.

Ahora creamos el objeto ventana que va a representar a la aplicación, cuyo título va a ser `WEBBER v2.0`; y dentro del cual tendremos que añadir todos

los widgets que van a dotar de funcionamiento a nuestra aplicación, y que han sido descritos en el apartado anterior.

Por último invoca a la función `CargarFichero()`.

5.3.2 La función `Parser_conf`

Esta función es idéntica a la que utiliza el núcleo de Webber denominada `Parser`, al igual que el resto de funciones que son invocadas desde ella (`ParserNode`, `CreaInstrucción` y las funciones que hacen de interface con los árboles DOM).

La única diferencia es que no devuelve una lista de árboles en formato DOM de las clases a las que pertenece el fichero, sino que devuelve la lista de los **nombres** de las clases (sus ficheros asociados), que es la que se presenta en la lista de la quinta fila de widgets.

El resto de funciones que son invocadas por ésta son idénticas a las que posee el núcleo de Webber.

5.3.3 La función `CargarFichero`

Esta función se ejecuta la primera vez que invocamos a `Xwebber` y cada vez que modificamos el fichero de entrada en el cuadro de texto correspondiente. Los pasos que realiza esta función son:

- Lo primero que hace es invocar a la función `ProcesaFichero`, que comprueba que el fichero de entrada es válido, mediante la función `LeerTemplates` (idéntica a la del núcleo de Webber), calcula la herencia implícita del fichero de entrada y por último llama a la función `Parser`. Esta función nuevamente es idéntica a la función `Parser` del núcleo de Webber, como ocurría con la función `Parser_conf`, sólo que en vez de devolver la lista de árboles DOM heredados, devuelve la lista de los nombres de las plantillas que es lo que se muestra en la lista de herencia.
- El siguiente paso es actualizar la lista de herencia de ficheros, borrando la posible lista antigua que hubiera de un fichero anterior y añadiendo la obtenida en el paso anterior mediante la función `ProcesaFichero`.
- Actualiza la línea de comando con el nuevo fichero de entrada.

- Si se estaba editando algún fichero lo cierra, y si se había realizado algún cambio sobre él preguntará antes si se quiere salvar.
- Y por último también limpia el contenido del widget que presenta la información de depuración de la ejecución de Webber.

5.3.4 La función InsertaComando

Esta función se ejecuta cuando activamos alguno de los checkboxes asociados con los argumentos de Webber, ya que en el cuadro de texto que representa al comando se añadirá el argumento seleccionado en su lugar adecuado.

Captura el contenido del cuadro de texto mencionado, divide la cadena que contiene en elementos que almacena en una array -ayudándose de la función **split**-, y después vuelve a concatenar cada uno de los elementos introduciendo entre el último elemento y el penúltimo el argumento seleccionado. Veamos un ejemplo:

Si el comando que se iba a ejecutar era `perl webber.pl .webmail.es` y activamos el argumento `-r`, el nuevo comando construido por esta función sería: `perl webber.pl -r .webmail.es`

5.3.5 La función EliminaComando

Esta función se ejecuta cuando desactivamos alguno de los checkboxes asociados con los argumentos de Webber, ya que en el cuadro de texto que representa al comando se eliminará el argumento seleccionado.

Captura el contenido del cuadro de texto mencionado, divide la cadena que contiene en elementos que almacena en una array (ayudándose de la función **split**).

Busca en el array el argumento que ha sido seleccionado para ser eliminado, cuando lo encuentra lo elimina del array y construye una cadena concatenando el resto de elementos del array en el orden en el que están. En el caso de que el argumento seleccionado hubiese sido `-C`, también tendríamos que eliminar del array el posible nombre del fichero de configuración seleccionado.

El último paso es actualizar el widget que visualiza el comando que se va a ejecutar con la cadena de texto obtenida en el paso anterior.

Si el comando que se iba a ejecutar era `perl webber.pl -C webber2.conf .webmail.es` y desactivamos el argumento `-C`, el nuevo comando construido por esta función sería: `perl webber.pl .webmail.es`.

5.3.6 La función EditarFichero

Esta función se invoca cada vez que seleccionamos un fichero de la lista de los ficheros heredados por el fichero de entrada. Con esto conseguimos que su contenido se muestre en el editor para que podamos observarlo y realizar modificaciones.

Lo primero que se hace es comprobar si había algún fichero actualmente en edición, en caso de que si lo hubiese, antes de cerrarlo comprueba si sufrió modificaciones para preguntarnos, mediante una ventana emergente, si queremos salvarlo. Una vez que se ha tratado el fichero que estaba en edición se carga en el editor el nuevo fichero seleccionado dispuesto para ser editado.

5.3.7 La función CerrarSalvar

Esta función es llamada cada vez que tenemos una ventana emergente que nos pregunta si queremos salvar el fichero que está actualmente en edición y el usuario elige la opción **Salvar**.

Cierra la ventana emergente e invocar a la función **SalvarFichero** para que el contenido del editor de textos lo almacene en el fichero que se estaba editando.

5.3.8 La función Cerrar

Esta función es llamada cada vez que tenemos una ventana emergente que nos pregunta si queremos salvar el fichero que está actualmente en edición y el usuario elige la opción **Cancelar**. Cierra la ventana emergente sin salvar el fichero que se estaba editando.

5.3.9 La función SalvarFichero

Esta es la función encargada de salvar el contenido del editor de texto (que recibe como parámetro) en el fichero de texto correspondiente (que también recibe como parámetro).

Lo primero que hace es abrir el fichero en modo escritura, ya que se le va a sobrescribir todo su contenido, y después vuelve el contenido que recibe como parámetro o si es cadena vacía, el contenido del editor de texto en dicho fichero. Por último, cierra el fichero.

5.3.10 La función **MostrarAyuda**

Esta función se ejecuta cuando seleccionamos la opción **Ayuda** del menú superior de la ventana. Muestra una ventana emergente con un breve manual de usuario sobre el uso de la herramienta Xwebber.

5.3.11 La función **About**

Esta función muestra una pequeña ventana con información sobre el programa Xwebber, concretamente indica la versión que se está ejecutando del programa, el nombre del autor y agradecimientos a las personas que han colaborado con él.

5.3.12 La función **EjecutaWebber**

Esta función es invocada cada vez que el usuario pulsa sobre el botón **RUN** que lanza la ejecución de Webber.

Lo primero que hace es leer el contenido del cuadro de texto que representa el comando que se va a ejecutar y en el cual el usuario introdujo los argumentos con los que quería que Webber se ejecutase.

Ese contenido lo pasa como parámetro a la función **CreaInstruccion**, para que ésta le convierta dicho comando del sistema en la cadena de texto que le debe pasar a la función **eval** de Perl para que se ejecute.

A continuación realiza la ejecución de Webber, obteniendo en una variable el posible contenido de depuración que éste nos pueda devolver. El contenido recogido será situado en el cuadro de texto que recoge la salida de depuración de Webber y que está en la parte inferior de la ventana principal.

Por último, para avisar de la finalización de la ejecución de Webber produce el sonido de una campana.

Capítulo 6

Código fuente

6.1 Código fuente del núcleo de Webber

6.1.1 Módulo Webber

```
#!/usr/bin/perl
package Webber;
use Shell;
use POSIX;
use XML::DOM;
use Cwd;
use Getopt::Std;
use DomWrapper;
tie %var, DomWrapper; #PARA QUE FUNCIONEN LOS ANTIGUOS PROCESADORES
#####
#CUERPO PRINCIPAL
#####
$opt_d=0;
$opt_f=0;
$opt_h=0;
$opt_i=0;
$opt_C="webber.conf";
$opt_r=0;
$opt_m=0;
$opt_H=0;
$opt_I=0;
getopts("C:dfhirmHI");

$arbol_final; #ARBOL RESULTANTE DEL GESTOR DE HERENCIA
```

```

if ($opt_h){showHelp();}#NOS PIDEN AYUDA

#AQUI ESTAN LOS ARBOLES QUE EL GESTOR DE HERENCIA
#TIENE QUE UNIR EN UNO SOLO
my @array_arboles;
#LEEMOS EL FICHERO DE CONFIGURACION
print STDERR "Debug: Usando el fichero $opt_C para configuracion.\n"
            if $opt_d;

my @conf_aux;
Parser($opt_C,\@conf_aux);
my $file_conf=$conf_aux[0];
if ($opt_H) {Procesadores("help");}
if ($opt_I) {Procesadores("info");}
if ($opt_i) {ProcesaSTDIN();}

while ($f_d=shift(@ARGV))#PROCESAMOS TODOS LOS FILES O DIR DE ENTRADA
{

    print STDERR "Debug: Procesando el fichero o directorio $f_d.\n"
                if $opt_d;
    if (-T $f_d)  #SI ES UN FICHERO DE TEXTO
    {
        ProcesaFichero($f_d);
    }
    elsif (-d $f_d) #SE TRATA DE UN DIRECTORIO
    {
        ProcesaDirectorio($f_d);
    }
    else
    {print "Directorio o fichero: $f_d no existe.\n";}
}

#####
#CADA UNA DE LAS FUNCIONES QUE EL WEBBER NECESITA
#####

#####
# La función Procesadores
#####

```

```

sub Procesadores
{
  my $output="=>";
  my $comm=$_[0];
  #LEEMOS LOS DIRECTORIOS DONDE PUEDEN ESTAR LOS PROCESADORES
  my $webbProcLib=leeDOM($file_conf,"webber.webbProcLib");
  my $dirs=$webbProcLib->getElementsByTagName("*",0);
  my $long=$dirs->getLength;
  my @directorios;
  for (my $i=0;$i<$long;$i++)
  {
    my $actual=$dirs->item($i);
    if ($actual->getFirstChild->getNodeTypeName eq "TEXT_NODE" &&
        $actual->getChildNodes->getLength==1)
    {
      my $dir=$actual->getFirstChild->getData;
      $dir=~/[\\n\\s]*([\\s].*[\\s])\\s*/;
      $dir=$1;
      push(@directorios,$dir);
    }
  }
  #EN @directorios ESTAN LOS DIRECTORIOS DONDE PUEDEN
  #ESTAR LOS PROCESADORES
  foreach $i (@directorios) {
    opendir DIR, $i || die "Can't open directory $i!!\n";
    @temp=readdir (DIR);
    closedir DIR;
    foreach $x (@temp ) {
      next if ($x eq ".");
      next if ($x eq "..");
      @xx= split /\./, $x;
      pop @xx;
      $y = join "." , @xx;
      require $i. "/"$x";
      $com= $y . ":" . $comm;
      if (defined &$com) {
        print $x, $output;
        $val = &$com;
      }
      else { print $x, $output, "No hay informacion disponible para este
procesador.\n"; }
    }
  }
}

```

```

    }
    exit;

}

#####
#La función showHelp
#####

sub showHelp
{
    printf "webber [-C <configFile>] [-d] [-f] [-h] [-i] [-I] [-m] [-r]
    <tplFichero>] (dir|fichero) [(dir|fichero)... ]\n\n";
    printf "-C establece un fichero diferente de configuracion\n";
    printf "-d establece el modo de depuracion\n";
    printf "-f fuerza a actualizar los ficheros de salida\n";
    printf "-h muestra este mensaje\n";
    printf "-H muestra la ayuda los procesadores instalados\n";
    printf "-i usa la i/o estandar, y no un fichero\n";
    printf "-I lista los procesadores instalados\n";
    printf "-m crea los directorios destino si no existen\n";
    printf "-r procesa recursivamente directorios\n";
    printf "<dir>   Procesa el contenido de un directorio\n";
    printf "<file>  Procesa un fichero fuente individual\n";
    printf "\nVer además $opt_C\n";
    exit;
}

#####
#La función STDIN
#####

sub ProcesaSTDIN
{
    my $entrada="";
    open INFILE, "-";
    while ($line=<INFILE>) {$entrada=$entrada.$line;}
    my @array_arboles;

```

```

Parser($entrada,\@array_arboles,"STDIN");
unshift(@array_arboles,$file_conf);
$arbol_final=GestorHerencia(\@array_arboles);
escribeTXT($arbol_final,"webber.wbbDate",
            POSIX::strftime ("%Y %m %d", localtime));
escribeTXT($arbol_final,"webber.wbbOut","");
LlamaProcesadores();
}

#####
#La función ProcesaFichero
#####

sub ProcesaFichero
{
    my $f_d=$_[0];
    #LEAMOS LAS VARIABLES QUE NECESITAMOS
    my $wbbSourceRegExp=leeTXT($file_conf,"webber.wbbSourceRegExp");
    my $wbbTemplateName=leeTXT($file_conf,"webber.wbbTemplateName");
    my $wbbTargetRoot=leeTXT($file_conf,"webber.wbbTargetRoot");
    my $wbbSourceRoot=leeTXT($file_conf,"webber.wbbSourceRoot");
    my $wbbExtension=leeTXT($file_conf,"webber.wbbExtension");
    my $wbbFileNameRegExp=leeTXT($file_conf,"webber.wbbFileNameRegExp");
    my $wbbFileLangRegExp=leeTXT($file_conf,"webber.wbbFileLangRegExp");
    my $wbbExtParser=leeTXT($file_conf,"webber.wbbExtParser");

    my @aux = split /\// , $f_d;
    my $fichero=$aux[$#aux];
    #CONTIENE EL NOMBRE DEL FICHERO FUENTE SIN LA RUTA

    #VEAMOS DE QUE IDIOMA SE TRATA
    if ($fichero =~ /$wbbFileLangRegExp/)
    {
        $lang = $1;
    }
    else { $lang = ""; }
    if (DefinidoIdioma($lang))
    {
        escribeTXT($file_conf,"webber.wbbLang",$lang);
    }
}

```

```

else {escribeTXT($file_conf,"webber.wbbLang","none");}
#FIN DE TRATAMIENTO DEL IDIOMA

$fichero =~ /$wbbFileNameRegExp/;
my $nombre_destino = $1;
pop @aux;
my $dir_source=join("/",@aux);
if ($fichero=~/$wbbSourceRegExp/
    && $fichero ne $wbbTemplateName)
{
    #ME VOY AL DIRECTORIO DONDE ESTA EL FICHERO A LEER LOS TEMPLATES
    my @array_arboles=LeerTemplates($f_d,\@array_arboles);
    #HAGO EL PARSER DEL FICHERO QUE ME PASAN COMO PARAMETRO
    #SIEMPRE QUE NO ESTE ACTIVA LA VARIABLE wbbExtParser
    if ($wbbExtParser eq "" || $wbbExtParser eq "0")
    {
        Parser($f_d,\@array_arboles);
    }
    #EL FICHERO DE CONFIGURACION ES EL ULTIMO EN SER PROCESADO POR EL
    #GESTOR DE HERENCIA, POR ESO LO PONGO AL PRINCIPIO
    unshift(@array_arboles,$file_conf);
    $arbol_final=GestorHerencia(\@array_arboles);
    my $target;
    my @stats_fuente=stat($f_d);#LEO SU FECHA DE CREACION
    escribeTXT($arbol_final,"webber.wbbDate",
        POSIX::strftime ("%Y %m %d", localtime ($stats_fuente[9])));
    #CALCULAMOS EL NOMBRE DEL FICHERO DE DESTINO
    my $wbbTargetName = leeTXT($file_conf,"webber.wbbTargetName");
    if ($wbbTargetName ne "")
    {
        if ($wbbTargetName !~ /\^\/)
        {
            $target=getcwd();
            $target =~ s/$wbbSourceRoot/$wbbTargetRoot/;
            $target.="/".$wbbTargetName;
        }
        else {$target=$wbbTargetName;}
    }
    else
    {
        my $wbbTarget = leeTXT($file_conf,"webber.wbbTarget");

```

```

    if ($wbbTarget ne "")
    {
        if ($wbbTargetName !~ /\^\//)
        {
            $target=getcwd();
            $target =~ s/$wbbSourceRoot/$wbbTargetRoot/;
            $target.="/".$wbbTarget.$wbbExtension;
        }
        else {$target=$wbbTarget.$wbbExtension;}
    }
else
{
    if ($f_d =~ /\^\//)
    {
        $dir_source =~ s/$wbbSourceRoot/$wbbTargetRoot/;
        mkdir $dir_source,0755 if ($opt_m and (not -d $dir_source));
        $target=$dir_source."/$nombre_destino".$wbbExtension;
    }
    else
    {
        $target=getcwd();
        if ($dir_source ne ""){ $target.="/$dir_source/";}
        $target =~ s/$wbbSourceRoot/$wbbTargetRoot/;
        mkdir $target,0755 if ($opt_m and (not -d $target));
        $target .="/".$nombre_destino.$wbbExtension;
    }
}
}

escribeTXT($arbol_final,"webber.wbbTarget",$target);
escribeTXT($arbol_final,"webber.wbbSource",$fichero);
escribeTXT($arbol_final,"webber.wbbOut","");
#COMPROBAMOS LA FECHA DEL FICHERO DE DESTINO
my @stats_destino=stat($target);
if ($stats_destino[9] <= $stats_fuente[9] || $opt_f)
{
    open PAGE,">".$target;
select PAGE;
    #EL DESTINO ES + ANTIGUO QUE EL FICHERO FUENTE
    #LLAMAMOS A LOS PROCESADORES QUE INDICA $wbbLib
    LlamaProcesadores();
}

```

```

        print leeTXT($arbol_final,"webber.wbbOut");
        select STDOUT;
        close PAGE;
        if ($wbbTargetFileMode ne "")
        {chmod oct $wbbTargetFileMode,$target";}
        else{ chmod 0444, "$target";}
    }
    else {print "El fichero destino es mas nuevo que el
              fuente o no se puede crear.\n";}
    }
    else
    {
        print "Fichero llamado $f_d no valido.No cumple
              expresion regular o es template.\n";
    }
}

#####
#La función LlamaProcesadores
#####

sub LlamaProcesadores
{
    #CARGAMOS LOS DIRECTORIOS DONDE PUEDEN
    #ESTAR LOCALIZADOS LOS PROCESADORES
    my @directorios=DOMtoARRAY("wbbProcLib");
    unshift @INC,@directorios;
    #Y AHORA LLAMAMOS A LOS PROCESADORES EN ORDEN,
    #ESTA EN LA VBLE wbbLIB
    my $wbbLib=leeDOM($arbol_final,"webber.wbbLib");
    my $procs=$wbbLib->getChildNodes;
    my $long=$procs->getLength;
    for (my $i=0;$i<$long;$i++)
    {
        my $proc=$procs->item($i);
        if ($proc->getNodeTypeName eq "TEXT_NODE")#ES UN PROCESADOR
        {
            my $texto=$proc->getData;
            my @files= split /\s/, $texto;
            for (my $j=0;$j<=$#files;$j++)

```

```

{
    #SI CONTIENE ALGUNA LETRA NOS INTERESA
    if ($files[$j] =~ /\w/)
    {
        #ELIMINAMOS LOS ESPACIOS EN BLANCO QUE CONTENGA
        $files[$j] =~ /\s*([^\s].*[^\s])\s*/;
        my $procesador=$1;
        if ($procesador ne "")
        {
            print STDERR "Debug: Llamamos al procesador:$procesador\n"
                if $opt_d;
            ($paquete,$nombre)=split /::/, $procesador;
            require $paquete.".pm";
            &$procesador();
        }
    }
}
}
else #ES UN <INLINE>
{
    my $texto=$proc->getFirstChild->getData;
    $texto =~ /\n\s*([^\s].*[^\s])\s*\n*/;
    $texto=$1;
    if ($texto ne "")#EJECUTAMOS LA SENTENCIA PERL
    {
        eval("$texto");
    }
}
}
}
}

```

```

#####
#La función DOMtoARRAY
#####

```

```

sub DOMtoARRAY
{
    my $var = $_[0];
    $var = "webber".".".$var;
    my $arbol=leeDOM($file_conf,$var);
    my $dirs=$arbol->getElementsByTagName("*",0);

```

```

my $long=$dirs->getLength;
my @directorios;
for (my $i=0;$i<$long;$i++)
{
    my $actual=$dirs->item($i);
    if ($actual->getFirstChild->getNodeTypeName eq "TEXT_NODE" &&
        $actual->getChildNodes->getLength==1)
    {
        my $dir=$actual->getFirstChild->getData;
        $dir=~/[\\n\\s]*([\\s].*[\\s])\\s*/;
        $dir=$1;
        push(@directorios,$dir);
    }
}
}#EN @directorios ESTAN LOS DIRECTORIOS
#DONDE PUEDEN ESTAR LOS PROCESADORES
return @directorios;
}

```

```

#####
#La función DefinidoIdioma
#####

```

```

sub DefinidoIdioma
{
    my $idioma = $_[0];
    my $wbbLangs=leeDOM($file_conf,"webber.wbbLangs");
    if (defined $wbbLangs)
    {
        my $lang=$wbbLangs->getElementsByTagName("lang",0);
        my $long=$lang->getLength;
        for (my $i=0;$i<$long;$i++)
        {
            my $aux=$lang->item($i)->getAttributeNode("id")->getValue;
            if ($aux eq $idioma) {return 1;}
        }
        return 0;
    }
    else {return 0;}
}

```

```

#####

```

```

#La función ProcesaDirectorio
#####

sub ProcesaDirectorio
{
    my $directorio=$_[0];
    my $dir_actual=getcwd();
    my $aux;
    my $aux2;
    #LEAMOS LAS VARIABLES QUE NECESITAMOS
    my $wbbSourceRegExp=leeTXT($file_conf,"webber.wbbSourceRegExp");
    my $wbbTemplateName=leeTXT($file_conf,"webber.wbbTemplateName");
    my $wbbTargetRoot=leeTXT($file_conf,"webber.wbbTargetRoot");
    my $wbbSourceRoot=leeTXT($file_conf,"webber.wbbSourceRoot");
    my $wbbExtension=leeTXT($file_conf,"webber.wbbExtension");
    my $wbbFileNameRegExp=leeTXT($file_conf,"webber.wbbFileNameRegExp");

    #DEJO UN ESPACIO EN BLANCO PARA APROVECHAR LA FUNCION LEERTEMPLATES
    #DISEÑADA PARA CUANDO LA ENTRADA ES UN FICHERO
    if ($directorio !~ /\$/ )
        {$aux=$directorio."/ ";}
    else {$aux=$directorio." ";}

    #SI EL DIRECTORIO NO ES ABSOLUTO CALCULAMOS SU RUTA COMPLETA
    if ($directorio !~ /\^\/) {$directorio=$dir_actual."/".$directorio;}
    #SI EL DIRECTORIO NO TRAIA / AL FINAL SE LA AÑADIMOS
    if ($directorio !~ /\$/ ) {$directorio=$directorio."/";}
    my @dirs=();
    opendir DIR,$directorio;
    while ($fic_dir=readdir(DIR))
    {
        next if ($fic_dir =~ /\^\.$/ || $fic_dir =~ /\^\.\.$/);
        $aux2=$fic_dir;
        $aux2 =~ /\$wbbFileNameRegExp/;
        my $nombre_destino=$1;
        #CALCULAMOS SU DIRECCION COMPLETA DESDE DONDE SE EJECUTA EL WEBBER
        $fic_dir=$directorio.$fic_dir;
        next if (-l $fic_dir || -p $fic_dir || -S $fic_dir || -b $fic_dir
            || -c $fic_dir || -t $fic_dir);
    }
}

```

```

if (-d $fic_dir && $fic_dir !~ /\.$/ && $fic_dir !~ /\.\.$/)
{
    push @dirs,$fic_dir;
}
elsif (-f $fic_dir && $aux2 =~ /$wbbSourceRegExp/ &&
        $aux2 ne $wbbTemplateName)
#SE TRATA DE UN FICHERO VALIDO
{
    #CREAMOS DE NUEVO EL FICHERO DE CONFIGURACION PORQUE CADA VEZ QUE
    #PROCESAMOS UN FICHERO ES MODIFICADO
    my @conf;
    Parser($opt_C,\@conf);
    $file_conf=$conf[0];
    print STDERR "Debug: Procesando el fichero o directorio
                  $fic_dir.\n" if $opt_d;
    ProcesaFichero($fic_dir);
}
}
#SI HABIA SUBDIRECTORIOS LOS PROCESAMOS SI ACTIVA OPCION -R
if ($opt_r)
{
    foreach $dir (@dirs)
    {ProcesaDirectorio($dir);}
}
}

#####
#La función LeerTemplates
#####

sub LeerTemplates
{
    my $lista_arboles=$_[1];
    my $f_d=$_[0];
    my $dir_actual=getcwd();
    my $wbbTemplateName=leeTXT($file_conf,"webber.wbbTemplateName");
    my $wbbSourceRoot=leeTXT($file_conf,"webber.wbbSourceRoot");
    my @aux = split /\// , $f_d;
    my $dir=join "/",@aux[0..$#aux-1];
    die "No puedo acceder al directorio $dir" if "$dir" ne "" and !(chdir
        ($dir));
}

```

```

chdir $dir_actual;#Me vuelvo a mi sitio por si la inst de antes tiene
                    #efecto
#SI ME DAN LA RUTA RELATIVA DONDE ESTA EL FICHERO DEBO CALCULAR LA
#ABSOLUTA PARA LEER RECURSIVAMENTE LOS TEMPLATES.
if ($dir !~ /\^\/)
{
    if ($dir ne "")
    {$dir=$dir_actual."/".$dir;}
    else {$dir=$dir_actual;}
}
my $template;
print STDERR "Debug: Procesando templates del
                directorio $dir.\n" if $opt_d;
if ($dir !~ /\$/){$template=$dir."/".$wbbTemplateName;}
else {$template=$dir.$wbbTemplateName;}
if (-T $template) {
    print STDERR "Debug: Procesando el template $template.\n" if $opt_d;
    Parser($template,\@$lista_arboles);}
while ($dir =~ /\^(.*)\/ && $dir ne $wbbSourceRoot && $dir ne "/")
{
    $dir=$1;
    if ($dir ne "")
    {
        $template=$dir."/".$wbbTemplateName;
        my @aux2;
        if (-T $template)
        {
            print STDERR "Debug: Procesando el
                        template $template.\n" if $opt_d;
            Parser($template,\@aux2);
            my $e;
            while ($e=pop(@aux2)) {unshift(@$lista_arboles,$e);}
        }
        if ($dir eq $wbbSourceRoot || $dir eq "/"){last;}
    }
    else{last;}
}
return @$lista_arboles;
}

```

```
#####
```

```

#La función Parser
#####

sub Parser
{

    # ESTA FUNCION HACE EL PARSER DEL FICHERO,
    #TRATA LA POSIBLE HERENCIA EN EL
    # ATRIBUTO INHERIT Y EJECUTA LAS INSTRUCCIONES
    #DE PROCESAMIENTO <?...?>
    # SE LE PASA COMO PARAMETRO EL NOMBRE DEL FICHERO
    my $arboles=$_[1];
    my $doc=$_[0];
    my $stdin=$_[2];
    my $file;
    my $parser = new XML::DOM::Parser(ProtocolEncoding=>'ISO-8859-1');
    if ($stdin ne "STDIN")
    {$file = $parser->parsefile($doc);}
    else {$file = $parser->parse($doc);}
    my $aux = $file->getElementsByTagName ("webber",0);
    my $length=$aux->getLength;
    my $padre;
    my $nodes;
    my $n;
    if ($length==0)
    {
        print "ERROR...NO HAY NINGUN <webber> DE ENTRADA.\n";
        $n=0;
    }
    else
    {
        $padre=$aux->item(0);
        $nodes = $aux->item(0)->getChildNodes;
        $n = $nodes->getLength;
    }

    my $inherit=$aux->item(0)->getAttributeNode("inherit");
    if (defined $inherit)
    {$inherit=$aux->item(0)->getAttributeNode("inherit")->getValue;}
    else {$inherit="";} #EN $inherit TENEMOS EL

```

#POSIBLE VALOR DEL FILE DE HERENCIA

```

for (my $i = 0; $i < $n; $i++)
{
    #HAY QUE CALCULAR LA LONGITUD DE LA LISTA A CADA ITERACION PORQUE
    #PUEDEN DESAPARECER HIJOS DE LA LISTA Y LOS INDICES VARIAN.DE LA
    #LISTA SE PIERDEN AQUELLAS P.I MAL CONSTRUIDAS
    my $n_aux=$nodes->getLength;
    if ($n_aux != $n) {$n=$n_aux;$i=$i-1;}
    my $node = $nodes->item($i);
    my $type=$node->getNodeTypes;
    ParserNode($node,$padre,$file);
}
#SI HAY HERENCIA,BUSCAMOS EL FICHERO EN LOS DIRECTORIOS DEFINIDOS EN
# WBBCLASSLIB Y METEMOS EN LA LISTA DICHO ARBOL
if ($inherit ne "")
{
    if ($inherit !~ /\// && !(-T $inherit))
    #TENEMOS QUE BUSCAR EL FICHERO
    {
        my $wbbClassLib=leeDOM($file_conf,"webber.wbbClassLib");
        #LEEMOS TODOS LOS DIRECTORIOS DE CADA UNO DE SUS HIJOS
        my $dirs=$wbbClassLib->getElementsByTagName("*",0);
        my $long=$dirs->getLength;
        my @dir_herencia;
        for (my $i=0;$i<$long;$i++)
        {
            my $actual=$dirs->item($i);
            if ($actual->getFirstChild->getNodeTypeName eq "TEXT_NODE" &&
                $actual->getChildNodes->getLength==1)
            {
                my $dir=$actual->getFirstChild->getData;
                $dir=~/[\\n\\s]*([\\s].*[\\s])\\s*/;
                $dir=$1;
                push(@dir_herencia,$dir);
            }
        }
        for (my $i=0;$i<=$#dir_herencia;$i++){
            $dir_herencia[$i]=~/^\\s*(.*)\\s*$/;
            my $dir=$1;
            if ($dir=~/. / && !(-T $inherit))

```

```

        #MIRAMOS SI EL FICHERO ESTA EN LOS DIRECTORIOS
        {
            my $actual=getcwd();
            chdir($dir);
            if (-T $inherit)
            {
                $inherit=$dir."/".$inherit;
                chdir ($actual);
                last;
            }
            chdir($actual);
        }
    }
}
Parser($inherit,\@$arboles);
}
#POR ULTIMO INCLUIMOS NUESTRO ARBOL

push(@$arboles,$file);

}

#####
#La función ParserNode
#####

sub ParserNode
{
    # RECIBE EL NODO A EXAMINAR Y EL PADRE POR SI SE
    # ENCUENTRA UNA P.I SUSTITUYA
    # Y EL DOCUMENTO RAIZ
    # SU VALOR SOLO TENDREMOS EN CUENTA LOS NODOS
    # DE TIPO 1(ELEMENTOS) Y 7(INST PROC).

    my $nodo=$_[0];
    my $padre=$_[1];
    my $doc=$_[2];
    my $type=$nodo->getNodeTypes;
    if ($type == 7)#SE TRATA DE UNA INSTRUCCION DE PROCESAMIENTO
    {
        my $target=$nodo->getTarget;
    }
}

```

```

my $data=$nodo->getData;
if ($target eq "eval")#es un eval de perl
{
    my $aux=eval("$data");
    my $new_nodo=$doc->createTextNode("$aux");
    $padre->replaceChild($new_nodo,$nodo);

}

elsif ($target eq "system")#SE TRATA DE UNA LLAMADA AL SISTEMA
{
    my $aux;
    my $instruccion=CreaInstruccion($data);
    $aux=eval($instruccion);
    my $new_nodo=$doc->createTextNode("$aux");
    $padre->replaceChild($new_nodo,$nodo);

}
elsif ($target eq "perl")#SE TRATA DE UNA FUNCIÓN PERL
{
    my $aux=eval("&" . $data . ";");
    my $new_nodo=$doc->createTextNode("$aux");
    $padre->replaceChild($new_nodo,$nodo);
}
else
{
    print STDERR "Debug: Instruccion de procesamiento
                  no valida:" if $opt_d;
    print STDERR $nodo->toString if $opt_d;
    $padre->removeChild($nodo);
}
}

if ($type==1) #Es un nodo normal
{
    my $hijos = $nodo->getChildNodes;
    my $num_hijos=$hijos->getLength;
    for (my $k = 0; $k < $num_hijos ; $k++)
    {
        my $hijo = $hijos->item ($k);
        ParserNode($hijo,$nodo,$doc);
    }
}

```

```

    }
}

}

#####
#La función CreaInstruccion
#####

sub CreaInstruccion
{
# A PARTIR DEL CAMPO DATA DE UNA INSTRUCCION DE PROCESAMIENTO CREA LA
# INSTRUCCION A EJECUTAR EN EL SISTEMA.
# DE "cp file1 file2" PASA A "cp("file1","file2");"
my $lista=$_[0];
my @arg=split /\s/ , $lista;
my $command=shift(@arg);
for (my $i=0;$i<=$#arg;$i++){
    $arg[$i]="\" . $arg[$i] . "\"";
}
my $aux=join(", ",@arg);
my $inst=$command."("$aux.")";
return $inst;
}

#####
#La función GestorHerencia
#####

sub GestorHerencia
{
my $arboles=$_[0];
#SI SOLO HAY UN ARBOL EN LA LISTA LO DEVOLVEMOS
if ($#$arboles==0)
    {return $$arboles[0];}
else #HAY MAS ARBOLES:AQUI SI PARTICIPA EL GESTOR DE HERENCIA
{
    #EXTRAEMOS LOS DOS PRIMEROS ARBOLES
    my $a1=shift(@$arboles);
    my $a2=shift(@$arboles);
    my $aux=CombinaArboles($a1,$a2);

```

```

    EliminaNodos($aux);
    if ($#$arboles<0)#HEMOS ACABADO SI NO HAY MAS ARBOLES
    {return $aux;}
    else #AUN QUEDAN MAS ARBOLES
    {
        unshift(@$arboles,$aux);
        GestorHerencia(\@$arboles);
    }

}

}

#####
#La función CombinaArboles
#####

sub CombinaArboles
{
    my $a1=$_[0];
    my $a1_aux=$a1->getElementsByTagName("webber",0)->item(0);
    my $a2=$_[1];
    my $a2_aux=$a2->getElementsByTagName("webber",0)->item(0);
    my $hijos2=$a2_aux->getElementsByTagName("*",0);
    my $n=$hijos2->getLength;
    for (my $i=0;$i<$n;$i++)
    {
        my $h2=$hijos2->item($i);
        my $name=$h2->getNodeName;
        #BUSCAMOS ESTE NODO EN EL PRIMER ARBOL
        my $h1=$a1_aux->getElementsByTagName($name,0)->item(0);
        if (defined $h1)
        {CombinaNodos($h1,$h2);}
        else {escribeDOM($a1_aux,$name,$h2);}
    }
    return $a1;
}

#####
#La función CombinaNodos

```

```
#####
```

```
sub CombinaNodos
{
    my $nodo1=$_[0];
    my $nodo2=$_[1];
    my $hijos2=$nodo2->getChildNodes;#AQUI COGEMOS TODOS LOS HIJOS
    # SUPONGO QUE NO HAY ELEMENTOS VACIOS.
    # LOS DOS NODOS CONTIENEN TEXTO.SI EL PRIMERO CONTIENE
    # TEXTO SUPONEMOS QUE EL SEGUNDO TAMBIEN.
    if (NodoTerminal($nodo2))
    {
        CombinaNodosSegunModo($nodo1,$nodo2);
    }
    else #EL SEGUNDO NODO TIENE ELEMENTOS HIJOS
    {
        $hijos2=$nodo2->getElementsByTagName("*",0);
        #SOLO HIJOS-ELEMENTOS
        my $n=$hijos2->getLength;
        for (my $i=0;$i<$n;$i++)
        {
            my $h2=$hijos2->item($i);
            #LO PROCESAMOS SI ES UN NODO ELEMENTO
            if($h2->getNodeTypeName eq "ELEMENT_NODE")
            {
                my $name=$h2->getNodeName;
                #BUSCAMOS ESTE NODO EN EL PRIMER ARBOL
                my $h1=$nodo1->getElementsByTagName($name,0)->item(0);
                if (defined $h1)
                { CombinaNodos($h1,$h2);}
                else {escribeDOM($nodo1,$name,$h2);}
            }
        }
    }
}
```

```
#####
#La función CombinaNodosSegunModo
#####
```

```

sub CombinaNodosSegunModo
{
  my $node1=$_[0];
  my $node2=$_[1];
  my $mode1=$node1->getAttribute("mode");
  my $mode2=$node2->getAttribute("mode");
  if ($mode1 ne "readonly")
  {
    if ($mode2 eq "delete")
    {
      my $padre1=$node1->getParentNode;
      $padre1->removeChild($node1);
    }
    if ($mode2 eq "" || $mode2 eq "overwrite"
        || $mode2 eq "readonly")
    {
      my $padre1=$node1->getParentNode;
      #HAY QUE CAMBIAR EL PROPIETARIO DEL NODO2
      my $owner=$node1->getOwnerDocument;
      $node2->setOwnerDocument($owner);
      $padre1->replaceChild($node2,$node1);
    }
    if ($mode2 eq "append")
    {
      if ($node1->getNodeName ne "wbbLib")
      {
        my $text1;
        my $text2;
        if ($node1->hasChildNodes)
        {$text1=$node1->getFirstChild->getData;}
        else {$text1="";}
        if ($node2->hasChildNodes)
        {$text2=$node2->getFirstChild->getData;}
        else {$text2="";}
        $text1=$text1. " ". $text2;
        if (defined $node1->getFirstChild){
          $node1->getFirstChild->setData($text1);}
        else{
          my $doc=$node1->getOwnerDocument;
          my $node_text=$doc->createTextNode("$text1");
          $node1->appendChild($node_text);
        }
      }
    }
  }
}

```

```

    }
  }
else #SE TRATA DEL NODO WBBLIB QUE PUEDE
    #TENER UN CONTENIDO ESPECIAL
{
  my $hijos2=$node2->getChildNodes;
  my $owner1=$node1->getOwnerDocument;
  my $n=$hijos2->getLength;
  for (my $i=0;$i<$n;$i++)
  {
    my $n_aux=$hijos2->getLength;
    if ($n_aux != $n) {$n=$n_aux;$i=$i-1;}
    my $aux=$hijos2->item($i);
    $aux->setOwnerDocument($owner1);
    $node1->appendChild($aux);
  }
}
}
if ($mode2 eq "prepend")
{
  if ($node1->getNodeName ne "wbbLib")
  {
    my $text1;
    my $text2;
    if ($node1->hasChildNodes)
    {$text1=$node1->getFirstChild->getData;}
    else {$text1="";}
    if ($node2->hasChildNodes)
    {$text2=$node2->getFirstChild->getData;}
    else {$text2="";}
    $text1=$text2. " ". $text1;
    if (defined $node1->getFirstChild){
      $node1->getFirstChild->setData($text1);}
    else{
      my $doc=$node1->getOwnerDocument;
      my $node_text=$doc->createTextNode("$text1");
      $node1->appendChild($node_text);
    }
  }
}
else #SE TRATA DEL NODO WBBLIB QUE PUEDE TENER UN
    #CONTENIDO ESPECIAL

```

```

    {
        my $hijos2=$node2->getChildNodes;
        my $first=$node1->getFirstChild;
        for (my $i=0;$i<$hijos2->getLength;$i++)
        {
            my $owner1=$node1->getOwnerDocument;
            my $aux=$hijos2->item($i);
            $aux->setOwnerDocument($owner1);
            $node1->insertBefore($aux,$first);
        }
    }
}
}

#####
#La función EliminaNodos
#####

sub EliminaNodos
{
    my $arbol=$_[0];
    #RECORREMOS EL ARBOL Y ELIMINAMOS LOS NODOS CON MODE=DELETE
    my $list=$arbol->getElementsByTagName("*",0);
    my $n=$list->getLength;
    for (my $i=0;$i<$n;$i++)
    {
        my $node=$list->item($i);
        my $child=$node->getChildNodes;
        #SI ES UN NODO TERMINAL MIRAMOS SU ATRIBUTO MODE.
        if (NodoTerminal($node))
        {
            {
                if ($node->getAttribute("mode") eq "delete")
                {
                    $arbol->removeChild($node);
                }
            }
        }
        else {EliminaNodos($node);}
    }
}
}

```

```
#####
#FUNCIONES PARA LA INTERFAZ CON LOS ÁRBOLES DOM
#####
```

```
#####
#La función leeTXT
#####
```

```
sub leeTXT
{
    my $doc=$_[0];
    my $nodo=$_[1];
    my $aux=$doc;
    my @camino=split(/\./,$nodo);
    #Buscamos el nodo en el arbol
    foreach $nodos (@camino)
    {
        $aux=$aux->getElementsByTagName($nodos,0);
        if (defined $aux){
            $aux=$aux->item(0);}
        else {last;}
    }
    if (defined $aux)
    {
        # UN NODO TIENE UN VALOR O HIJOS(hay que distinguir)
        my $nodo_buscado=$aux;
        my $contenido=$nodo_buscado->getChildNodes;
        my $n=$contenido->getLength;
        if ($n==0){return "";}
        else{
            if ($n == 1 && $nodo_buscado->getFirstChild->getNodeTypeName eq
                "TEXT_NODE")
            {
                my $aux=$nodo_buscado->getFirstChild->getData;
                #ELIMINAMOS LOS ESPACIOS EN BLANCO O
                #RETORNOS DE CARRO QUE HUBIESE AL FINAL O AL PRINCIPIO
                $aux =~ s/^\s*(.*?)\s*$/$1/;
                return $aux;
            }
            else{

```

```

        #RECORRO CADA UNO DE LOS HIJOS Y EN FORMATO
        #TEXTO LOS CONCATENO PARA LA SALIDA.
        $contenido=$nodo_buscado->getChildNodes;
        $n=$contenido->getLength;
        my $salida="";
        for (my $j = 0;$j < $n; $j++){
            $salida = $salida . $contenido->item($j)->toString;}
        if ($salida=~ /\n\s*([^\s].*[\s])[\s\n]*/ ||
            $salida=~ /\n\s*([^\s])[\s\n]*/)
        {$salida=$1;}
        return $salida;
    }
}
else
{
    #print "Nodo buscado:\"$nodo\" en leeTXT no encontrado\n";
    return "";
}
}

```

```

#####
#La función leeDOM
#####

```

```

sub leeDOM
{
    my $doc=$_[0];
    my $aux=$doc;
    my $nodo=$_[1];
    my @camino=split(/\./,$nodo);
    my $nombre_nodo=$camino[$#camino];
    my $padre;
    foreach $nodo (@camino)
    {
        $aux=$aux->getElementsByTagName($nodo,0);
        $aux=$aux->item(0);
    }
    if (defined $aux){
        $nodo_buscado=$aux;}
}

```

```

        else {print "Nodo buscado en leeDOM no encontrado\n";}
        return $nodo_buscado;
    }
#####
#La función escribeTXT
#####

sub escribeTXT
{
    #SOBREESCRIBE EL VALOR SI EXISTE
    my $doc=$_[0];
    my $aux=$doc;
    my $nodo=$_[1];
    my $valor_escritura=$_[2];
    my @camino=split(/\./,$nodo);
    my $nombre_nodo=$camino[$#camino];
    my $padre;
    foreach $nodos (@camino)
    {
        $aux=$aux->getElementsByTagName($nodos,0);
        $aux=$aux->item(0);
        if (defined $aux) {$padre=$aux;}
        else {last;}
    }
    $nodo_buscado=$aux;
    my $e;
    if (defined $nodo_buscado)
    {
        $e=$nodo_buscado->getChildNodes;
        if (NodoTerminal($nodo_buscado)){
            #si el nodo estaba machacamos su valor
            if ($e->getLength==0)
            {
                my $node=$doc->createTextNode("$valor_escritura");
                $nodo_buscado->appendChild($node);
            }
            if ($e->getLength==1 &&
                $e->item(0)->getNodeTypeName eq "TEXT_NODE")
            {
                $nodo_buscado->getFirstChild->setData($valor_escritura);
            }
        }
    }
}

```

```

        else #Nodo wbbLib
        {
            while ($e->getLength > 0)
            {
                $nodo_buscado->removeChild($nodo_buscado->geFirstChild);
            }
            my $node_text=$doc->createTextNode("$valor_escritura");
            $nodo_buscado->appendChild($node_text);
        }
    }
    else
    {
        print "No se puede escribir en el nodo: $nombre_nodo. ";
        print "Tiene elementos hijos.\n";
    }
}
else #SI EL NODO NO EXISTE LO CREAMOS Y SE LOS PONEMOS COMO HIJO AL PADRE
{
    if (defined $padre)
    {
        if (defined $doc->getOwnerDocument)
        {$doc=$doc->getOwnerDocument;}
        my $node=$doc->createElement($nombre_nodo);
        my $node_text=$doc->createTextNode("$valor_escritura");
        $padre->appendChild($node);
        $node->appendChild($node_text);
    }
    else #EL NODO HAY QUE CREARLO COMO HIJO DEL RAIZ
    {
        my $padre=$doc;
        if (defined $doc->getOwnerDocument)
        {$doc=$doc->getOwnerDocument;}
        my $node=$doc->createElement($nombre_nodo);
        my $node_text=$doc->createTextNode("$valor_escritura");
        $padre->appendChild($node);
        $node->appendChild($node_text);
    }
}
}

```

```
#####
```

```

#La función escribeDOM
#####

sub escribeDOM
{
#machacaria el valor si existe
  my $doc=$_[0];
  my $aux=$doc;
  my $nodo=$_[1];
  my $nodo_escritura=$_[2];
  my @camino=split(/\./,$nodo);
  my $padre;
  for (my $nodos=0;$nodos<=$#camino;$nodos++)
  {
    $aux=$aux->getElementsByTagName($camino[$nodos],0)->item(0);
    if (defined $aux && $nodos != $#camino) {$padre=$aux;}
    else {last;}
  }
  if (! defined($padre)) {$padre=$doc;}
  #EL NOMBRE DEL NODO EN EL CAMINO
  #DEBE COINCIDIR CON EL NOMBRE DEL NODO A ESCRIBIR
  my $name1=$nodo_escritura->getNodeName;
  my $name2=$camino[$#camino];
  if ($name1 ne $name2)
  {print "El nombre exigido no es igual al del nodo.Se ha escrito como
    \"$name1\".\n";}
  if (defined $aux)
  {
    #SI EL NODO ESTABA SOBREESCRIBIMOS SU VALOR
    my $owner;
    if (defined $doc->getOwnerDocument)
    { $owner=$doc->getOwnerDocument;}
    else {$owner=$doc;}
    $nodo_escritura->setOwnerDocument($owner);
    $padre->replaceChild($nodo_escritura,$aux);
  }
  else #SI EL NODO NO EXISTE SE LO PONEMOS COMO ULTIMO HIJO AL PADRE
  {
    #PARA PONER EL NODO TENGO QUE CAMBIARLE SU PROPIETARIO AL DOCUMENTO
    #AL QUE SE LO VOY A PEGAR.
    my $owner;

```

```

        if (defined $doc->getOwnerDocument)
        {$owner=$doc->getOwnerDocument;}
        else {$owner=$doc;}
        $nodo_escritura->setOwnerDocument($owner);
        $padre->appendChild($nodo_escritura);

    }
}

#####
#La función anadeFinalTXT
#####

sub anadeFinalTXT
{
    my $doc=$_[0];
    my $aux=$doc;
    my $nodo=$_[1];
    my $valor_escritura=$_[2];
    my $nodo_buscado;
    my $contenido;
    my @camino=split(/\./,$nodo);
    my $nombre_nodo=$camino[$#camino];
    my $padre;
    foreach $nodos (@camino)
    {
        $aux=$aux->getElementsByTagName($nodos,0);
        $aux=$aux->item(0);
        if (defined $aux) {$padre=$aux;}
        else {last;}
    }
    $nodo_buscado=$aux;
    my $e=$nodo_buscado->getLength;
    if (defined $nodo_buscado)
    {
        if ($e->getLength==0)
        {
            my $node=$doc->createTextNode("$valor_escritura");
            $nodo_buscado->appendChild($node);
        }
        if ($e->getLength==1 && $e->item(0)->getNodeTypeName eq "TEXT_NODE")

```

```

        {
            my $texto=$nodo_buscado->getFirstChild->getData;
            $nodo_buscado->getFirstChild->setData(
                $texto." $valor_escritura");
        }
        else #NODO WBBLIB
        {
            my $node_text=$doc->createTextNode("$valor_escritura");
            $nodo_buscado->appendChild($node_text);
        }
    }
    else #SI EL NODO NO EXISTE LO CREAMOS Y
        #SE LO PONEMOS COMO HIJO AL PADRE
    {
        if (defined $padre)
        {
            if (defined $doc->getOwnerDocument)
            {$doc=$doc->getOwnerDocument;}
            my $node=$doc->createElement($nombre_nodo);
            my $node_text=$doc->createTextNode("$valor_escritura");
            $padre->appendChild($node);
            $node->appendChild($node_text);
        }
        else #EL NODO HAY QUE CREARLO COMO HIJO DEL RAIZ
        {
            if (defined $doc->getOwnerDocument)
            {$document=$doc->getOwnerDocument;}
            else{ $document=$doc;}
            my $node=$document->createElement($nombre_nodo);
            my $node_text=$document->createTextNode("$valor_escritura");
            $doc->appendChild($node);
            $node->appendChild($node_text);
        }
    }
}

```

```

#####
#La función anadePrincipioTXT
#####

```

```

sub anadePrincipioDOM

```

```

{
  my $doc=$_[0];
  my $aux=$doc;
  my $nodo=$_[1];
  my $nodo_escritura=$_[2];
  my @camino=split(/\./,$nodo);
  my $padre;
  for (my $nodos=0;$nodos<=$#camino;$nodos++)
  {
    $aux=$aux->getElementsByTagName($camino[$nodos],0)->item(0);
    if (defined $aux && $nodos != $#camino) {$padre=$aux;}
    else {last;}
  }
  if (! defined($padre)) {$padre=$doc;}

  if (defined $aux)
  {
    #SI EL NODO ESTABA LO ELIMINAMOS Y PONEMOS EL NUEVO AL PRINCIPIO
    $padre->removeChild($aux);
  }
  #PONGO AL NODO COMO PRIMER HIJO DE SU PADRE
  my $owner;
  if (defined $doc->getOwnerDocument)
  {$owner=$doc->getOwnerDocument;}
  else {$owner=$doc;}
  $nodo_escritura->setOwnerDocument($owner);
  my $first_child=$padre->getFirstChild;
  $padre->insertBefore($nodo_escritura,$first_child);
}

#####
#La función anadeFinalDOM
#####

sub anadeFinalDOM
{
  my $doc=$_[0];
  my $aux=$doc;
  my $nodo=$_[1];
  my $nodo_escritura=$_[2];
  my @camino=split(/\./,$nodo);
  my $padre;

```

```

for (my $nodos=0;$nodos<=$#camino;$nodos++)
{
    $aux=$aux->getElementsByTagName($camino[$nodos],0)->item(0);
    if (defined $aux && $nodos != $#camino) {$padre=$aux;}
    else {last;}
}
if (! defined($padre)) {$padre=$doc;}

if (defined $aux)
{
    #SI EL NODO ESTABA LO QUITAMOS Y PONEMOS EL NUEVO AL FINAL
    $padre->removeChild($aux);
}
#PONGO AL NODO COMO ULTIMO HIJO DE SU PADRE
my $owner;
if (defined $doc->getOwnerDocument)
{$owner=$doc->getOwnerDocument;}
else {$owner=$doc;}
$nodo_escritura->setOwnerDocument($owner);
$padre->appendChild($nodo_escritura);
}

```

```

#####
#La función anadePrincipioTXT
#####

```

```

sub anadePrincipioTXT
{
    my $doc=$_[0];
    my $aux=$doc;
    my $nodo=$_[1];
    my $valor_escritura=$_[2];
    my $nodo_buscado;
    my $contenido;
    my $padre;
    my @camino=split(/\./,$nodo);
    my $nombre_nodo=$camino[$#camino];
    foreach $nodos (@camino)
    {
        $aux=$aux->getElementsByTagName($nodos,0);
        $aux=$aux->item(0);
    }
}

```

```

    if (defined $aux) {$padre=$aux;}
}
$nodo_buscado=$aux;
my $e=$nodo_buscado->getChildNodes;
if (defined $nodo_buscado)
{
    if ($e->getLength==0)
    {
        my $node=$doc->createTextNode("$valor_escritura");
        $nodo_buscado->appendChild($node);
    }
    if ($e->getLength==1 && $e->item(0)->getNodeTypeName eq "TEXT_NODE")
    {
        my $texto=$nodo_buscado->getFirstChild->getData;
        $nodo_buscado->getFirstChild->setData("$valor_escritura." $texto");
    }
    else #NODO WBBLIB
    {
        my $node_text=$doc->createTextNode("$valor_escritura");
        my $first=$nodo_buscado->getFirstChild;
        $nodo_buscado->insertBefore($node_text,$first);
    }
}
else #EL NODO NO EXISTE
{
    if (defined $padre)
    {
        if (defined $doc->getOwnerDocument)
        {$doc=$doc->getOwnerDocument;}
        my $node=$doc->createElement($nombre_nodo);
        my $node_text=$doc->createTextNode("$valor_escritura");
        $padre->appendChild($node);
        $node->appendChild($node_text);
    }
    else #EL NODO HAY QUE CREARLO COMO HIJO DEL RAIZ
    {
        if (defined $doc->getOwnerDocument)
        {$document=$doc->getOwnerDocument;}
        else{ $document=$doc;}
        my $node=$document->createElement($nombre_nodo);
        my $node_text=$document->createTextNode("$valor_escritura");
    }
}

```

```

        $doc->appendChild($node);
        $node->appendChild($node_text);
    }
}
}

```

```

#####
#La función NodoTerminal
#####

```

```

sub NodoTerminal
{
    my $nodo=$_[0];
    my $nodes=$nodo->getChildNodes;
    my $n=$nodes->getLength;
    if ($n == 0){return 1;}
    else
    {
        if ($n == 1 && $nodes->item(0)->getNodeTypeName eq
            "TEXT_NODE") {return 1;}
        if ($nodo->getNodeName eq "wbbLib")
        {
            my $result=1;
            for (my $i=0;$i<$n;$i++)
            {
                if (!(($nodes->item($i)->getNodeTypeName eq "TEXT_NODE") ||
                    ($nodes->item($i)->getNodeTypeName eq "ELEMENT_NODE" &&
                    $nodes->item($i)->getNodeName eq "inline"))
                {$result=0; }
            }
            return $result;
        }
        if ($nodo->getNodeName eq "wbbProcLib" ||
            $nodo->getNodeName eq "wbbClassLib")
        {
            my $result=1;
            for (my $i=0;$i<$n;$i++)
            {
                if (!($nodes->item($i)->getNodeTypeName eq "ELEMENT_NODE" &&
                    $nodes->item($i)->getFirstChild->getNodeTypeName

```

```

        eq "TEXT_NODE"))
    {$result=0; }
}
return $result;
}
if ($nodo->getNodeName eq "wbbLangs")
{
    my $result=1;
    for (my $i=0;$i<$n;$i++)
    {
        if (!($nodes->item($i)->getNodeTypeName eq "ELEMENT_NODE" &&
            $nodes->item($i)->getNodeName eq "lang"))
        {$result=0; }
    }
    return $result;
}
}
}

```

6.1.2 Módulo DomWrapper

```

package DomWrapper;
use XML::DOM;
#ESTE PAQUETE VA A SIMULAR QUE CUANDO LOS PROCESADORES ANTIGUOS ACCEDAN
#AL HASH %VAR SE ACCEDA AL ARBOL DOM QUE REALMENTE IMPLEMENTA A VAR
sub TIEHASH
{
    return bless {};
}
sub STORE
{
    my $clave=$_[1];
    my $valor=$_[2];
    #LE PONEMOS A LA CLAVE EL PREFIJO webber.
    $clave="webber".".".$clave;
    my $arbol=\$Webber::arbol_final;
    &Webber::escribeTXT($$arbol,$clave,$valor);
}
sub FETCH
{

```

```

my $clave=$_[1];
$clave="webber".".".$clave;
my $arbol=\$Webber::arbol_final;
return &Webber::leeTXT($$arbol,$clave);
}
sub EXISTS
{
my $clave=$_[1];
$clave="webber".".".$clave;
my $arbol=\$Webber::arbol_final;
my $aux= &Webber::leeTXT($$arbol,$clave);
if ($aux eq ""){return 1;}
else {return 0;}
}

1;

```

6.2 Código fuente del interface gráfica de Webber

```

#!/usr/bin/perl
use XML::DOM;
use Cwd;
use Tk;
use Shell;
my $file_ed="";#SI VALE DISTINTO DE "" ES EL
                #FICHERO ACTUALMENTE EN EDICION
my $cont_ed="";#SI VALE DISTINTO DE "" ES EL
                #CONTENIDO DEL FICHERO EDICION
my $file_conf;
if ($in=shift @ARGV) #TIENE QUE SER INVOCADO CON UN FICHERO
{
    my @conf_aux;
    #CARGAMOS EL FICHERO DE CONFIGURACION
    Parser_conf("webber.conf",\@conf_aux);
    $file_conf=$conf_aux[0];

    $file_webber=$in;
    #FICHERO CON EL QUE SE VA A EJECUTAR EL WEBBER

```

6.2. CÓDIGO FUENTE DEL INTERFACE GRÁFICA DE WEBBER 137

```
$mw=MainWindow->new;
$mw->title("WEBBER v2.0");
#CREAMOS EL MENU DE AYUDA
$menu_help=$mw->Menubutton(-text=>"Help",
    -menuitems => [['command'=>"Help",
        -command=>\&MostrarAyuda ],
        ['command'=>"About",
        -command=>\&About]]);
#CREAMOS CUADRO DE TEXTO DEL COMANDO A EJECUTAR
$text_command=$mw->Entry(-width=> 50,
    -background=>'salmon1',-cursor=>'pencil');
$text_command->insert('end','./webber.pl");
#CREAMOS EL MENU FILE
$menu_file=$mw->Menubutton(-text=>"File",
    -menuitems => [['command'=>"Save",
        -command=>\&SalvarFichero ],
        ['command'=>"Exit",
        -command=>sub{exit}]]->grid(
    $menu_help,"x","x","x","x");
#CREAMOS EL BOTON DE EJECUCION
$b_run=$mw->Button(-text=>"Run",-background=>'grey70',
    -command=>\&EjecutaWebber)->grid(
    $text_command,"-","-","-","-",-sticky=>"w");

#CREAMOS LOS CHECKBUTTONS CON LAS OPCIONES DEL WEBBER
$c_f=$mw->Checkbutton( -text=>" -f",
    -variable=>\$c_fvalue,
    -background=>'grey50',
    -command=>sub
    {
        if ($c_fvalue)
            {InsertaComando("-f")}
        else {EliminaComando("-f")} });
$c_d=$mw->Checkbutton( -text=>" -d",
    -variable=>\$c_dvalue,
    -background=>'grey50',
    -command=>sub
    {
        if ($c_dvalue)
            {InsertaComando("-d")}
```

```

                                else {EliminaComando("-d")} });
$c_m=$mw->Checkbutton( -text=>" -m",
                        -variable=>\$c_mvalue,
                        -background=>'grey50',
                        -command=>sub
                        {
                            if ($c_mvalue)
                                {InsertaComando("-m")}
                            else {EliminaComando("-m")} });
#CREAMOS EL CUADRO DE TEXTO PARA ABRIR UN NUEVO FICHERO
$c_r=$mw->Checkbutton( -text=>" -C",
                        -variable=>\$c_rvalue,
                        -background=>'grey50',
                        -command=>sub{
                            if ($c_rvalue)
                                {InsertaComando("-C")}
                            else {EliminaComando("-C")} })->grid(
                                $c_f,$c_d,$c_m,"x","x");

$text_file=$mw->Entry(-width=>40,-textvariable=>
                    \$file_webber,-relief=>'sunken');
$text_file->bind("<Return>",&CargarFichero);
$label=$mw->Label(-text=>"Open File          ",-background=>
                "aliceblue")->grid(
                    "-",$text_file,"-","-","-",-sticky=>"w");

#CREAMOS EL EDITOR DE TEXTOS Y LA LISTA DE HERENCIA
$ed_txt=$mw->Scrolled("Text",-scrollbars=>
                    'se',-width=>50,-height=>10);
$lb=$mw->Scrolled("Listbox",-scrollbars=>'se',
                    -selectmode=>"single")->grid(
                        "-","-","-",$ed_txt,"-");
$lb->bind('<Button-1>',sub{EditarFichero($lb->
                    get($lb->curselection()))});

#CREAMOS EL CUADRO DE TEXTO DEL DEBUGGER
$deb_txt=$mw->Scrolled("Text",-scrollbars=>'e',-width=>80,
                        -background=>"snow",-relief=>'ridge',
                        -state=>'disabled',-cursor=>'hand2',
                        -height=>5)->grid("-","-","-","-","-");

```

6.2. CÓDIGO FUENTE DEL INTERFACE GRÁFICA DE WEBBER 139

```
#INSERTO AL FICHERO DE ENTRADA EN LA LINEA DE COMANDO
$text_command->insert('end'," $in");
CargarFichero();
MainLoop;
}
else
{print "No hay fichero de entrada.\n";}

#####
#CADA UNA DE LAS FUNCIONES QUE EL XWEBBER NECESITA
#####

#####
# La función InsertaComando
#####

sub InsertaComando
{
    #AÑADE EN LA LINEA DE COMANDO LA OPCION ELEGIDA
    #ENTRE LA ULTIMA OPCION Y EL NOMBRE DEL FICHERO
    my $in=$_[0];
    my $act=$text_command->get();
    @words=split(/\s/, $act);
    my $after=$words[0];
    for (my $i=1;$i< $#words;$i++)
    {
        $after=$after." ".$words[$i];
    }
    $after=$after." ".$in." ".$words[$#words];
    $text_command->delete(0,'end');
    $text_command->insert('end',$after);
}

#####
# La función EliminaComando
#####

sub EliminaComando
{
    #ELIMINA DE LA LINEA DE COMANDO LA OPCION ELEGIDA
    my $in=$_[0];
```

```

my $act=$text_command->get();
@words=split(/\s/,$act);
my $after=$words[0];
if ($in ne "-C")
{
    for (my $i=1;$i<=$#words;$i++)
    {
        if ($words[$i] ne $in)
        {$after=$after." ".$words[$i];}
    }
}
else #SI ES LA OPCION -C A LO MEJOR HAY QUE BORRAR TB EL FICHERO DE
    #CONFIGURACION SI HA SIDO ESCRITO
{
    for (my $i=1;$i<=$#words;$i++)
    {
        if ($words[$i] ne $in)
        {$after=$after." ".$words[$i];}
        else
        {
            if (($words[$i+1] !~ /\w/) && ($i+1 != $#words))
            {$i=$i+1;}
        }
    }
}
$text_command->delete(0,'end');
$text_command->insert('end',$after);
}

```

```

#####
# La función Parser_conf
#####

```

```

sub Parser_conf
#Es igual que Parser pero se utiliza para el fichero de
#configuracion ya que necesitamos que devuelva una lista
#arboles y no una lista de nombres de ficheros como
#hace el Parser.
{

    # ESTA FUNCION HACE EL PARSEER DEL FICHERO,

```

6.2. CÓDIGO FUENTE DEL INTERFACE GRÁFICA DE WEBBER 141

```
# TRATA LA POSIBLE HERENCIA EN EL
# ATRIBUTO INHERIT Y EJECUTA LAS
# INSTRUCCIONES DE PROCESAMIENTO <?...?>
# SE LE PASA COMO PARAMETRO EL NOMBRE DEL FICHERO
my $arboles=$_[1];
my $doc=$_[0];
my $stdin=$_[2];
my $file;
my $parser = new XML::DOM::Parser(ProtocolEncoding=>'ISO-8859-1');
if ($stdin ne "STDIN")
{$file = $parser->parsefile($doc);}
else {$file = $parser->parse($doc);}
my $aux = $file->getElementsByTagName ("webber",0);
my $length=$aux->getLength;
my $padre;
my $nodes;
my $n;
if ($length==0)
{
    print "ERROR...NO HAY NINGUN <webber> DE ENTRADA.\n";
    $n=0;
}
else
{
    $padre=$aux->item(0);
    $nodes = $aux->item(0)->getChildNodes;
    $n = $nodes->getLength;
}

my $inherit=$aux->item(0)->getAttributeNode("inherit");
if (defined $inherit)
{$inherit=$aux->item(0)->getAttributeNode("inherit")->getValue;}
else {$inherit="";}
#EN $inherit TENEMOS EL POSIBLE VALOR DEL FILE DE HERENCIA

for (my $i = 0; $i < $n; $i++)
{
    #HAY QUE CALCULAR LA LONGITUD DE LA LISTA A CADA ITERACION PORQUE
    #PUEDEN DESAPARECER HIJOS DE LA LISTA Y LOS INDICES VARIAN.DE LA
    #LISTA SE PIERDEN AQUELLAS P.I MAL CONSTRUIDAS
```

```

my $n_aux=$nodes->getLength;
if ($n_aux != $n) {$n=$n_aux;$i=$i-1;}
my $node = $nodes->item($i);
my $type=$node->getNodeTypes;
ParserNode($node,$padre,$file);
}
#SI HAY HERENCIA,BUSCAMOS EL FICHERO EN LOS DIRECTORIOS DEFINIDOS EN
# WBBCLASSLIB Y METEMOS EN LA LISTA DICHO ARBOL
if ($inherit ne "")
{
  if ($inherit !~ /\// && !(-T $inherit))
  #TENEMOS QUE BUSCAR EL FICHERO
  {
    my $wbbClassLib=leeDOM($file_conf,"webber.wbbClassLib");
    #LEEMOS TODOS LOS DIRECTORIOS DE CADA UNO DE SUS HIJOS
    my $dirs=$wbbClassLib->getElementsByTagName("*",0);
    my $long=$dirs->getLength;
    my @dir_herencia;
    for (my $i=0;$i<$long;$i++)
    {
      my $actual=$dirs->item($i);
      if ($actual->getFirstChild->getNodeTypeName eq "TEXT_NODE" &&
          $actual->getChildNodes->getLength==1)
      {
        my $dir=$actual->getFirstChild->getData;
        $dir=~/[\\n\\s]*([\\s].*[\\s])\\s*/;
        $dir=$1;
        push(@dir_herencia,$dir);
      }
    }
  }
  for (my $i=0;$i<=$#dir_herencia;$i++){
    $dir_herencia[$i]=~/^\\s*(.*)\\s*$/;
    my $dir=$1;
    if ($dir=~/. / && !(-T $inherit))
    #MIRAMOS SI EL FICHERO ESTA EN LOS DIRECTORIOS
    {
      my $actual=getcwd();
      chdir($dir);
      if (-T $inherit)
      {
        $inherit=$dir."/". $inherit;
      }
    }
  }
}

```

6.2. CÓDIGO FUENTE DEL INTERFACE GRÁFICA DE WEBBER 143

```
        chdir ($actual);
        last;
    }
    chdir($actual);
}
}
}
Parser($inherit,\@$arboles);
}
#POR ULTIMO INCLUIMOS NUESTRO ARBOL

push(@$arboles,$file);

}

#####
# La función ProcesaFichero
#####

sub ProcesaFichero
{
    my $f_d=$_[0];
    my @aux = split /\// , $f_d;
    my $fichero=$aux[$#aux];
    if ($fichero=~/$wbbSourceRegExp/ && $fichero ne $wbbTemplateName)
    {
        my @array_arboles=();
        @array_arboles=LeerTemplates($file_webber,\@array_arboles);
        Parser($f_d,\@array_arboles);
        unshift(@array_arboles,"webber.conf");
        return @array_arboles;
    }
}

#####
# La función LeerTemplates
#####

sub LeerTemplates
{
    my $lista_arboles=$_[1];
```

```

my $f_d=$_[0];
my $dir_actual=getcwd();
my $wbbTemplateName=leeTXT($file_conf,"webber.wbbTemplateName");
$wbbTemplateName =~ /\n\s*([^\s].*[\s])\s*/;
$wbbTemplateName=$1;
my $wbbSourceRoot=leeTXT($file_conf,"webber.wbbSourceRoot");
$wbbSourceRoot =~ /\n\s*([^\s].*[\s])\s*/;
$wbbSourceRoot=$1;
my @aux = split /\// , $f_d;
my $dir=join "/",@aux[0..$#aux-1];
die "No puedo acceder al directorio $dir" if "$dir" ne "" and !(chdir
    ($dir));
chdir $dir_actual;#Me vuelvo a mi sitio por si la inst de antes tiene
    #efecto
#SI ME DAN LA RUTA RELATIVA DONDE ESTA EL FICHERO DEBO CALCULAR LA
#ABSOLUTA PARA LEER RECURSIVAMENTE LOS TEMPLATES
if ($dir !~ /\//)
{
    if ($dir ne "")
    {$dir=$dir_actual."/".$dir;}
    else {$dir=$dir_actual;}
}
my $template;
if ($dir !~ /\$/){$template=$dir."/".$wbbTemplateName;}
else {$template=$dir.$wbbTemplateName;}
if (-T $template) {Parser($template,\@$lista_arboles);}
while ($dir =~ /^(.*)\// && $dir ne $wbbSourceRoot && $dir ne "/")
{
    $dir=$1;
    if ($dir ne "")
    {
        $template=$dir."/".$wbbTemplateName;
        my @aux2;
        if (-T $template)
        {
            Parser($template,\@aux2);
            my $e;
            while ($e=pop(@aux2)) {unshift(@$lista_arboles,$e);}
        }
        if ($dir eq $wbbSourceRoot || $dir eq "/"){last;}
    }
}

```

6.2. CÓDIGO FUENTE DEL INTERFACE GRÁFICA DE WEBBER 145

```
        else{last;}
    }
    return @$lista_arboles;
}

#####
# La función Parser
#####

sub Parser
{

my $arboles=$_[1];
my $doc=$_[0];
my $parser = new XML::DOM::Parser(ProtocolEncoding=>'ISO-8859-1');
my $file = $parser->parsefile($doc);
my $aux = $file->getElementsByTagName ("webber",0);
my $length=$aux->getLength;
my $padre;
my $nodes;
my $n;
    if ($length==0)
    {
        print "ERROR...NO HAY NINGUN <webber> DE ENTRADA.\n";
        $n=0;
    }
    else
    {
        $padre=$aux->item(0);
        $nodes =$aux->item(0)->getChildNodes;
        $n = $nodes->getLength;

    }

    for (my $i = 0; $i < $n; $i++)
    {
#HAY QUE CALCULAR LA LONGITUD DE LA LISTA A CADA ITERACION PORQUE
#PUEDEN DESAPARECER HIJOS DE LA LISTA Y LOS INDICES VARIAN.DE LA
#LISTA SE PIERDEN AQUELLAS P.I MAL CONSTRUIDAS
        my $n_aux=$nodes->getLength;
        if ($n_aux != $n) {$n=$n_aux;$i=$i-1;}
    }
}
```

```

    my $node = $nodes->item($i);
    my $type=$node->getNodeTypes;
    ParserNode($node,$padre,$file);
}

my $inherit=$aux->item(0)->getAttributeNode("inherit");
if (defined $inherit)
{
    $inherit=$aux->item(0)->getAttributeNode("inherit")->getValue;
}
else {
    $inherit="";
}
#EN $inherit TENEMOS EL POSIBLE VALOR DEL FICHERO DE HERENCIA

#SI HAY HERENCIA,BUSCAMOS EL FICHERO EN LOS DIRECTORIOS
#DEFINIDOS EN
# WBBCLASSLIB Y METEMOS EN LA LISTA DICHO ARBOL
if ($inherit ne "")
{
    if ($inherit !~ /\:\/\/ && !(-T $inherit))
    #TENEMOS QUE BUSCAR EL FICHERO
    {
        my $wbbClassLib=leeDOM($file_conf,"webber.wbbClassLib");
        #LEEMOS TODOS LOS DIRECTORIOS DE CADA UNO DE SUS HIJOS
        my $dirs=$wbbClassLib->getElementsByTagName("*",0);
        my $long=$dirs->getLength;
        my @dir_herencia;
        for (my $i=0;$i<$long;$i++)
        {
            my $actual=$dirs->item($i);
            if ($actual->getFirstChild->getNodeTypeName eq "TEXT_NODE" &&
                $actual->getFirstChild->getLength==1)
            {
                my $dir=$actual->getFirstChild->getData;
                $dir=~/[\\n\\s]*([\\s].*[\\s])\\s*/;
                $dir=$1;
                push(@dir_herencia,$dir);
            }
        }
        for (my $i=0;$i<=@dir_herencia;$i++){
            $dir_herencia[$i]=~/^\\s*(.*)\\s*/;
            my $dir=$1;
            if ($dir=~/. / && !(-T $inherit))

```

6.2. CÓDIGO FUENTE DEL INTERFACE GRÁFICA DE WEBBER 147

```
#MIRAMOS SI EL FICHERO ESTA EN LOS DIRECTORIOS
{
    my $actual=getcwd();
    chdir($dir);
    if (-T $inherit)
    {
        $inherit=$dir."/".$inherit;
        chdir ($actual);
        last;
    }
    chdir($actual);
}
}
}
Parser($inherit,\@$arboles);
}
#POR ULTIMO INCLUIAMOS NUESTRO ARBOL

push(@$arboles,$doc);

}
```

```
#####
# La función ParserNode
#####
```

```
sub ParserNode
{
#RECIBE EL NODO A EXAMINAR Y EL PADRE POR SI
#SE ENCUENTRA UNA P.I SUSTITUYA Y EL DOCUMENTO RAIZ
# SU VALOR
#SOLO TENDREMOS EN CUENTA LOS NODOS DE TIPO
# 1(ELEMENTOS) Y 7(INST PROC)
```

```
    my $nodo=$_[0];
    my $padre=$_[1];
    my $doc=$_[2];
    my $type=$nodo->getNodeType;
    if ($type == 7)#si es una inst. de procesamiento
    {
        my $target=$nodo->getTarget;
```

```

my $data=$nodo->getData;
if ($target eq "eval")#es un eval de perl
{
    my $aux=eval("$data");
    my $new_nodo=$doc->createTextNode("$aux");
    $padre->replaceChild($new_nodo,$nodo);

}

elsif ($target eq "system")#es una llamada al sistema
{
    my $aux;
    my $instruccion=CreaInstruccion($data);
    $aux=eval($instruccion);
    my $new_nodo=$doc->createTextNode("$aux");
    $padre->replaceChild($new_nodo,$nodo);

}
elsif ($target eq "perl")#es una funcion de perl
{
    my $aux=eval("&" . $data . ";"");
    my $new_nodo=$doc->createTextNode("$aux");
    $padre->replaceChild($new_nodo,$nodo);
}
else
{
    print "Instruccion de procesamiento no valida:";
    print $nodo->toString;
    $padre->removeChild($nodo);
}
}

if ($type==1) #Es un nodo normal
{
    my $hijos = $nodo->getChildNodes;
    my $num_hijos=$hijos->getLength;
    for (my $k = 0; $k < $num_hijos ; $k++)
    {
        my $hijo = $hijos->item ($k);
        ParserNode($hijo,$nodo,$doc);
    }
}

```

6.2. CÓDIGO FUENTE DEL INTERFACE GRÁFICA DE WEBBER 149

```
}

}

#####
# La función CreaInstruccion
#####

sub CreaInstruccion
{
#A PARTIR DEL CAMPO DATA DE UNA INSTRUCCION DE PROCESAMIENTO CREA LA
#INSTRUCCION A EJECUTAR EN EL SISTEMA.
#DE "cp file1 file2" PASA A "cp("file1","file2");"
  my $lista=$_[0];
  my @arg=split /\s/ , $lista;
  my $command=shift(@arg);
  for (my $i=0;$i<=$#arg;$i++){
    $arg[$i]="\" . $arg[$i] . "\"";
  }
  my $aux=join(" ",@arg);
  my $inst=$command."("$aux.")";
  return $inst;
}

#####
# La función leeTXT
#####

sub leeTXT
{
  my $doc=$_[0];
  my $nodo=$_[1];
  my $aux=$doc;
  my @camino=split(/\./,$nodo);
  #Buscamos el nodo en el arbol
  foreach $nodos (@camino)
  {
    $aux=$aux->getElementsByTagName($nodos,0);
    if (defined $aux){
      $aux=$aux->item(0);}
  }
}
```

```

        else {last;}
    }
    if (defined $aux)
    {
        # Un nodo tiene un valor o hijos(hay que distinguir)
        my $nodo_buscado=$aux;
        my $contenido=$nodo_buscado->getChildNodes;
        my $n=$contenido->getLength;
        if ($n==0){return " ";}
        else{
            if ($n == 1 && $nodo_buscado->getFirstChild->getNodeTypeName eq
                "TEXT_NODE"){ return $nodo_buscado->getFirstChild->getData;}
            else{
                #recorro cada uno de los hijos y en formato texto los
                #concateno para la salida
                $contenido=$nodo_buscado->getChildNodes;
                $n=$contenido->getLength;
                my $salida="";
                for (my $j = 0;$j < $n; $j++){
                    $salida = $salida . $contenido->item($j)->toString;}
                return $salida;
            }
        }
    }
    else
    {
        print "Nodo buscado:\"$nodo\" en leeTXT no encontrado\n";
        return "";
    }
}

#####
# La función leeDOM
#####

sub leeDOM
{
    my $doc=$_[0];
    my $aux=$doc;
    my $nodo=$_[1];
    my @camino=split(/\./,$nodo);

```

6.2. CÓDIGO FUENTE DEL INTERFACE GRÁFICA DE WEBBER 151

```
my $nombre_nodo=$camino[$#camino];
my $padre;
foreach $nodo (@camino)
{
    $aux=$aux->getElementsByTagName($nodo,0);
    $aux=$aux->item(0);
}
if (defined $aux){
    $nodo_buscado=$aux;}
else {print "Nodo buscado en leeDOM no encontrado\n";}
return $nodo_buscado;
}

#####
# La función CargarFichero
#####

sub CargarFichero
{

    #CALCULAMOS LA LISTA ORDENADA DE FICHEROS QUE PARTICIPAN
    #Y LA MOSTRAMOS EN UN LISTBOX

    my $cont_ant="";
    my $cont="";
    ###PRIMERO PRESENTAMOS LA LISTA DE HERENCIA###
    my @res=();
    @res=ProcesaFichero($file_webber);
    #SI LA LISTA DEVUELTA ESTA VACIA EL
    #FICHERO NO ES VALIDO, AVISAR DEL ERROR
    $lb->delete(0,'end');#BORRAMOS LA LISTA DE HERENCIA ANTERIOR
    #ESCRIBIMOS LA NUEVA LISTA DE HERENCIA
    for ($i=0;$i<=$#res;$i++)
    {
        $lb->insert('end',$res[$i]);
    }
    #MODIFICAMOS LA LINEA DE COMANDO PARA AÑADIR EL NUEVO FICHERO
    my $act_com=$text_command->get();
    my @words_aux=split(/\s/, $act_com);
    $words_aux[$#words_aux]=$file_webber;
    $act_com=join(" ",@words_aux);
```

```

$text_command->delete(0,'end');
$text_command->insert('end',$act_com);
#COMPROBAMOS SI HA HABIDO CAMBIOS EN EL ANTIGUO
open(FILE2,$file_ed);
while (<FILE2>) {$cont_ant=$cont_ant.$_;}
close FILE2;
$cont_ed=$ed_txt->get("1.0","end");
#QUITAMOS UN RETORNO DE CARRO QUE SE CUELA AL FINAL DE ESTA VBLE
$cont_ed =~ s/(.*)\n$/1/;

if ($file_ed ne "" && $cont_ant ne $cont_ed)
#SI HAY ALGUN FICHERO EDITANDOSE PREGUNTAMOS PARA SALVAR
{
    my $mw2=MainWindow->new;
    $mw2->title("SAVE FILE");
    my $aux=$file_ed;
    my $aux_cont=$ed_txt->get("1.0","end");
    my $b_c=$mw2->Button(-text=>"Cancel",-background=>'DarkRed',
                        -command=>sub{\&Cerrar($mw2)});
    my $b_ok=$mw2->Button(-text=>"Ok",-background=>'DarkRed',
                        -command=>sub{\&CerrarSalvar($mw2,$aux,$aux_cont)});
    my $text_f=$mw2->Label(-text=>"Save File $file_ed",
                        -background=>"alice blue")->grid($b_c,$b_ok);

    Mainloop;
}
#LIMPIAMOS EL EDITOR DE TEXTOS
$ed_txt->delete("1.0","end");
$file_ed="";
#LIMPIAMOS EL DEBUGGER
$deb_txt->configure(-state=>'normal');
$deb_txt->delete("1.0","end");
$deb_txt->configure(-state=>'disabled');

}

#####
# La función EditarFichero
#####

```

6.2. CÓDIGO FUENTE DEL INTERFACE GRÁFICA DE WEBBER 153

```
sub EditarFichero
{
    my $cont="";
    my $cont_ant="";
    if ($_[0] ne $file_ed)
    {
        #ABRIMOS EL FICHERO QUE VAMOS A EDITAR
        open(FILE,$_[0]);
        while (<FILE>) {$cont=$cont.$_;}
        close FILE;
        #COMPROBAMOS SI HA HABIDO CAMBIOS EN EL ANTIGUO
        open(FILE2,$file_ed);
        while (<FILE2>) {$cont_ant=$cont_ant.$_;}
        close FILE2;
        $cont_ed=$ed_txt->get("1.0","end");
        #QUITAMOS UN RETORNO DE CARRO QUE SE CUELA AL FINAL DE ESTA VBLE
        $cont_ed =~ s/(.*)\n$/\1/;
        if ($file_ed ne "" && $cont_ant ne $cont_ed)
        {
            #SALVAR EL FICHERO DE EDICION QUE ACTUALMENTE SE ESTA EDITANDO
            my $mw2=MainWindow->new;
            #ALMACENAMOS EL NOMBRE DEL FICHERO QUE SE ESTA
            #EDITANDO Y SU CONTENIDO
            my $aux=$file_ed;
            my $aux_cont=$ed_txt->get("1.0","end");
            $mw2->title("SAVE FILE");
            my $b_c=$mw2->Button(-text=>"Cancel",-background=>'DarkRed',
                                -command=>sub{\&Cerrar($mw2)});
            my $b_ok=$mw2->Button(-text=>"Ok",-background=>'DarkRed',
                                -command=>sub{\&CerrarSalvar($mw2,$aux,$aux_cont)});
            my $text_f=$mw2->Label(-text=>"Save File $file_ed",
                                -background=>"alice blue")->grid($b_c,$b_ok);

            Mainloop;
        }
        $file_ed=$_[0];
        $ed_txt->delete("1.0",'end');
        $ed_txt->insert('end',$cont);
        $cont_ed=$cont;
    }
}
```

```
#####
# La función CerrarSalvar
#####

sub CerrarSalvar
{
    $_[0]->destroy();
    my $file=$_[1];
    my $cont=$_[2];
    SalvarFichero($file,$cont);
}

#####
# La función Cerrar
#####

sub Cerrar
{
    $_[0]->destroy();
}

#####
# La función SalvarFichero
#####

sub SalvarFichero
{
    my $aux=$_[0];
    my $cont=$_[1];
    if ($aux eq "")
    {
        $aux=$file_ed;
        $cont=$ed_txt->get("1.0","end");
    }
    open (FILE,">$aux");
    print FILE $cont;
    close FILE;
}

#####
# La función MostrarAyuda
```

6.2. CÓDIGO FUENTE DEL INTERFACE GRÁFICA DE WEBBER 155

#####

```
sub MostrarAyuda
{
    my $mw2=MainWindow->new;
    #ALMACENAMOS EL NOMBRE DEL FICHERO QUE SE
    #ESTA EDITANDO Y SU CONTENIDO.
    $mw2->title("HELP XWEBBER v1.0");
    $mw2->Label(-text=>"
        - Para abrir un fichero nuevo teclear su nombre en OPEN FILE.
        - Para ejecutar el webber pulsar en RUN.
        - Las distintas opciones del webber para la ejecucion de un fichero
          se pueden habilitar o deshabilitar con los checkbuttons.
        - Pinchando en algun fichero de la lista de herencia lo podemos
editar.
        - En el menu fichero si pinchamos en EXIT salimos de la aplicacion
          y pinchando en SAVE guardamos el actual fichero en edicion.
        - En la parte superior vemos el comando que ejecutara el Webber.",
        -background=>"alice blue",
        -justify=>'left')->pack();

    Mainloop;
}
```

#####

La función About

#####

```
sub About
{
    my $mw2=MainWindow->new;
    #ALMACENAMOS EL NOMBRE DEL FICHERO QUE SE ESTA EDITANDO Y SU CONTENIDO
    $mw2->title("ABOUT XWEBBER v1.0");
    my $about=$mw2->Label(-text=>"
        Xwebber v1.0 creado por Manuel Pérez Coca.
        Agradecimientos a Diego López y Javier Masa.",
        -background=>"alice blue",
        -justify=>'left');
    my $logo=$mw2->Photo(-file=> "logopuente.gif");
    $mw2->Label(-image=>$logo)->grid($about);
}
```

```

        Mainloop;

}

#####
# La función EjecutaWebber
#####

sub EjecutaWebber
{
    my $command= $text_command->get();
    #$command=CreaInstruccion($command);
    #my $aux=eval($command);
    #HAY QUE CAPTURAR EN LA VARIABLE AUX TODO LO QUE VAYA A STDERR
    my $aux = '$command 2>&1 1>/dev/null';
    #eval($command);
    #AUX CONTIENE LO QUE APARECERA EN EL DEBUGGER
    $deb_txt->configure(-state=>'normal');
    $deb_txt->delete("1.0","end");
    $deb_txt->insert('end',$aux);
    $deb_txt->configure(-state=>'disabled');
    #AL FINAL DE LA EJECUCION SONARA UN PITIDO PARA AVISAR
    $deb_txt->bell();
}

```

Capítulo 7

Conclusiones

Con Webber hemos conseguido ofrecer un entorno de desarrollo de contenidos Web que sigue un modelo basado en el uso de componentes, donde se reflejan los principios de la programación orientada a objetos: encapsulación, abstracción y modularidad. El modelo orientado a objetos se extiende a la gestión de los contenidos, aportando una alta capacidad de reutilización de los mismos.

El funcionamiento de Webber se basa en la colaboración de elementos específicos con una funcionalidad muy concreta que cooperan entre sí, que las funciones del núcleo de Webber van invocando conforme su ejecución es requerida. Estos elementos utilizan un espacio de memoria común que acceden de manera asociativa (un espacio de tuplas) para comunicarse entre ellos, lo que garantiza la absoluta independencia de unos respecto a otros. El resultado final de la ejecución de Webber lo conforman por tanto, el conjunto de los resultados de los recursos y procesadores que han participado en dicha ejecución.

Además se suministra un interface gráfico de usuario para el manejo de esta herramienta, el cual facilita información sobre la lista de clases heredadas por el recurso procesado, y la manipulación de los recursos que contienen la descripción de estas clases.

Capítulo 8

Bibliografía

- 1. Chuck Musciano and Bill Kennedy. *HTML y XHTML. La guía definitiva*. Ed. Anaya O'Reilly, 2001
- 2. Elizabeth Castro. *XML*. Ed. Prentice Hall, 2001
- 3. Elliotte Rusty Harold. *XML BIBLE*. IDG Books Worldwide, Inc. 1990
- 4. Hans-W. Gellersen y Martin Gaedke. Object-Oriented Web Application Development. *IEEE Internet Computing (Enero/Febrero 1999)*. Disponible en <http://computer.org/internet/>
- 5. Nancy Walsh. *Learning Perl/Tk*. Ed. O'Reilly, 1999
- 6. Larry Wall, Tom Christiansen, y Randal L. Schwartz. *Programming Perl*. Ed. O'Reilly, 1996
- 7. Martin Bryan. *SGML—An Author's Guide to the Standard Generalized Markup Language*. Addison-Wesley, 1998.
- 8. Ken Arnold y James Gosling. *El lenguaje de programación Java*. Ed. Addison-Wesley, 1997
- 9. Diego R. López. *Internet. La Red con Mayúsculas*. Ed. MAD, 1997
- 10. Thuan L. Thai. *Learning DCOM*. Ed. O'Reilly, 1999
- 11. Robert Englander. *Developing JavaBeans*. Ed. O'Reilly, 1997
- 12. Unicode. *Edición on-line de Unicode 3.0*. Disponible en www.Unicode.org/uni2book/u2.html

- 13. World Wide Consortium. *Especificación de HTML 4.0*. Disponible en

<http://www.w3.org/TR/1998/REC-html40-19980424/>

Capítulo 9

Apéndice A. Manual de usuario

9.1 Instalación de módulos

Antes de lanzar la ejecución de Webber, debemos realizar la instalación en nuestra máquina del módulo Perl `XML::DOM` que nos facilita el interface orientado a objetos para manipular los ficheros fuente y plantilla. Para ello, necesitamos además instalar otros dos módulos utilizados por este último:

- El módulo `XML-Parser` que suministra el interface para parsear documentos XML.
- El módulo `expat` que le suministra al anterior una librería de aún más bajo nivel para manipular y parsear documentos XML.

Estos tres módulos pueden ser adquiridos en la Web de **CPAN** (Comprehensive Perl Archive Network), en el URL siguiente: <http://www.cpan.org>. En este sitio web podemos encontrar todo tipo de recursos para el language de programación Perl.

Debido a la dependencia que existe entre los módulos, es decir, que el módulo `XML::DOM` necesita al módulo `XML-Parser` y que éste a su vez necesita al módulo `expat`; el orden que se debe seguir para la instalación es la siguiente: `expat`, `XML-Parser` y por último `XML::DOM`.

Todos los módulos se instalan siguiendo una metodología común que se detalla a continuación:

- Después de haber bajado el módulo de CPAN, el primer paso es realizar la descompresión y desempaquetamiento del fichero:

```
$ tar cxvz modulo.tar.gz
```

- El segundo paso es realizar la configuración, para ello debemos escribir:

```
$ cd modulo  
$ perl Makefile.PL
```

- Por último realizamos la compilación y la instalación:

```
$ make  
$ make test
```

9.2 Manual de usuario de Webber

Antes de lanzar la ejecución de Webber debemos haber preparado previamente la estructura de directorios con los ficheros fuente y plantilla del sitio del servidor Web que queremos actualizar. Una vez que hemos ubicado estos ficheros en sus directorios correspondientes y las variables que poseen tienen asignados los valores correctos, ya estaremos en condiciones de invocar Webber sobre los ficheros fuente o directorios que se desean actualizar. Además es imprescindible crear el fichero de configuración del cual Webber lee las variables necesarias para comenzar su ejecución.

En el documento donde se detalla el diseño de Webber podemos aprender cómo construir estos ficheros de configuración, ver la sintaxis que se sigue, las variables del núcleo de Webber, y los posibles valores que éstas pueden recibir.

En el apartado siguiente veremos los diferentes argumentos que se le pueden pasar a Webber y que nos permiten diferentes modos de ejecución en él.

9.2.1 Ejecutando Webber

Una vez instalados los módulos y Webber en el sistema, ya estamos en condiciones de invocar a Webber desde el shell de nuestra máquina. Las formas más simples de hacer una llamada a Webber son las siguientes:

```
$ webber Fichero  
o  
$ webber Directorio
```

En estos dos casos Webber se ejecuta utilizando el fichero de configuración definido por defecto para él. En el primer caso Webber se ejecuta procesando como fichero de entrada el identificado en el parámetro denominado **Fichero**; y en el segundo caso, Webber se ejecuta procesando todos los ficheros fuente contenidos en el directorio identificado en el parámetro **Directorio**.

También podemos invocar a Webber pasándole como argumento varios ficheros y directorios fuente, como se muestra en el siguiente comando:

```
$ webber Fichero1 Directorio2 Fichero3
```

Webber procesará en primer lugar el fichero fuente denominado **Fichero1**, en segundo lugar todos los ficheros fuente contenidos en el directorio denominado **Directorio2**, y por último el fichero denominado **Fichero3**.

9.2.2 Argumentos de Webber

Webber presenta diferentes modos de ejecución que pueden ser controlados mediante los siguientes argumentos:

- **-C:** Con esta opción le indicamos a Webber que se ejecute utilizando como fichero de configuración el indicado junto con este argumento, y no utilice el fichero de configuración por defecto **webber.conf**. Si invocamos a Webber de la siguiente forma le estamos indicando que utilice como fichero de configuración **otro.conf** y como fichero fuente **.mail** ubicado en el directorio **PruebasDeEjecucion**:

```
$ webber.pl -C otro.conf PruebasDeEjecucion/.mail
```

- **-d:** Mediante este argumento podemos poner a Webber en modo de depuración, con lo que conseguiremos que nos muestre información sobre su estado, el fichero de configuración utilizado y su estado, mientras realiza la ejecución.

Si invocamos a Webber de la siguiente forma:

```
$webber.pl -C otro.conf -d PruebasDeEjecucion/.mail
```

obtendremos la siguiente información en pantalla:

```
Debug: Usando el fichero otro.conf para configuracion.
Debug: Procesando el fichero o directorio
PruebasDeEjecucion/mail/.iris-mail.
Debug: Procesando templates del directorio
/home/manolo/PROYECTO/WEBBER/PruebasDeEjecucion/mail.
Debug: Procesando el template
/home/manolo/PROYECTO/WEBBER/PruebasDeEjecucion/mail/.template.
Debug: Procesando el template
/home/manolo/PROYECTO/WEBBER/.template.
Debug: Procesando el template
/home/manolo/PROYECTO/.template.
Debug: Llamamos al procesador:Iris2::cab
Debug: Llamamos al procesador:PrintIn::printIn
Debug: Llamamos al procesador:Iris2::pdp
```

- **-f:** Webber testea si la fecha de modificación del fichero destino es más reciente que la fecha de modificación del fichero fuente con el que se corresponde, en cuyo caso Webber no actualiza el fichero destino. Si establecemos esta opción, Webber actualizará el fichero destino aunque su fecha de modificación sea más reciente que la del fichero fuente.
- **-h:** Si invocamos a Webber con este argumento nos mostrará una breve ayuda sobre su funcionamiento y finaliza la ejecución:

```
webber [-C <configFile>] [-d] [-f] [-h] [-i] [-I] [-m] [-r]
      <tplFichero>] (dir|fichero) [(dir|fichero)... ]
```

```
-C establece un fichero diferente de configuracion
-d establece el modo de depuracion
-f fuerza a actualizar los ficheros de salida
-h muestra este mensaje
-H muestra la ayuda los procesadores instalados
-i usa la i/o estandar, y no un fichero
-I lista los procesadores instalados
-m crea los directorios destino si no existen
-r procesa recursivamente directorios
<dir>   Procesa el contenido de un directorio
<file>  Procesa un fichero fuente individual
```

Ver además webber.conf

- **-H:** Con este argumento Webber muestra la ayuda disponible por los procesadores conocidos actualmente, que serán los situados en el mismo directorio que él o en los directorios indicados en la variable `wbbProcLib`.
- **-i:** Con este argumento Webber en lugar de procesar un fichero fuente va a procesar su entrada estándar. Es importante tener en cuenta que en este modo de funcionamiento Webber no procesará ningún fichero plantilla, por lo que en el caso de que se quiera heredar alguno, se deberá utilizar el atributo `inherit` del elemento principal `webber`.
- **-I:** Mediante esta opción Webber nos muestra la lista de los procesadores conocidos por él, además de una breve descripción de ellos. Estos procesadores son los ubicados en su mismo directorio o los ubicados en los directorios indicados en `wbbProcLib`

Si invocamos a Webber con el siguiente comando:

```
$ webber -I
```

obtendremos en pantalla una información parecida a ésta de los procesadores que hay actualmente en el sistema:

```
PrintIn.pm=>PrintIn v1.0: Copia el contenido de wbbIn en wbbOut.
BodyFaq.pm=>BodyFaq 0.5: Crea un FAQ HTML con tabla de
                        contenido.
Caparse.pm=>Caparse v1.0: Extracción de variables basado
                        en un parser HTML/XML.
Capaweb.pm=>Capaweb v1.0: Extrae variables webber de datos
                        etiquetados.
DumpVars.pm=>DumpVars v1.0: Almacena en wbbOut las variables
                        seleccionadas.
Exec.pm=>Exec v1.0: Ejecuta programas y almacena la salida en
                        variables webber.
Maketoc.pm=>Maketoc v1.0: Genera una tabla de contenido.
PgpSign.pm=>PgpSign 0.2: Firma páginas HTML con PGP.
```

- **-m:** Con esta opción permitimos a Webber que vaya creando los directorios necesarios cuando está generando los ficheros de salida. Por defecto, Webber asume que esos directorios están creados. En caso contrario debemos invocarlo con esta opción habilitada.

- **-r:** Este argumento le indica a Webber que debe recorrer los directorios fuente de forma recursiva, empezando en el directorio que se le ha indicado en la línea de comandos. Por defecto, cuando Webber va a procesar un directorio, sólo procesará los ficheros incluidos en él, ignorando los subdirectorios que pueda contener; si habilitamos esta opción procesará todos los subdirectorios que contenga el directorio fuente.

9.3 Manual de usuario de Xwebber

Xwebber es un entorno gráfico simplificado para Webber. Simplificado porque desde él perdemos cierta funcionalidad de Webber, como el procesamiento de directorios o de múltiples ficheros y directorios simultáneamente. Desde Xwebber sólo podemos utilizar a Webber para procesar un único fichero fuente de entrada, pero a cambio recibimos otras ventajas, además del hecho de utilizar un interface gráfico de usuario, que Webber por sí sólo no nos ofrece.

Cuando ejecutamos Xwebber para realizar el tratamiento de un fichero fuente, en su interface gráfica observamos información que Webber no nos ofrece, como por ejemplo la lista ordenada de clases heredadas por el fichero fuente. También nos suministra un editor de texto para poder editar el fichero que seleccionemos de esa lista, y realizar las operaciones básicas de edición sobre él: cortar, pegar, salvar, etc..

También hay que destacar, que tras finalizar la ejecución de Webber, nos muestra en cuadro de texto la salida de errores que se haya generado. Con la ventaja de que podremos salvar su contenido en el fichero que deseemos, para poder realizar comprobaciones posteriores o simplemente como fichero de *log*.

A pesar de las ventajas que Xwebber nos llega a ofrecer, se podría llegar a mejorar mucho, para que desde él sigamos teniendo toda la potencia que Webber nos ofrece añadiéndole un interface gráfico intuitivo y fácil de utilizar.

9.3.1 Ejecución de Xwebber

Una vez iniciadas las X-Windows en nuestro sistema y desde un terminal podemos invocar Xwebber de la siguiente forma:

\$ Xwebber Fichero

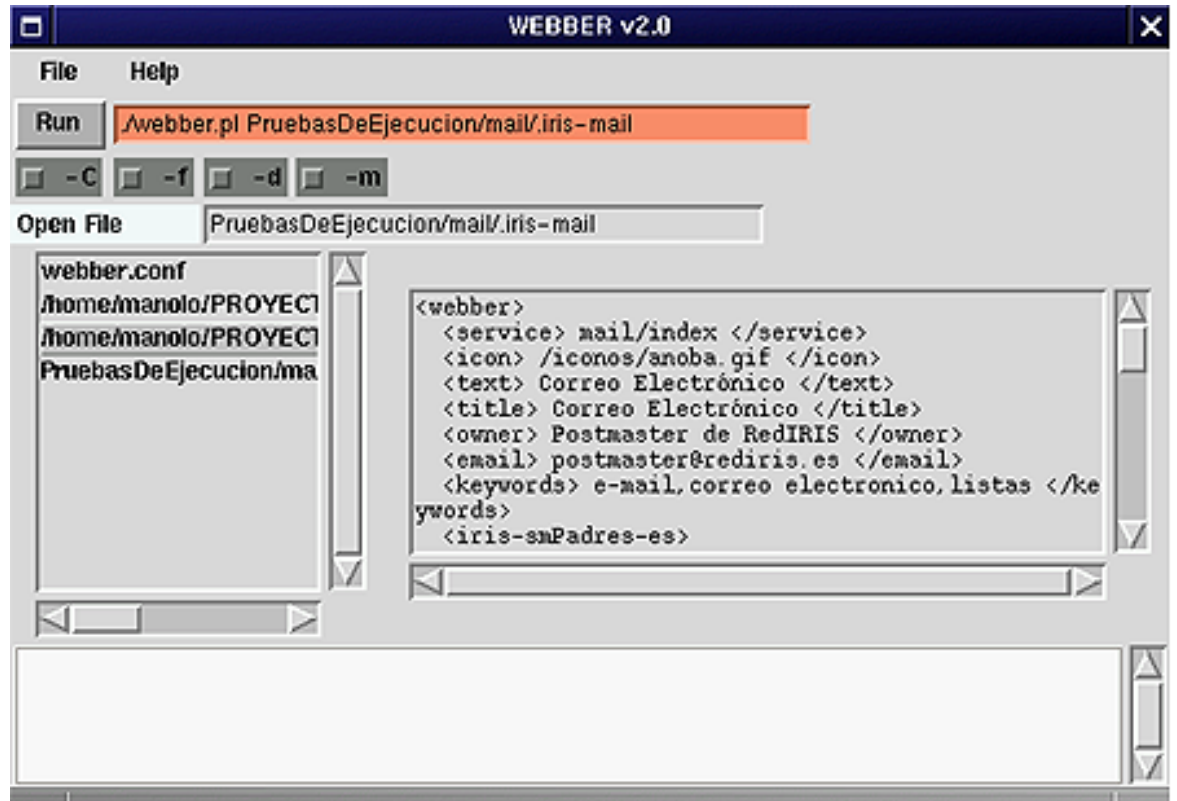
donde el argumento **Fichero** es el nombre del fichero fuente con el que queremos ejecutar webber. Este argumento es obligatorio suministrárselo a Xwebber para que comience su ejecución; en el caso de que lo invoquemos sin suministrarle este argumento en nuestro terminal recibiremos el siguiente aviso:

No hay fichero de entrada.

Una vez que hemos aportado un fichero de entrada a Xwebber vamos a pasar a describir el manejo de este interface gráfico, pero antes se avisa de que para que esta aplicación funcione adecuadamente debe ubicarse en el mismo directorio que Webber.

9.3.2 Manejo del interface gráfico

Si invocamos a Xwebber adecuadamente, es decir suministrándole un fichero fuente como argumento nos aparecerá una ventana que representa al programa y que tiene el siguiente aspecto:

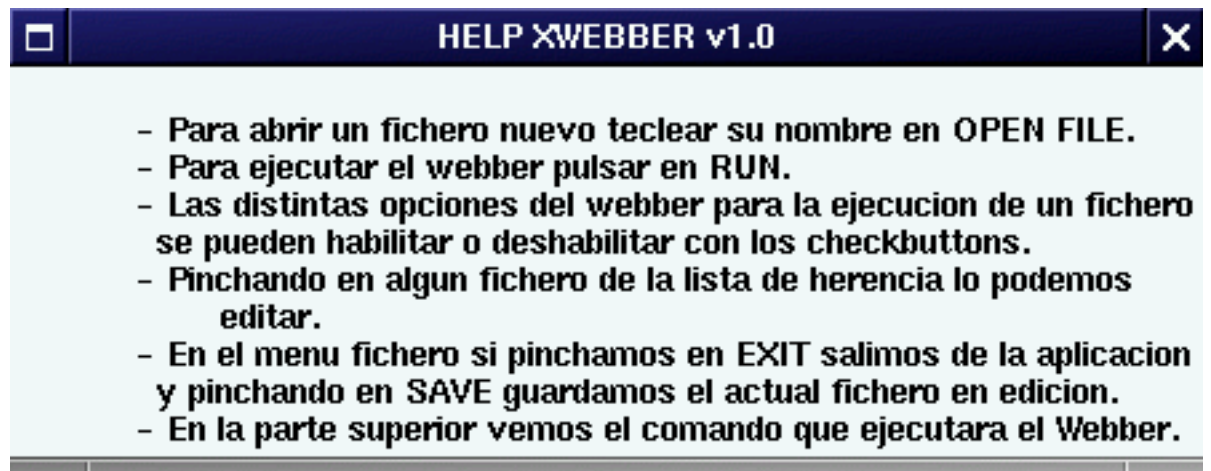


Veamos a continuación la función de cada uno de los widgets que componen la ventana principal de la aplicación.

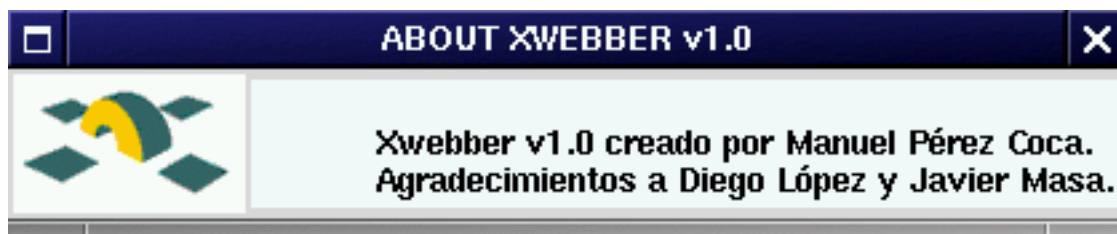
Menú

En la parte superior de la ventana tenemos un menú compuesto a su vez de dos submenús desplegables a su vez:

- El primero de ellos y etiquetado como **File** nos permite salvar el fichero que actualmente se está editando en el editor de texto en la opción **Save**, o bien salir de la aplicación con la opción **Exit**.
- El segundo de ellos denominado **Help**, nos permite ver una breve ayuda sobre el manejo de esta aplicación pinchando en la opción **Help**:



Y también algo de información sobre la aplicación como el autor, colaboradores y versión, pinchando en la opción About.



Ejecución de Webber

La segunda fila de widgets que se puede apreciar en la ventana, está compuesta por un botón y un cuadro de texto de una sola fila. El botón etiquetado con RUN lo pulsaremos cuando queramos lanzar la ejecución de Webber, teniendo presente que el comando que será ejecutado es el que podemos visualizar en el cuadro de texto nombrado anteriormente.

El cuadro de texto que contiene el comando de Webber que se ejecutará puede ser editado por el usuario pinchando con el ratón sobre él. Además mediante los *checkboxes* que explicaremos a continuación podemos añadir o quitar argumentos con los que se desee invocar a Webber.

Cuando finaliza la ejecución de Webber la aplicación nos avisa con un pitido.

Argumentos de Webber

La segunda fila de widgets está formada por cuatro *checkboxes*, cada uno representando uno de los posibles argumentos que puede recibir Webber cuando se ejecuta únicamente sobre un sólo fichero fuente, que recordamos es una de las limitaciones que esta herramienta presenta.

Si habilitamos un **checkbox**, el argumento con el se corresponde se visualiza en el comando con el que se invocará a Webber y que está representado en el cuadro de texto de la fila superior. Si lo deshabilitamos, automáticamente dejará de ser visible en dicho cuadro de texto.

Si habilitamos la opción **-C**, debemo pinchar en el cuadro de texto que representa el comando que se va a ejecutar, para añadir a continuación del argumento **-C**, el nombre del fichero de configuración utilizado. Cuando deshabilitemos esta opción, desaparecerá del cuadro de texto tanto la opción **-C** como el fichero que hemos añadido.

Información del fichero fuente

La tercera fila de widgets, está formado fundamentalmente por un cuadro de texto de una sólo fila, etiqueta a su derecha como **Open File**, que contiene el nombre del fichero fuente con el que ha sido invocado Xwebber.

Si queremos cambiar este fichero fuente que Webber procesará pinchamos en este cuadro de texto y lo editamos indicando el nombre del nuevo fichero. Cuando finalizamos su escritura debemos pulsar **Enter**:

- Si estábamos editando algún fichero, se nos preguntará con la siguiente ventana si queremos salvarlo antes de que se cierre el fichero fuente actual.



- Además se actualiza el nombre del fichero fuente en el cuadro de texto que contiene el comando que se va a ejecutar.
- Y por último se actualiza la lista de las clases heredadas por el nuevo fichero.

La cuarta fila de widgets también muestra información sobre el fichero fuente actual. El primer widget es una lista ordenada con los nombres de las clases heredadas por el fichero fuente. Si pinchamos en cualquiera de ellos podremos editarlo en el siguiente widget, que es un editor de textos. Si en ese momento estábamos editando otro fichero y habíamos realizado modificaciones sobre él, se nos preguntará antes de cerrarlo si queremos almacenar los nuevos cambios.

9.3.3 Información de depuración

La última fila está formada por un único widget, un cuadro de texto en el que se muestra, al finalizar la ejecución de Webber, toda la información de depuración, en el caso de que hayamos habilitado el argumento `-d`, que Webber suministra durante su ejecución.

Este cuadro de texto no es editable por el usuario, pero hay gestores de ventanas de X-Windows que si nos permiten realizar todo tipo de operaciones de edición sobre él: salvar en un fichero, copiar, etc.. De esta forma podríamos crear ficheros de log de las distintas ejecuciones de Webber que realicemos.